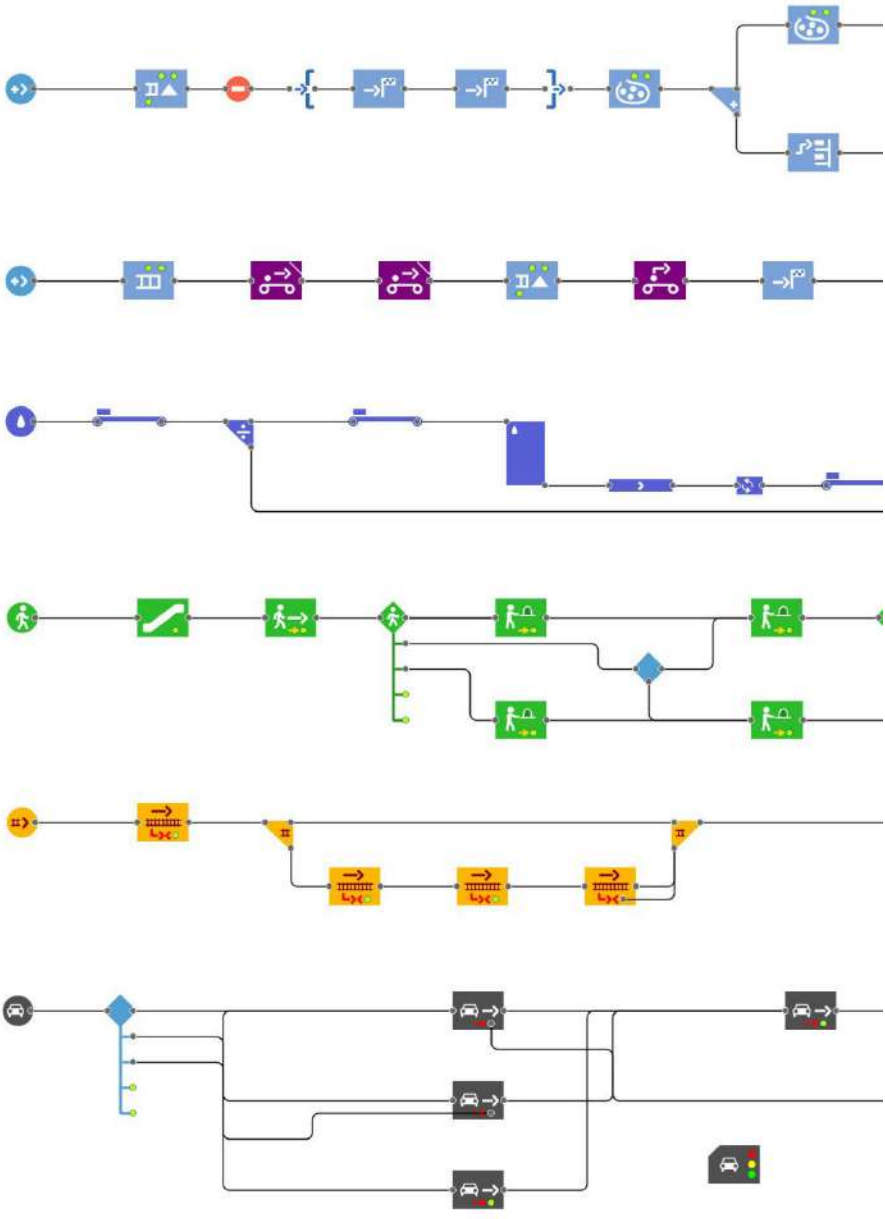
The Art of Process-Centric Modeling

with AnyLogic



Foreword

One of my great joys is the opportunity to speak with former students who have come to understand simulation modeling's vast potential. Many of them who adopt it at work reach out to me months, even years, later to ask me how so many organizations can overlook it.

I think the main issue is the learning curve. It isn't easy to master a discipline which draws on elements of design, imagination and reimagination, philosophy, computer programing, mathematics and statistics. And even after one comes to understand simulation modeling concepts and essential prerequisites, you still need to understand the many modeling paradigms we can use to solve specific problems.

Today, I'm pleased to share this book with you. It covers the theoretical concepts, implementation techniques, mathematical and statistical backgrounds you expect. But it also touches on other things in and around simulation modeling. You'll find most of it is about process-centric models based on flow systems, which many of you know as discrete-event models. I also look at state-of-the-art designs that use elements of agent-based modeling. While I don't cover agent-based modeling on its own, I discuss multi-method models and the advantages they offer.

It is suggested that you read this book like a story, chapter-by-chapter in a continuous fashion. As you move along the chapters, more complex concepts and designs will be introduced to you. In each chapter, you may be referred to an appendix if it contains relevant prerequisite information. You may also treat the appendices as mini-chapters and read them on their entirely if you want to learn about, or refresh your memory on, the topic covered on that appendix.

I would like to thank:

John Yedinak, Managing Director of AnyLogic North America, who first recognized my passion for simulation and AnyLogic software. His mentorship, support, and encouragement made this book possible.

Andrei Borshchev, AnyLogic co-founder, CEO, and the simulation mastermind. I learned invaluable modeling techniques and concepts from Andrei indirectly via "The Big Book of Simulation Modeling" and directly in the workshops and seminars we've done together. His vision, genius, and elegant worldview shines through AnyLogic's software.

Nikolay Churkov, head of AnyLogic Development, for his continuous stream of new features and capabilities which make AnyLogic even more amazing. Nikolay and his development team will make it difficult to keep this book up-to-date!

Ilya Grigoryev, head of Training Services at The AnyLogic Company, for the fantastic AnyLogic help. His body of work in AnyLogic Help accounts for much of this book references.

Pavel Lebedev, Gregory Monakhov, and **Tatiana Gomzina,** the AnyLogic Support team, who patiently answered all my questions and taught me many techniques and features throughout the years.

Tyler Wolfe-Adam, for his months of reading, editing, and suggesting improvements. I can't imagine writing this book without his help. You will hear more about him as he is one of the most talented newcomers in the field of simulation modeling.

Ted Engel for his meticulous editing and all his work in preparing the book for publication. He had a great impact on the quality of this book.

This book is based on my experience teaching simulation to hundreds of consultants, engineers, and researchers who work in world-class consulting firms, Fortune 100 companies, and renowned research universities. I hope this book helps you in your simulation modeling journey.

Arash Mahdavi

Objectives of this textbook and author's requests

Prominent scholars have written many fantastic books on discrete-event simulation. This book tries to add to the body of knowledge by:

1. Using the most popular simulation modeling tool to revisit legacy topics from a fresh perspective.
2. Explaining the abstraction aspect of modeling that other resources often neglect.
3. Expanding the process of describing to include the thought process ("WHY") and the techniques and procedures ("HOW").
4. Replacing unnecessary explanations with information that helps brings focus, clarity, and beautiful simplicity (we often call this elegance!) to the covered topics.
5. Showing state-of-the-art, event-driven modeling techniques for process-centric models by:
    a. Adding state-oriented modeling constructs (statecharts)
    b. Incorporating agent-based modeling (and its vast capabilities)
6. Adding clarity and intuitiveness to learning new simulation-related concepts

This book tries to explain subjects others tried to evade – something they have good reason for doing! Since the model building process is as much an art as it is a science, explaining it well is difficult.

Modelers with artistic taste may suggest better alternatives for the architecture of example models. Conversely, scientific purists might think provided explanations are incomplete. They may expect better mathematical explanations, rigid definitions or persuasive references.

I humbly accept all the above and acknowledge their statements have some merit. However, it takes compromise to teach a complex subject. My only request is you think of this manuscript as an open-source project we'll complete together and share your comments and suggestions. Together, we can make the fascinating world of simulation accessible to a larger group of students and enthusiasts.

Chapter 1 : Fundamental Ideas and Key Concepts

What is a model?

A **symbol** is something that represents something else. We humans need symbols to simplify and expedite communication. The symbol isn't the subject, it stands for something else. Instead of re-describing all the tangible and intangible attributes or behaviors of *something* every time, we agree on a common symbol and use it instead. This means a symbol can be a sign, a word or name, or a physical object.

However, a symbol is a black box. Somehow, we have a general knowledge of what it describes, but not in its entirely. Apart from the fact that a symbol can represent something intangible and abstract (e.g., love, fear, death), it's still very vague for even tangible and common physical things. For example, "car" is word that symbolizes a vehicle that moves on four wheels. However, it still leaves most of its attributes and behaviors undefined. So, using a symbol instead of what it stands for can simplify the communication, but isn't good enough for pin-pointing specifics of the actual object. Regardless of this downside, symbols will keep us from getting stuck in infinite loops of redefining concepts. Instead we use it as an inherent tool in our toolbox to deal with our sensory inputs.

Aside from describing and communicating what something is, we need to describe *how* that thing works. In this case, a **Model** represents how something works. Putting together a model is a completely different problem compared to coming up with a symbol. When several people agree on a definition, they can replace that definition with a symbol. Afterward, they can use that symbol to simplify their communication. But when you want to communicate how something works, you're dealing with a question you need to answer before you can communicate it.

Let's say I want to help others understand how a car works. No matter how much detail my answer offers (the model), I need to examine the subject and find a logical answer. My answer might be a generalization: "A car works when its engine operates". I could also provide a detailed blueprint that shows every nut and bolt. However, I can't avoid answering the question. I need to offer more than an accepted definition.

Understanding how something works may well be more important than defining it. Basically, a good model of something could also be its black boxed symbolic definition (or a substitute for it). For example, if I can give you a good map of how a car works, we could consider the map as the definition.

Knowing how something works (being able to build a model from it) has important benefits. It allows us to correctly interact with the object. Our interactions with a car can help us learn how to drive it and change the oil. It also helps us predict its future behaviors and states: We'll learn a car won't run after a serious crash. Most important, it allows us to reproduce (or regenerate) the car. That information might also allow us to create an improvement such as a new car or a better exhaust pipe.

Our ability to build models was as important as symbols in our evolutionary journey, and arguably even more important for our survival. It's hard to argue symbols didn't play an important role in our evolution. They helped us communicate thoughts and ideas. The knowledge we gained from them enhanced the individual and collective capabilities of humanity.

I'm sure many of you question my claim about the evolutionary necessity of models. There are different types of models: conceptual models (for example, organizational charts), mathematical models (for example, probability models, differential equations), physical models (for example, wind tunnels), simulation models, etc. None of them could be possibly attributed to homo-sapiens who lived tens of

thousands of years ago. Does that mean humans only began to use models within the past few centuries? The answer is a resounding no!

Mental models are another important category. Scientists attribute our superiority to other species to our ability to anticipate and plan for events. As you'll see, we can think of this as our ability to build and run mental models.

Our brain uses two types of mental models: inherited (intuitively-based ideas of how essential things work) and learned (experience-based). Our brain is a fascinating machine that continually develops, remembers, and executes these mental models.

Our brain can build a model (that is, understand how something works) and interact with it to anticipate how that object will behave in a variety of scenarios. For example, some of us have mental models of how the macroeconomy affects the housing market. If you're convinced the economy is headed for a downturn, your version of the model might prevent you from paying too much for a property.

We also have mental models for interpersonal relationships. When you empathize with someone, you're allowing a mental model to help you understand their situation and imagine how it would affect you. We even have mental models for trivial things. If you want to lift something, you'll use pre-existing knowledge to estimate its weight, run a mental model imagining the lift, and then try to lift it.

This ability to model things is what separated us from our ancestors and was the main source of our species' success. We'll discuss mental models in more detail later in this chapter.

Now that I've (hopefully) convinced you you're an astute modeler, I hope to introduce you to a specific type of manmade models: simulation models. You should consider all other types of models (mathematical, physical, conceptual, simulation, etc.) as extensions of mental models. These external tools are much like physical tools such as a hammer and chisel in that they help your brain to extend its modeling capabilities. Keep in mind all types of modeling tools and techniques have nothing of value apart from the user's mental model. The external models are delegated parts of a bigger mental model.

In the next section, we'll focus on the definition of and general information about simulation models.

Simulation models

Simulation modeling is a discipline that tries to explain how something works by building a replica of it. Before we go further, let's borrow some of the fundamental definitions from the *DoD Modeling and Simulation Glossary*:

- **Simulation**: A method for implementing a model over time.
- **Simuland**: The system being simulated by a simulation.
- **Simulator**: A device, computer program, or system that performs simulation.
- **Modeling and simulation**: The discipline that comprises the development and/or use of models and simulations.

Compared to other types of models, simulation models are extremely versatile. We can say a simulation model is a set of rules that defines the workings of simuland. A simulator such as AnyLogic allows us to set the model's starting state and use the rules we select to simulate the system's evolution through time. In stochastic (or random) systems and their models, a system's evolution could take many unexpected turns. This means the systems may end up in different final states (Figure 1-1).

One of the main benefits of simulation approach is that we can run our models as often as we see fit. The outputs of our experiments show us the spectrum of possibilities. This is extremely useful for practical decision-making scenarios. These virtual and risk-free experiments can also help us evaluate the advantages and disadvantages of one or many scenarios.

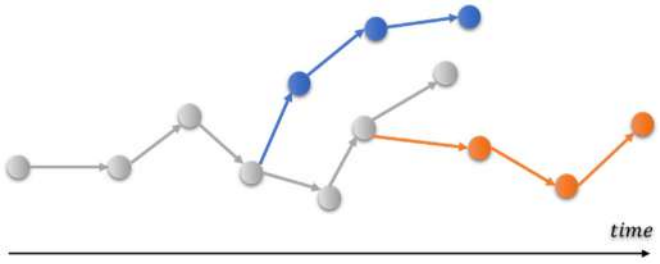


Figure 1-1: Different trajectories of the same system in different runs

Since "simulation" is an ambiguous term – practitioners use it in many contexts – it's useful to review the types and categories of simulation models. Before we start, let's remember this book and the DoD definitions we reviewed above are for *dynamic* computer simulation models. While *static* or *physical* simulation models are outside of our scope, we want to ensure you receive a full overview:

I) **Physical vs Computer**: The replica which represents how something works could be physical or digital (inside a computer). A good example of a physical replica is a wind tunnel simulation made up of a tubular passage with the object we're testing mounted inside. Instead of examining a car or a plane at real scale, we use a smaller model to simulate and measure their aerodynamic characteristics.

Note physical models are outside of this book's scope. On the other hand, a computer model doesn't usually have any physical components. It's a computer program that virtually replicates an object's behaviors and attributes. For example, we could use computer software and rules on how birds align, separate, and move toward one another to simulate how flocks behave.

II) **Static vs Dynamic**: We can set up a model to incorporate or ignore time. A static simulation model doesn't incorporate the passage of time, while a dynamic model does. A static model first performs all the tasks and calculations and then returns the output.

Let's use a model to show how static and dynamic models differ. Imagine a model of a machine that has five buttons. Its task is to follow a prespecified rule as it pushes the buttons.

Static version: The machine's task is to push the buttons based on their label in alphabetical order, and then print the sequence of the pushed buttons. This means a model with this behavior will return a list of buttons it pushed in the order it pushed them: [A,B,C,D,E] (as shown in Figure 1-2). In this example, time didn't play any role in how the machine operates or the model output (at least in the way we setup its behavior).

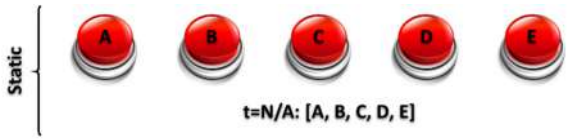


Figure 1-2: Output of the static example model

Dynamic version: The availability of buttons and the button pushing behavior depends on time. Like the static model, the buttons must be pushed in alphabetical order. In this version, the model must wait one second before it presses each button. In addition, not all buttons are active for the entire time: buttons B and D are deactivated after 3 seconds and won't be available afterward. If the machine encounters a deactivated button, it will move to the next one. As you can see, time plays a key role in how the machine (and its model) works. It also has a significant effect on the output.

Based on our rules, at time 1 button A is pushed; at time 2, button B is pushed. At time 3, button C is pushed; at the same time, buttons B and D are deactivated. At time 4, button D isn't available anymore, so the machine moves to the next button and pushes the E. By the end the sequence, the pushed buttons would be [A,B,C,E] (Figure 1-3).

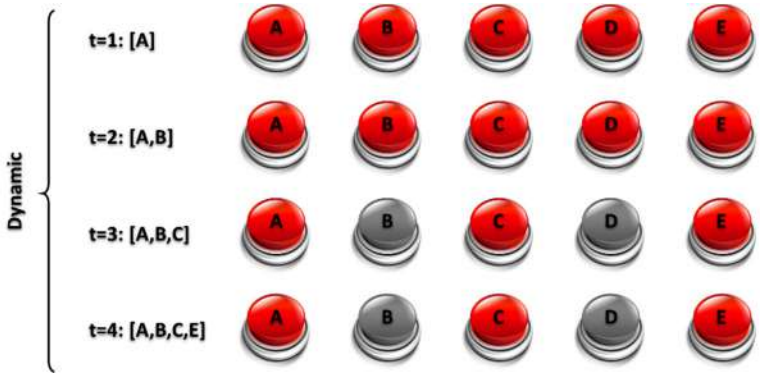


Figure 1-3: Output of the dynamic example model

By adding a time aspect to the machine, we've significantly increased the behavior's complexity. The scope of this book will focus purely on dynamic models, and not static ones.

III) **Deterministic vs Stochastic:** If the simuland works the same way under identical conditions, it's deterministic. But if it exhibits different behavior under identical conditions, it is stochastic (or random).

Think about the imaginary machine. It must wait one second between each button press and it must follow an alphabetical order. In addition, all buttons are always active. Such a behavior is deterministic, as the outputs are the same every time we start the machine.

If we run the machine (more specifically, our model of this machine) and wait for 5 seconds, the results will always be the same: [A,B,C,D,E]. You can see three example runs and their identical results in Figure 1-4.

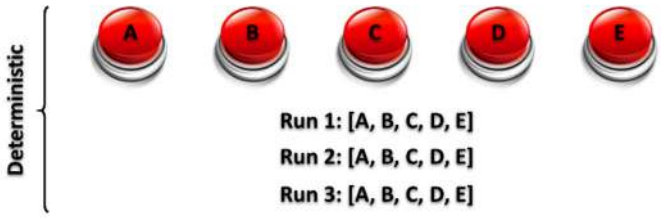


Figure 1-4: A deterministic model run three times

A stochastic model allows rules and behaviors that are random. For example, let's assume the pushing rule is the same as above. But let's make one change: after the machine waits for one second, it doesn't press the next alphabetical button. Instead, it selects a random button that it hasn't pressed.

If we run our model three times and wait for five seconds for it to finish, we'll receive three different sequences of pushed buttons similar to what you see in Figure 1-5.

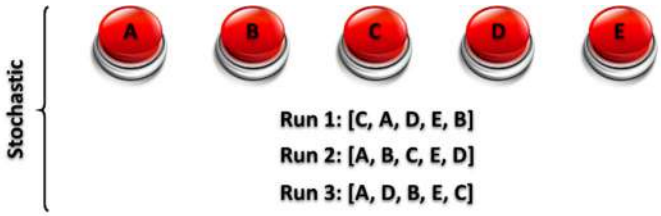


Figure 1-5: A stochastic model run three times

Most realistic systems have elements of randomness. This means it's important to incorporate randomness into a simulation model. In some cases, the reason for the push toward building simulation reflects the inherent randomness inside the simuland.

IV) **Continuous vs Discrete**: We can categorize dynamic simulations into discrete or continuous depending on how we incorporate the effect of time into the model. Without getting overly philosophical, time isn't an independent concept - it depends on change. What we call or experience as "time" is simply continuous change.

Most real systems change continuously, and this means time should continuously move forward. A model of a continuously changing system must effectively simulate the simuland's state at any point in time during the simulated period. For example, imagine simulating something that moves from point A to B, as shown in Figure 1-6. In a continuous movement, we can stop the model at any point and know the moving object's location will be different from a moment before it.

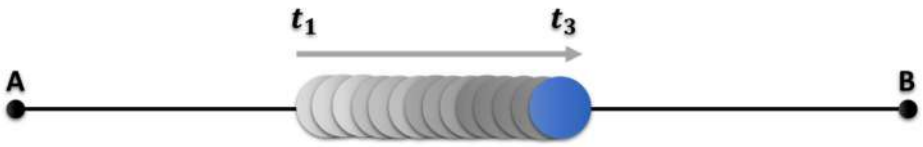


Figure 1-6: Illustration of continuous movement

A discrete model, where change happens only at specific moments or intervals, is a good contrast. Using the previous example, a discrete movement would be much like jumping from one point to the next in each step, as shown in Figure 1-7.

In a discrete model, nothing happens between two events (where events are subjectively selected noteworthy happenings). We know real systems rarely jump to their next state. However, if the interim states are trivial or we could ignore them, we can use a discrete approach to simplify the model.

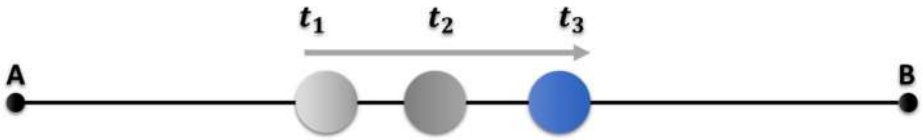


Figure 1-7: Illustration of discrete movement

We won't cover all categories of simulation models we discuss introduced in this section. Instead, we'll focus on dynamic computer models. In AnyLogic, you can build deterministic or stochastic dynamics, and those models could be discrete, continuous, or a combination of both. Our focus is simulation models which are *dynamic, discrete, and stochastic*. As an aside, you can consider deterministic models as a subset of stochastic models (when random parts are fixed). This means we're also covering deterministic models.

Different types of simulation methods

Now that we know more about simulation models, we need to discuss simulation **methods**. In simulation modeling, a method is a general framework for mapping a real-world system to its model (Borshchev, 2013). You might think of it as a language that allows you to communicate the simuland's inner workings with the simulator. Of course, simulation methods weren't all created at the same time. They also weren't created with the same type of simulands in mind. That's why each simulation method works better for a specific scenario.

Describing these methods to someone who doesn't have simulation modeling experience can be a challenge. It's like trying to describe the difference between driving an automobile with a manual transmission and one with an automatic transmission to someone who has never driven. The best way to learn each method's features and the differences between them is by gaining experience.

In later chapters, we'll describe each method in detail and review a series of examples. To give you a preview of these methods, we'll briefly describe them here. For an excellent explanation of these methods, consider the *Big Book of Simulation Modeling* (Andrei Borshchev, 2013).

The three main methods of dynamic simulation modeling are:

1) **System Dynamics**

   Created in the 1950s by MIT professor Jay Forrester, system dynamics is noticeably different from the other methods. It's best known for its high abstraction level (fewer details), typically being deterministic, and its continuous modeling of time. Under the hood, it's a system of coupled first-order differential equations solved with numerical methods. Since we don't cover system dynamics in this book, we'll leave the details for another time.

2) **Process-Centric Modeling (Discrete-Event Modeling/Simulation)**

   This method focuses on systems in which a sequence of operations or tasks needs to be performed. This means we can think of these operations as a process and illustrate them with a flowchart.

   In this method, the study looks at the things that pass through the process and the resources that help them complete their journey. We often see these types of scenarios in manmade systems. The reason? It's usually easier to break large and complex operations into multiple steps. Process-centric models are dynamic, discrete, and mostly stochastic. We also cover deterministic cases, a subset of stochastic models.

   Process-centric models, what many call "discrete-event modeling/simulation", have remained essentially the same since the 1960s. In chapter 2, we'll review queueing systems which are probability models of systems we can simulate with process-centric models. Reviewing those mathematical models alongside their process-centric model counterparts will help us better understand this method's origins. In chapter 3, we'll review and build several state-of-the-practice process-centric models. We save the new advances in this field for chapter 4, where we add agent-based modeling to the mix.

3) Agent-Based Modeling

This methodology is made up of subsystems that may work independently or collaborate. The simuland is often a system without a coherent centrally-directed behavior, and its behavior emerges from the collective behavior of subsystems which may or may not be interconnected. Much like how these systems came to exist in the real world, we can model them with a bottom-up approach that models the subsystems first.

Agent-based modeling is the most recent of the three methods. It's based on object-oriented modeling in computer science. By borrowing some ideas from object-oriented modeling, agent-based modeling lets us encapsulate attributes and behaviors inside subjectively separated sub-models (or agents, in its terminology). We could use System Dynamics, process-centric modeling, or statecharts to model the actions that take place inside agents.

We usually don't treat statecharts – visual constructs that let you build event-driven models that focus on the simuland's states – as a separate and fundamental modeling method (like process-centric or agent based). Nonetheless, statecharts are a powerful method for building state-based models or sub-models. If we don't use System Dynamics to model the inner workings of agent-based models and limit it to process-centric modeling and statecharts, we can categorize the agent-based models as dynamic, *discrete*, and stochastic (which includes deterministic). We'll review the basics of agent-based modeling and statecharts in chapter 4.

Multi-Method Modeling (Hybrid Simulation/Modeling)

As you might have guessed, multi-method modeling is a mix of abovementioned modeling paradigms. There's a significant synergy between different simulation methods, and this means the capabilities of multi-method models are greater than the sum of their parts. Many people use "hybrid modeling" interchangeably with multi-method modeling. But you should always check the definition provided in the specific context.

AnyLogic software is the champion of the multi-method modeling approach and the only platform that allows you to build truly unrestricted multi-method models. There are many architectures (mixes of simulation methods) possible for multi-method modeling. However, this book's emphasis is on process-centric models. That means we'll focus on multi-method architectures that improve process-centric models, such as processes inside agents and processed with entities and resource units that are agents too.

It's almost impossible to grasp the scope of the architectures we named here without prior experience. If you're new to this concept, be patient. You'll gain a better understanding as you learn the state-of-the-art process-centric modeling from the multi-method approach in chapter 4.

What is model abstraction?

Abstraction is arguably the most important piece of the puzzle in building a simulation model. It is the process of converting some reality (or a perception of a reality) into a representation of one. It is a deep concept and is the fundamental thought process that a model is built upon! To better understand abstraction in a simulation model, I'll start by putting it in the context of how we perceive the world; through that, you'll realize that a simulation model is just the tip of an iceberg in how we're abstracting away from the objective reality.

Let's start by recognizing we're all modelers. The objective reality that surrounds us is much more complex than our neural networks - our brains - can effectively process. That's why we must resort to abstraction to induce something meaningful from the sensory inputs we receive.

We do this by using two types of abstraction to build a model of the external world (Figure 1-8):

1. **First-degree abstraction**: This is our brain's transformation of the external world into our biological perception system.
2. **Second-degree abstraction**: This is the process of transforming the first-degree abstraction (that is human perception) and casting it into a manmade perception system.

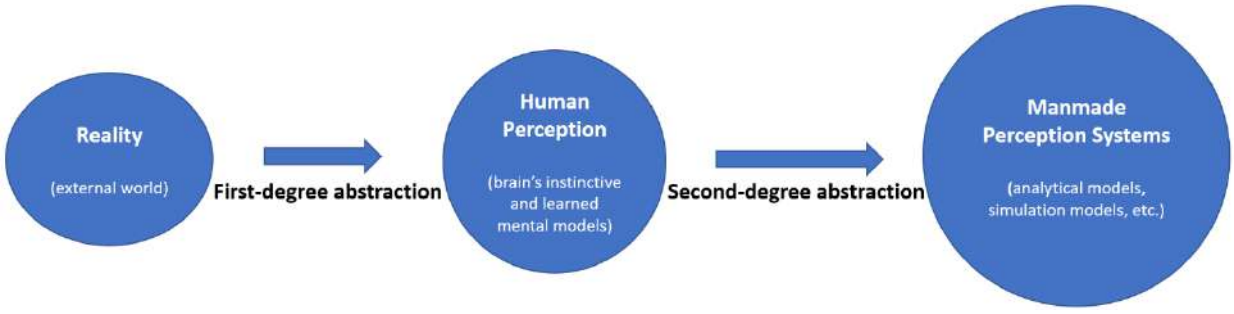


Figure 1-8: Two degrees of abstraction to build a model of the external world

First-degree abstraction

In first-degree abstraction, our brain uses two types of mental models to make sense of sensory input data:

1. **Inherited/Instinctive mental models** are hard-wired in your brain
2. **Learned mental models** are formed and developed from the outcome of prior (similar) experiences

Both mental model types are high-level. This simply means they receive and respond to data based on a broad and simplified representation of the system.

Imagine you're shopping for a formal occasion. When a red coat catches your eye, your inherited mental model tells you red is often associated with passion, danger, and aggression. At the same time, your learned mental model tells you red isn't appropriate for formal attire. After you've run the input (a red coat) through your mental models, you decide to not buy it.

In this process, your brain limits the input it receives by disregarding low-level information such as the color of the coat's inner lining, the intricate weaving pattern in the fabric, or the number of pockets. This in turn reduces the complexity of its model. To reach a decision, our brains only choose partial sensory inputs to run through the limited number of internal models.

The important caveat here is that both the sensory inputs and our mental models are incomplete. Our sensory inputs are limited because our senses are limited. What we sense is just part of the reality which produces infinite streams of ever-changing data. In fact, our brains even trim down this limited data into a focused selection based on what it needs for a specific mental model (for example, scanning the fabric's color rather than its abrasiveness). At the same time, our instinctive and learned mental models are much simpler than the actual systems they try to analyze.

This lengthy description of first-degree abstraction was necessary to establish the fact that our base perception of reality at its simplest form (hearing a name, seeing a color, etc.) is subjective, simplified, and imperfect. Knowing this will help us better understand our more conscious abstraction effort in modeling the world with manmade models (analytical models, simulation models, etc.).

Second-degree abstraction

Human beings have always looked for ways to move beyond the limitations of their mental models. We learned how to use external methods and tools to gain insight into concepts and systems that we don't understand intuitively. Some examples include: mathematical models depicting growth of a population, analytical models comparing the rise and fall of a products demand, and flowcharts that show the thought process in making some decision. In these types of manmade models, our perception of reality is further simplified into models representative of the system (or at least, what we perceive the system to be).

Analogous to our physical tools that enhance our interaction with the world at physical level (for example axe, arrow, forklift, car), manmade models are soft tools that help our brain to better understand the universe. Because of their narrow scope, these manmade models can only assist the brain to concentrate its limited power in deciphering the complexity of a specific problem.

Abstraction level

Now that we've explored the two degrees of abstraction, we'll discuss how abstraction works in simulation models. Let's start with a few basic concepts:

- **Abstraction** in this context is the transformation of the modeler's perceived reality of a system into a representative model that can emulate that system's dynamic behavior.
- **Abstraction level** is the amount of information and detail we subjectively include inside the model scope. Can be thought of like the zoom level in a camera - where a low abstraction level equates to being more zoomed in (and thus has the most detailed view)

To better understand the sliding scale of abstraction level, look at the three pictures below (Figure 1-9). How are they alike? How do they differ?

Figure 1-9: Abstraction level from low to high (left to right)

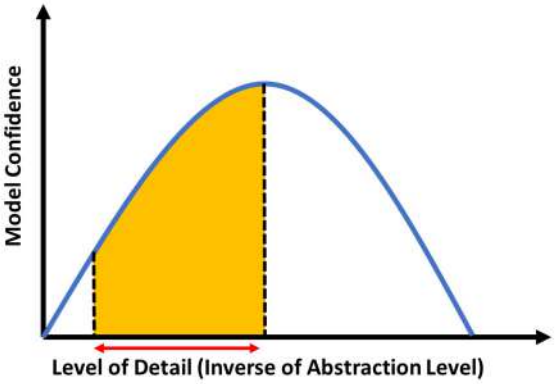
None of the pictures above is a true representation of Lisa Gherardini! A photo would have had more detail and been closer to her actual appearance; however, that's still only a 2D representation of her.

As you can see, Leonardo da Vinci's painting is an imperfect abstraction of reality. The sketch in the middle has even less detail, and the smiley face on the right has little in common with the painting. If our goal was just to convey a smiling face, did we really need to paint a Mona Lisa?

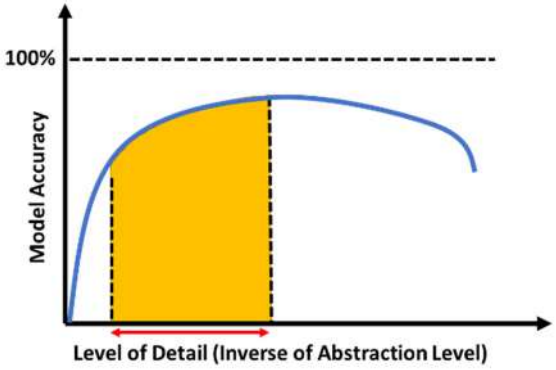
A similar thought process goes into the model abstraction. One of the first decisions we must make before we build a simulation model is what abstraction level to use (or in other words, the level of detail that should go into the model). However, we shouldn't assume a lower level of abstraction (that is, more detail) is always desirable.

Before we compare abstraction levels, let's define two terms (defined in relation to model building):

1. **Confidence** - a subjective feeling or belief that the model's results are reliable
2. **Accuracy** - the extent to which a simulation output agrees with the actual observed outcomes



Model complexity versus model confidence, Lobao and Porto (1997)



Model complexity versus accuracy, Robinson (2004)

Figure 1-10: The level of detail increases from left to right (abstraction level decrease from left to right), with desirable levels in yellow

These two metrics can be evaluated against the level of complexity, which is usually a direct implication of abstraction level. Up to a certain point, our decision to increase the level of detail (that is, to lower the abstraction level) increases our confidence in the model and its accuracy. However, a model can have too much detail. If we reach the point where that added complexity prevents us from reasonably validating and verifying it, our confidence in the model's performance and accuracy will start to decline (Figure 1-10). The range of the desirable level of detail (shown by the red line below the yellow area in Figure 1-10) starts at the point that model accuracy reaches the minimum acceptable level and ends at the inflection point.

Although all the models in the acceptable range meet the required specifications, the simplest acceptable model is usually the best choice. It takes less time to develop simple models. They're easier to update and they require less data. In many cases, the model's *utility*, or its usefulness in making a specific decision, doesn't require maximum accuracy. Our Mona Lisa likenesses help illustrate this point. If our goal is simply to show users a smile, a smiley face will work well. But if we want to offer a more accurate sense of Lisa Gherardini's beauty, a painting is the better choice.

Our choice of modeling method also attributes to the level of abstraction. As discussed before there are three main simulation methods: process centric modeling, System Dynamics, and agent-based modeling. Each is well-suited for abstracting specific types of systems. A well-chosen method results in an elegant model structure that enhances the modeler's focus and helps both the modeler and the user understand the system's inner working with a better clarity. Using a combination of simulation methods in a single model – what we call "multi-method modeling" – usually results in less workarounds and a more elegant model structure. Today, AnyLogic is the *only* simulation software which provides a *true* multi-method modeling platform.

It's important to mention that the added complexity that comes from selecting a needlessly low abstraction level won't necessarily cause problems during the model building phase. However, the increased levels of effort often arise during the verification and validation phases of model development. That is why experienced modelers are usually more hesitant in lowering the level of abstraction - they're more aware of the hardships they will face down the road!

How to make sure our models are right?

Building a simulation model that runs and produces outputs is not the end of modeling development. In most cases when effort is put into building a simulation model of a system, it's indicative of an important decision process taking place (and thus is possibly a critical component in the plan of a business or operation). In this section, we review two main steps that each model should go through to increase its confidence in its representativeness: verification and validation.

Initially, the explanation provided is at a purely conceptual level that seeks to clarify the philosophy behind verification and validation. The implementation process of the two steps will be expounded upon in later sections.

Verification: *In this step, we compare the model's performance to the model's specifications*. Bear in mind that our model's specification may not be complete, representative, or even correct. However, our focus in this step is on the consistency of the built model and its description in the specification.

Let's consider an example. If our model's specification says a truck delivers packages to point A at 8:00 am and to point B at 10:00 am, we need to run the model to verify these delivery times and locations. It is not part of the verification phase to review the overall operation and judge whether the 8:00 or 10:00 am delivery times are realistic.

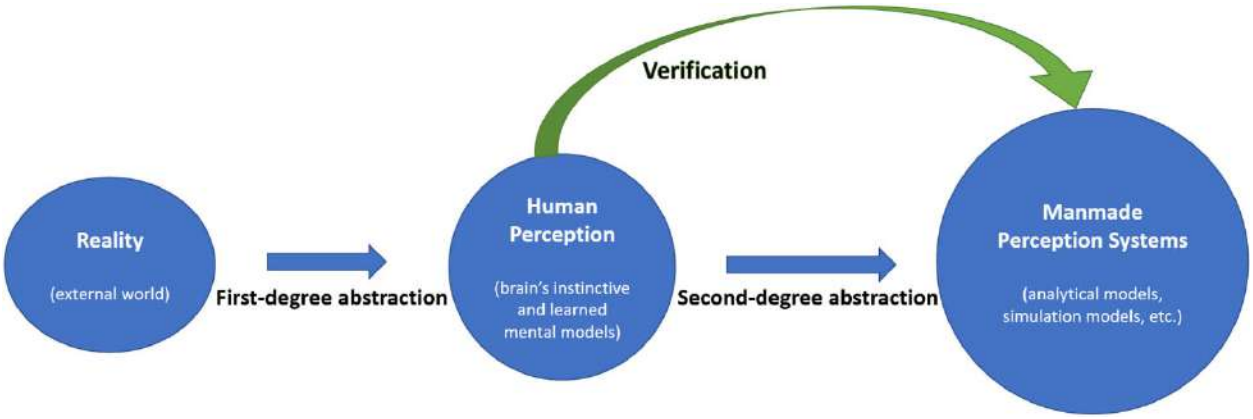


Figure 1-11: verification is checking the relationship between human perception and manmade perception

In the context of model abstraction, we can say that verification checks the link between the human perception and the manmade perception *(*Figure 1-11*).* Since verification only covers the second level abstraction, both perceptions of reality came from one person (or a team) and therefore expected to be consistent. In other words, any discrepancy between the simulation model (the manmade system) and the model specification (human perception) comes from human error or inadequacy in the methods used.

In case the discrepancy is due to human mistakes, it must be corrected. However, if the disagreements reflect a methodological shortcoming, we can change our modeling approach or accept the resulting inaccuracies.

Validation: After verifying the model and being confident that the model was built correctly, it's time to check if the correct model was built! In this step, focus is shifted from the specifications to the actual system.

Let's assume you need to model a production line in a car factory. You use the specifications you receive from a line manager to build a detailed model that works well. But there's a problem: the simulation's daily throughput is 20% greater than the actual line's throughput.

Up to this point you've trusted your understanding of the system's behavior – an understanding you based on the information you received from the domain expert. However, the discrepancy is a sign you should question your understanding of the real system. You may find your observations of the line's operation don't match the line manager's expectations. The employees might not start at exactly 8:00 am. Or perhaps maybe not all forklifts are consistently operational throughout the day, etc.

Defined in the abstraction context: *validation checks the link between the reality and manmade perception, but any uncovered discrepancy during validation is originated from the link between reality and human perception.* The confusing (and sometimes misleading) thing about the validation step is that while we're comparing our model with reality, to correctly solve the issues we should examine our mental models more than the simulation model (Figure 1-12). In other words, the actual flaw is in first level perception: if our perceptions aren't correct, we can't validate our model. This makes validation much more complicated than verification.

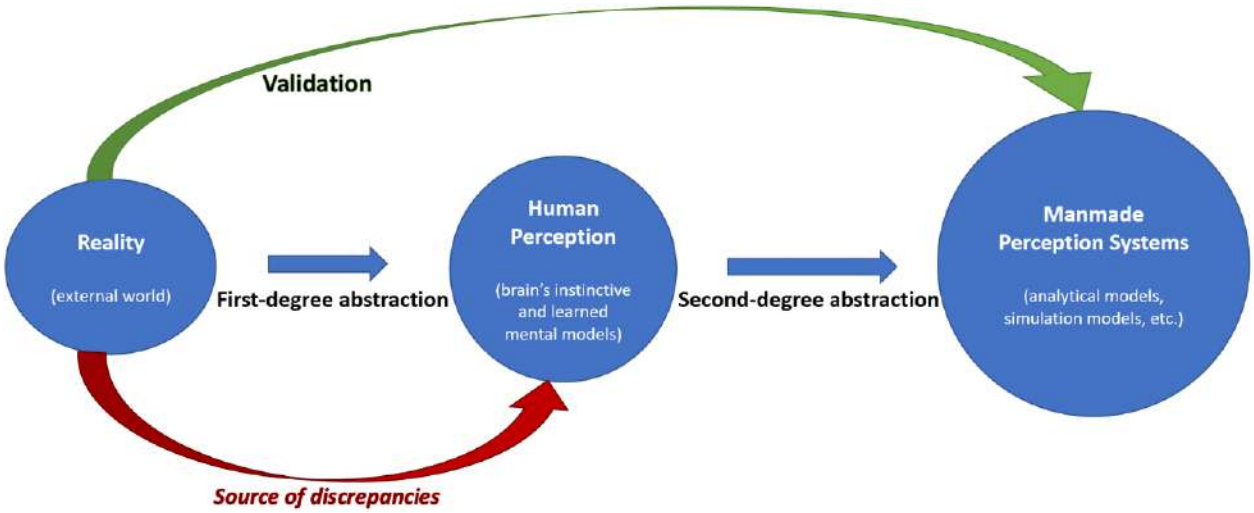


Figure 1-12: Validation is checking the relationship between the reality and manmade perception system, with problems stemming from our perception of reality

In the validation phase, it is common to compare historical data with the simulation output to verify the model. Historical data is used as a baseline to establish that the simulation's outputs are consistent with the observed reality. Showing that the simulation output and historical data are consistent increases our confidence in the simulation model's assumptions and behaviors. However, we should always remember that human perception will never be perfect.

> *"Any 'objective' model-validation procedure rests eventually at some lower level on a judgment or faith that either the procedure or its goals are acceptable without objective proof" Forrester (1961)*

Knowing all these limitations, our goal as modelers is to maximize the usefulness of the model. Whenever you observe discrepancy between the simulation model and the reality, a learning feedback loop will begin; you should embrace that as an opportunity to learn more about the system under study and not

as a problem to resolve. The whole point of building a simulation model is to check and validate your mental models. If you had sufficiently known the system without your model, there would have been no need to take the extra effort to model it.

As previously mentioned, we'll get back to the topic of verification and validation in a more methodical fashion in later sections; our objective here was simply to introduce you to the underlying thought processes.

Modeling our reality as a System

Before diving into how simulation can help us build representative models, we need to review three key concepts: system, flow system, and process.

The most generic frame of reference that can help us conceptualize a reality is a **system**.

> *"A system is a set of elements in interaction" (Ludwig von Bertalanffy, 1968)*

To paraphrase Ludwig von Bertalanffy, a system is a combination of elements or components that, at some level, are interacting and together forming a whole. The understanding of a system starts with the notion of a **system boundary**, "a distinction made by an observer which marks the difference between an entity he takes to be a system and its environment" (Checkland 1999).

The setting of a boundary, and hence the identification of a system, is manmade and ultimately the choice of the observer. The fact that any identification of a system is a human construct is because in reality, everything is connected! It is the limitation of the observer that forces them to draw the imaginary boundary and reduce the complexity of the reality. This in turn makes the system easier to study and analyze. When the observer (or the modeler in our case) draws the boundary around some interactive elements to construct a system, they separate those elements from the rest of the universe - a manifestation of the first-degree abstraction we discussed before.

Consider the subjective viewpoint of a person who wants to better understand the production capacity of a factory. They draw an imaginary line around that factory and focus on what happens inside the factory boundary. In rare cases where we're dealing with something very abstract or theoretical, the elements inside the boundary are self-contained (**closed systems**). However, in almost all practical cases this imaginary boundary does not include everything that interacts with the bounded elements (**open systems**). The elements that are outside the boundary but still interacting with the bounded elements are referred to as the environment. Therefore, the boundary of an open system sets apart the elements and relationships that are part of the system from elements in the environment (Figure 1-13). In an open system, there are interactions across the boundary between the system elements (inside the boundary) and elements in the environment (outside the boundary).

Definition of a system from the Systems Engineering Body of Knowledge (SEBoK) summarized what we've discussed in this section: "*A system is a collection of elements and a collection of inter-relationships amongst the elements such that they can be viewed as a bounded whole relative to the elements around them. Open Systems exists in an environment described by related systems with which they may interact and conditions to which they may respond*".

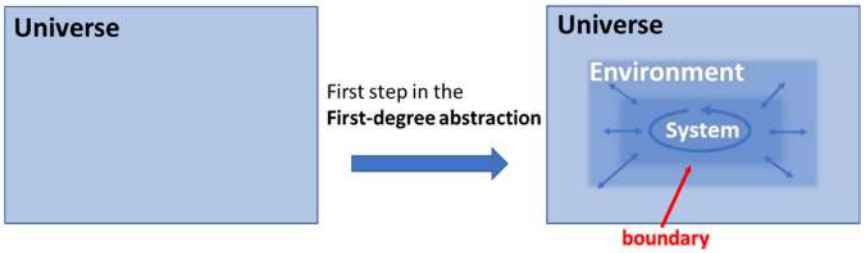


Figure 1-13: A system as a group of interacting elements, subjectively separated from the surrounding environment

'Flow' systems

After we set the boundary of our system, we need to analyze the inner workings of the system based on archetypes (mental models) that we're familiar with. **Flow systems** are a common archetype used as a class of systems in simplifying the modeling.

> *"A flow system is one in which some commodity [entity] flows, moves, or is transferred through one or more finite-capacity channels in order to go from one point to another" (Kleinrock, 1975)*

Many systems can be classified as flow systems. (for example, flow of patients through a hospital, passengers through different sections of an airport, or goods through a warehouse). This classification helps us to immediately think about that system in terms of flows which in turn simplifies and standardizes our analysis and thought process (Figure 1-14).

Imagine you're asked to model a hospital. Your first step is to decide what constitutes the boundary of the system (that is, the hospital). To do this, you need to answer some important questions. Which buildings should you focus on? Are the parking lot and entrance gate part of the hospital?

The immediate consequence of drawing a boundary around the elements that constitute the system (that is, the hospital) is that everything that is outside the boundary, but still interacts with the system, will be part of the environment (for example parking lot). After establishing the boundary, you then need to conceptualize how the system works before you can start modeling it!

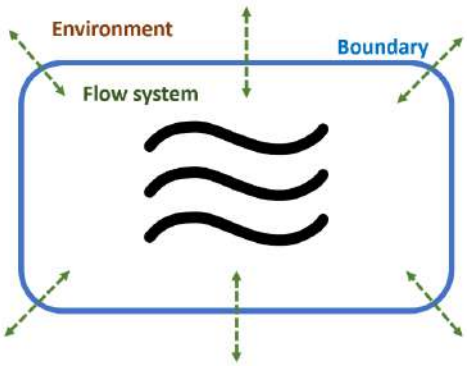


Figure 1-14: Conceptual illustration of a flow system

In our example which we conceptualized as a system, one might ask, "What exactly *is* a hospital?". This is a vague question. and it's hard to think about it without any reference framework to start with. However, if you start thinking about it in the context of a *flow* system, things start to take shape: Patients enter the hospital; they move, receive care, wait in line, may be hospitalized for a while, and will eventually leave the hospital. At the same time, there are limitations and capacity constraints on the number of patients that can get care or stay at a specific location at any point in time. By conceptualizing (abstracting) the hospital operation into a flow system, it's easier to design a standard framework.

Now that we have put a hospital in the conceptual framework of a flow system, we can start thinking about how the various parts of the flow: how it happens, capacities at different parts, the time that the patients spend at different segments, things that might help or hinder the flow, etc.

The ‘flow’ in a flow system can be categorized into two main categories: **continuous** and **discrete**. In a continuous flow, the thing that flows is not naturally compounded from discernable units (for example in a flow of water in pipelines, we cannot observe water as separated units). In contrast, a discrete flow has clearly identifiable individual units flowing in the system (for example, car bodies moving forward in an assembly line).

In AnyLogic, the fluid and System Dynamics libraries are two powerful modeling toolsets for continuous flow. The fluid library is designed to model storage and transfer of fluids, and includes blocks for routing, merging, and diverging simulated liquids. In addition, system dynamics is a more general simulation methodology that could also be used for continuous material flow.

There’s no inherent issue in treating continuous flows as discrete – for example, abstracting a continuous flow of water in pipes into 1 cubic meter containers moving on a conveyor. However, in using discrete techniques it’s important to be mindful not to lose critical system behavior because of the higher-level abstraction. Our focus throughout this textbook is on flow systems with discrete flows.

Process: representing the innerworkings of a flow system

Categorizing a system as a flow system is the first step in clearing the way in abstracting the slice of reality in focus; we're however still dealing with an amorphous and unstructured problem. The next step could be to think of the innerworkings of a flow system as a process or several processes.

The term "process" in day-to-day spoken English means a series of changes that happen to something, possibly conducing to an end (for example aging process, immigration process). In our context, it refers more to the events needed to change some inputs into some other outputs (Figure 1-15).

Figure 1-15: Generic illustration of a process

The flow system classification is a very high-level abstraction; it only tells us that the system is interacting with its environment and there are things flowing through it. However, considering the inner working of a flow system as a process will help us to conceptually visualize the structure of the flow.

In Figure 1-16 below, the internal process is shown with an array (signified by '[…]') of inputs, processes, and outputs, emphasizing that we might have more than one process in the flow system. However, this illustration does not specify whether the processes are in series or parallel, it is merely an illustration that there might be more than one process representing the flow.

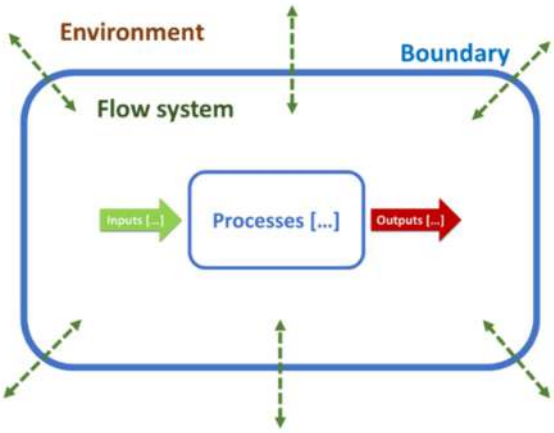


Figure 1-16: Flow system and internal processes

Abstraction level and processes

A process could be broken down into constituent processes to have a more refined picture of how the transformation of inputs to outputs happen. Take the simple example of a stream of cars passing through a carwash – this is a flow system that could be considered having one process: cars are coming in (input), get washed (process), and leave (output).

You can break this process into more processes to analyze the operation at a finer detail. The "get washed" process can be broken into the processes of payment, washing, and drying. Similarly, any process could be modeled at a low-level abstraction with several processes connected to one another in a series, in parallel, or any other meaningful arrangement or at a high abstraction level with only one or a few processes (as shown in Figure 1-17).

The crucial point here is that when we visualize the model in our heads, we can see both abstraction levels (Figure 1-17: left). However, we cannot model two simultaneous abstraction levels of the same system. As we create our model, we must choose the high abstraction perspective that models the entire operation as one process (option 2), or a low abstraction perspective that separates the operation into three consecutive processes (option 1).

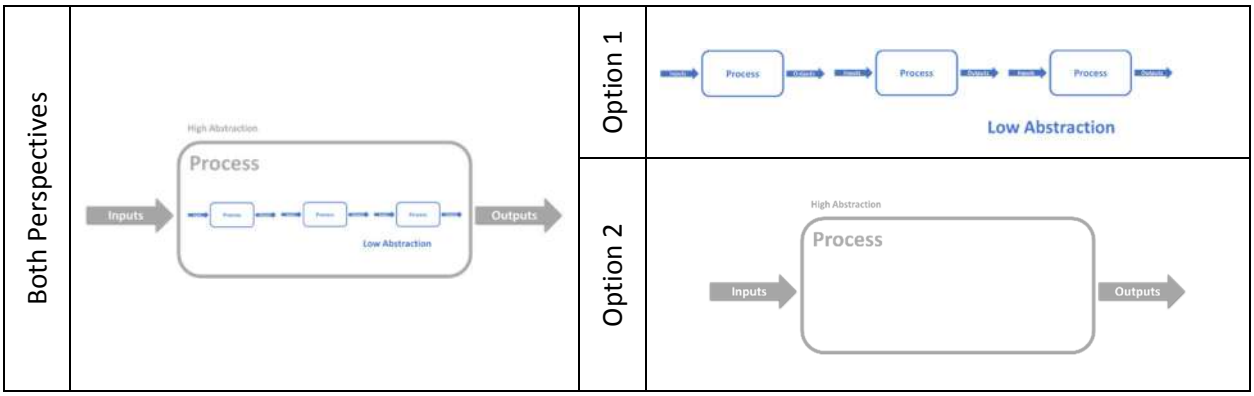
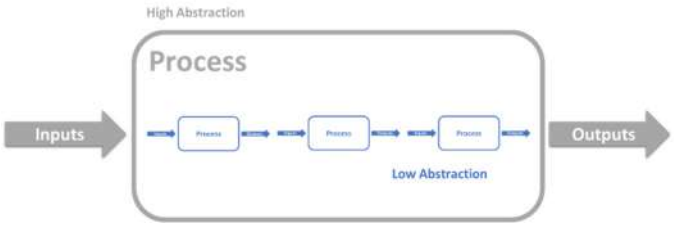
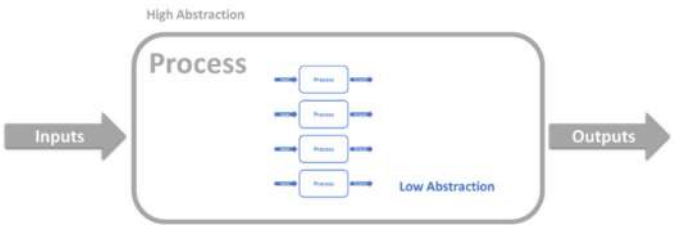


Figure 1-17: the left picture shows both abstraction levels in one illustration, however at the implementation phase you must choose a proper abstraction level and build the model accordingly (pictures on the right).

Figure 1-18 shows more example of how a process could be imagined as one process (high abstraction – gray) or several processes (low abstraction – blue).

Examples of processes configuration	Illustration
Series of processes	
Parallel processes	

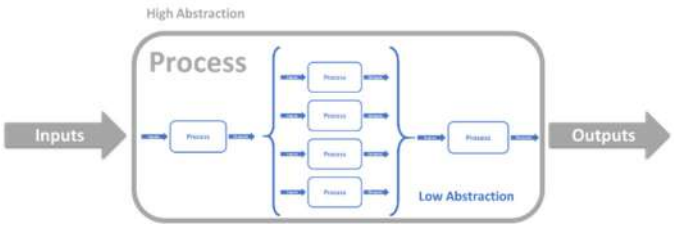
Combination of serial and parallel processes	

Figure 1-18: shows more examples of how a flow system could be imagined as one process (high abstraction – gray) or several processes (low abstraction – blue).

Figure 1-19 shows a short recap of what we've discussed so far:

- We can define the system being modeled based on the subject of our study (which focuses on a specific part of the universe) and separate it into a set of interacting elements. Some of these separated elements will interact with other elements from outside the imaginary boundary. External elements that are important to the system's function will be part of our system's environment.
- Flow systems are a specific type of system in which the internal interactions could be thought of as a flow - something moves through one or more channels. This flow system is just one subset of all types of systems we can represent using a simulation model (and are not exhaustive by any means); many systems (especially manmade) could be considered flow systems.
- Conceptualizing the problem as a flow system will help us to think about the system's inner working in a more standardized way. One popular and effective approach is to think of the flow channels as one or several processes. These processes are a transformational vehicle that converts some inputs to outputs.

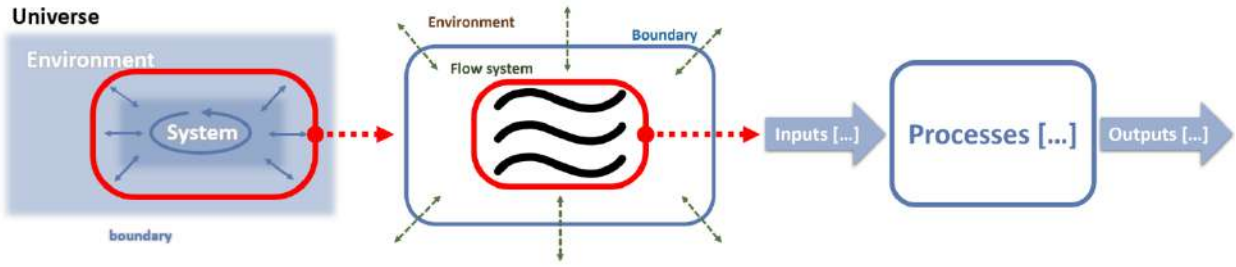


Figure 1-19: Steps of abstraction from reality to a system, a flow system, and to one or more processes

Deciding on treating a flow system as a process (or several processes) will let us tap into all the capabilities of process centric modeling. In most simulation packages, we abstract the model at process level. This means first identifying the things that flow (entities), then deciding on the sequence of operations being performed over those entities. After abstracting the process, we use the out-of-the-box process blocks of a simulation package and build the process through a **flowchart**, or a diagram that models and controls the flow. However, the simulation engine (the computer program under the hood that drives the model execution) does not understand the flowcharts directly. Simulation engine does not speak in "process" language! It speaks a lower level language of events which we'll describe in more detail in the next section. Because of this, flowcharts are a way to generating the events that the engine understands. We'll see in later chapters that in AnyLogic, you can also generate your own events without the help of a flowchart.

Discrete event modeling

Process centric modeling uses a lower level abstraction that is called **discrete event modeling** to dynamically track and quantify the state of the process. Discrete event modeling is a powerful and comprehensive paradigm that is capable of modeling almost any system that changes over time through events (providing those changes could be reasonably approximated within these events or important happenings).

First, we need to appreciate how discrete event modeling works. This will help us understand how the program turns the flowcharts we draw with a graphical user interface into a queue of events that the simulation engine (the computer program that executes the simulation model) understands.

Our perception of change (or time) in the real world is continuous; anything that happens in the real world is happening gradually. So, if our objective is to keep track of a system's state, we must keep track of the continuous changes. However, we can model change at a higher abstraction level and only focus on the system's **events**, or important moments in the system lifetime.

To summarize how a discrete event simulation works (Figure 1-20) depicts the four steps described below):

1. We observe the system's "real" dynamics (which is almost always continuous).
2. We only consider important moments in the system lifetime and treat them as instantaneous indivisible events. All changes in the system will be associated with these events.
3. The simulation engine maintains the events' queue and serializes the events; at the same time, it makes sure that the **model time**, or the unit of time the engine is running in, associated with these realized events are synchronized.
4. The simulation engine then executes the model by jumping from one event to the other, keeping track of the model time that is associated with the events.

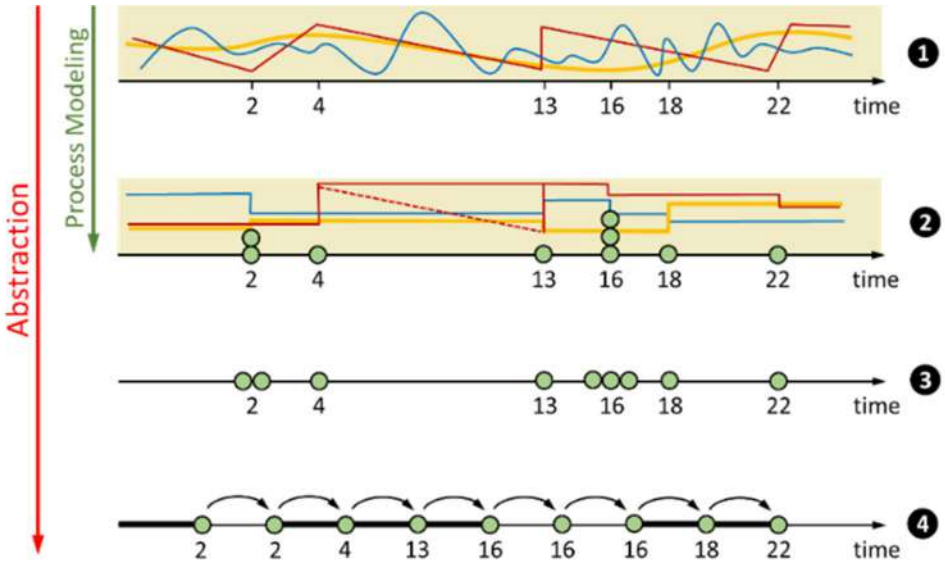


Figure 1-20: Abstraction from continuous reality into discrete events (Borshchev, 2013)

For a long time, it was solely the process centric models that were using discrete event modeling. Therefore, the terms “process modeling”, “discrete event modeling”, and “discrete event simulation” are commonly used synonymously.

However, **agent-based modeling** also utilizes discrete events; discrete event modeling therefore cannot be associated solely to process centric models anymore. For this reason, we’re following the separation of these two terms suggested by Borshchev (2013):

> “…we will be using the term “process modeling” for the modeling method where entities use resources and wait in queues, and the term “discrete event” for the more general idea of approximating the reality by instant changes at discrete time moments”.

In other words, what we call process centric models is a subset of discrete event models, which also include a subsection of agent-based models. What we call “process modeling” or “process centric modeling” in this book is analogous to “discrete event modeling” or “discrete event simulation” in other resources.

If the explanations above are still confusing, you can imagine process centric modeling as an abstraction approach that uses flowcharts to visually define how the flow happens in a system. Discrete event modeling/simulation on the other hand is a method for simulating a system’s evolution with a computer program. Process modeling is using discrete event modeling approach to simulate the process.

*It should be mentioned that “discrete event” is both an abstraction and simulation method, but for the sake of simplicity we’ll just consider it a simulation methodology.

In the past, modelers used to write computer programs to keep track of a system’s state based on a queue of events. This is a painstaking task and is analogous to writing a program in Assembly - a computer language that is only one abstraction level above raw machine code.

In a modern simulation package, modelers build their models with process blocks, which visually construct the flowchart. When you execute a process centric simulation model, the simulation engine generates the discrete events based on your flowchart. These events are maintained in the event’s queue and will be executed one by one.

> *As modelers, you do not need to worry about the underlying event generating mechanism. The simulation package’s job is to make things easier for you by providing a higher-level experience (through the convenience of modeling processes using flowchart blocks).*

To insure the modeler has full flexibility, AnyLogic provides the option to generate your own low-level events (in case you want to make a change to the system’s state that does not fit in a process). In the next section, we expand upon the process centric modeling approach and its common elements.

Process-centric modeling

Now that we clarified the possible confusions about the difference between process centric modeling and discrete event modeling, it's time to better clarify the formalization of the process modeling approach. To make things clear, there are three definitions of process relevant to this book:

1. An activity or set of activities that make something happen; or, according to ANSI/EIA-632, *"process is set of interrelated tasks that, together, transform inputs into outputs"* (**Definition A**).
    - In previous section, our definition of process was adopted from ANSI.
    - This definition is consistent with the broad definition of process used in engineering contexts.
2. *"A sequence of operations being performed over entities. These processes typically include delays, usage of resources, and waiting in queues" (Borshchev, 2013)* (**Definition B**)
    - This definition is more contextual compared to the ANSI one and is not in contradiction with it.
    - Throughout this book, the word process is used both in its broad ANSI definition and its specific simulation related definition proposed by Borshchev.
3. This definition of process is only used in the context of the phrase "*stochastic processes",* which has a very specific meaning in probability models. A stochastic (or random) process may be thought of as a process where the outcome is probabilistic rather than deterministic in nature. In other words, a random process is just a set of random variables. (**Definition C**)
    - For example: if we toss a die and look at the face-up number: tossing the die is the random process, and the number on top is the value of the random variable.
    - In a general sense, randomness and stochasticity are synonyms; my suggestion is to either consider stochastic and random as identical terms or to think of stochasticity as *exhibition of random behavior over time.*
    - The related mathematical terminology and definition can be found in **APPENDIX**.

Whenever you've encountered the word "process" in this book it is used in the context of definitions A or B. Whenever definition C is used (in some of the theoretical discussion), it is in the context of random processes.

Now that we clarified some confusing terminologies, let's review some of the general concepts that enhance our understanding of a process and define its attributes. These concepts apply to the general concept of process centric models. They also apply to its subcategories like queuing models, which could be abstracted using process centric modeling.

1. **Entities** are things that move through the process. In a flow, they go through a sequence of operations before they emerge at the end. (e.g. patients in a hospital, cars in a car wash, packages in a sorting facility, customers in a restaurant)

- Entities may not represent physical objects; they could represent intangible things like orders in an ecommerce store, requests for vaccination in a healthcare center, or loan applications in a bank
- Entities are the center focus and the main things in a flow system. The process exists because we've needed some channels to control the flow. For example: the process of washing cars exists because of cars - A car wash without cars is pointless! The same is true of many other examples: a hospital without patients, a factory without jobs, an amusement park without visitors; all are meaningless regardless of how sophisticated or interesting the process is, as the process is there to serve the entities

2. **Resources** are dedicated to assisting entities flow through the process; resources are used when entities might need other things to help them to move forward.
    - When a pallet of batteries arrives at a warehouse and need to be moved to a designated location. The battery cannot move on its own, and therefore the flow is broken! At this point we need a resource to help the entity continue its flow.
    - The need for resource is not necessarily related to flows in a physical setting. A loan application will eventually reach the desk of an underwriter. For that application (entity) to complete its flow, it needs a resource (underwriter) to spend some time with it and make the final decision.
    - In process centric models, it is the entity that requests the resource, not the opposite. For example, when a pallet arrives at the warehouse, it is the pallet that requests a forklift. Forklifts themselves are not actively looking for pallets. This is because the entities are the commanding objects in the model. Entities are the subject and the primary focus of the flow! Resources are passive things that help the entities whenever they're needed, contingent on their availability.
    - We usually build a pool of a specific resource and units of that pool can be assigned to a specific part of the process flow. We call each resource inside this pool a **resource unit**. For example: each forklift in a pool of forklifts is a resource unit. One of the main reasons we put resource units in a pool is because several processes might need to share a group of resources (the pool) (e.g. two bank branches processing the loan applications independently but use the same pool of underwriters)
3. **The calling population** is the population that the entities passing through the process are coming from. We call these populations **finite** or **infinite** depending on the definition of the arrival rate.
    - For example: customers of a bank (the entities) which arrive in a single day are a subset of the large population consisting of the entire customers who use that bank. When the calling population is large enough, arrivals in the past won't affect future arrivals and therefore we consider the calling population *infinite*. When the calling population is small compared to the entities in the system, we consider the calling population *finite*.
    - For example: consider a golf club that has 20,000 members. In a normal day, only 20 members are coming to the club; compared to the overall members, the day's customers consist of a negligible 0.1 percent. In this case, the number of previous arrivals won't

affect the expected arrival rate. In contrast, if the club had only 100 members and 60 people are in the club on a given day (60%), the remaining population is only 40% of the overall population. Therefore, it might be reasonable to assume that the expected arrival rate is disturbed.

4. A **Service time (Delay)** is the time an entity needs to wait before it can move to the next step in the flow. The value of time that we associate with the service time or delay could be constant or a random variate (an outcome of a random variable).

    - Imagine it's late at night and you are craving a pepperoni pizza. In a perfect world, you would have your pizza immediately! However, some processes must take place before you can enjoy your pizza slice, and all those processes take time.
    - Note that service time is strictly defined by the time that is associated with the task and not the time an entity spends waiting for its turn (assuming the task cannot be started immediately).
        - If placing an order over the phone takes exactly two minutes, but there was a 20-minute wait in the line to speak to an operator, the service time would still be two minutes.

5. **Queue** is a line (which can be physical or implied) of entities waiting in some order to be attended to or to proceed forward.

    - We encounter queues frequently in our lifes because of resource limitations in all types of flow systems, and especially in manmade systems. Processes that have limited resources could experience queues of waiting entities in case the entity arrivals (that is, the demand) surpasses the availability of resources.
    - Queues could be observed in all types of industries that fit in flow systems, such as: customer service (call centers, banks, shopping centers), healthcare (surgery schedules, emergency room, pharmacy), IT (servers, networks), and manufacturing (production line, conveyor systems).

6. **Queue discipline** is the definition of how the entities are ordered in a queue. Queue disciplines are present in any process centric model and they decide which entities are at the head (or front) of the queue.

    Several common queueing disciplines include:

    - **First-in-first-out** (FIFO) services entities in the order they arrive in the queue. This is the conventional queue that we're all familiar with: lines in restaurants, airports, and movie theatres.
    - **Last-in-first-out** (LIFO): entities are serviced in the reverse order that they arrive in the queue, operating like a stack. Some examples include people in elevators or CDs on a spindle.
    - **Service according to priority** (PR): entities are sorted based on a value representing their priority, with the entity who has the highest priority being at the head of the queue. When

an entity enters the queue, sorting is necessary to determine their position. The priority is determined based on one or more of the entity's properties.

- For example: incoming patients in an ER queue are serviced based on the severity of their health. Upon arrival, patients may be rated on a scale or have a value calculated which takes different factors into account. Thus, patients who have life threatening conditions will take priority over others regardless of when they joined the queue.

- **Random order** (RO): entities are either chosen randomly, or the queue they're waiting in is shuffled at specific moments. Random shuffling of entities is usually triggered with an external signal.
  - For example: an online market that accumulates all the bids for a specific product throughout a week, then randomly shuffles the bids and lets the first 10 bids go through.

7. **Entity's behavior** is the description of an entity's actions while waiting in a queue. There could be several plausible behaviors; however, there are a few common generic behaviors for entities in queueing systems (or any other process involving a queue):
   - **Balk:** deciding not to join a queue if it's too long
   - **Timeout (Renege):** abandoning its position in the queue after some amount of time (e.g. if it's taking too long to be serviced)
   - **Jockey**: jumping from a line to another if the chosen line is moving slowly, or another is moving more quickly
8. **The arrival process** is a description of the timing and number of entities that enter the process. The word "process" is used here because, in most cases, arrivals are happening randomly; consequently, these are *stochastic processes* as described in *definition C.*
   - The **interarrival time** is the time interval between two consecutive arrivals. If we assume that interarrival times are IID random variables, the arrival rate is equal to the inverse of the expected interarrival time $\left(\lambda = \frac{1}{E(A)}\right)$
     - For example: ambulances in a hospital are bringing patients that are going to receive care in the hospital (the care process). Arrivals of these ambulances could be scheduled on a fixed arrival rate, such as one ambulance bringing one patient exactly every 30 minutes. '30 minutes' is the *interarrival time* between two arrivals and the exact arrival rate is 2 per hour (1/30 = 2/60) In a more realistic scenario, arrivals are stochastic and are not exactly known. However, we can approximate the expected arrival rate over a time span by counting the number of ambulance arrivals over a set period (e.g., 5000 hours) and dividing by the time span. For example, if there are 11,653 arrivals over a 5000-hour experiment, then the expected arrival rate per hour is $\frac{11653}{5000} \cong \frac{2.33}{hour}$.

- The most used arrival process is called a **Poisson process** in which interarrivals are independent and exponentially distributed; this is further explained in detail with its mathematical background in appendix.
- Other important arrival types are:
    - **scheduled arrivals** - where the time of arrival is known, for example, from historical data
    - **batch (or bulk) arrivals –** a fixed or random number of entities enter the system at each arrival
    - **time dependent arrival rates** - the arrival rate varies according to the time of day

9. **System Capacity** is the total number of entities that can be inside the system at any point in time. For some systems, it is possible to have infinite room for entities and therefore having infinite capacity.
    - Whenever we have a limited capacity, **arrival rate** and **effective arrival rate** will be different.
        - Arrival rate will point to all the entities that attempted to enter the system per unit of time
        - Effective arrival rate is a specific subset of arrival rate which reports the number of entities that *successfully* entered the system per unit of time
    - Capacity can be measured for some or all parts of a process. The upper limit of the system capacity is the summation of all positions in the process, except for an artificial constraint that's set to limit the overall capacity.
        - For example: imagine a car manufacturer with a production line that consists of four major segments, each connected by a single conveyor.
        - If the segments in each section hold 100, 25, 35, and 50 cars (respectively), then the maximum capacity of the line would be 210 $(100 + 25 + 35 + 50)$ cars.
        - As mentioned, there may be some constraints on the entire process (or just some part of it) because of external limitations; if a safety regulation prohibits us to have an upper limit of 40 cars in each segment, then the maximum system capacity will drop to 140 $(40 + 25 + 35 + 40)$ cars.

The elements and concepts we've discussed here apply to the general process centric modeling approach. However, most of the above elements (or a variation of them) are also used in a branch of mathematics called "queuing theory" which focuses primarily on lines (also known as queues in British-English). The idea is to identify the locations in the flow that there might be bottlenecks and as the result a queue. Each of these queues will be associated with some resources represented with servers. Lack of servers will result in entities waiting in the queue. The next chapter will go over how to build simulation models of queueing systems, along with the probability models (mathematical solutions) that provide closed form solutions for these systems.

Chapter 2 : Origins of Process Centric Models (Queueing Models)

Queuing system

In a simulation model, the process centric abstraction of a flow system could be modeled with blocks that control the flow. In any state-of-the-practice simulation package, you will have access to a variety of blocks that let you control the flow in a flexible manner. The core behavior of this approach is inherited from a branch of mathematics called **queueing theory**. Although queueing theory focuses more on the mathematical solutions (that is, analytical approach and not simulation) of flow systems, it's important to review it to have a better understanding of the original methodology behind process centric modeling.

In this chapter we review the mathematical solutions and simulation models of several fundamental queueing systems. This will give us a solid insight into the origin of process modeling approach and its mathematical foundations. You should treat the mathematical solutions and their formulas as a reference and focus more on the analogous simulation models.

Again, what we'll discuss in this chapter is a small subset of scenarios covered under the umbrella of process centric modeling, however what you will learn from these seemingly simple queuing models will be extremely useful in your depth of understanding of more comprehensive process centric models that we'll cover in later chapters. We'll see later that in AnyLogic's Process Modeling Library (PML), you have access to a more comprehensive set of blocks for controlling the flow than what is inherited from queueing theory.

Queueing theory is the mathematical study of waiting lines/queues. A queuing system is one in which entities (things that flow) come in, receive a service (or several services) from resources in the system, and then either leave the system or circle back to a previous point. In this flow, there's a chance that there are more entities demanding a service than what is available. This lack of resource will force entities to wait in lines/queues before receiving the requested service.

> *"A queuing system is within the class [subset] of systems of flow" (Kleinrock, 1975)*

In our effort to formalize the abstraction phase of a flow system, we used the term "process" to better conceptualize the flow systems. We adopted the "process" terminology and concept from the manufacturing industry. However, in a queueing theory literature (a mathematical discipline), the step to identify process/processes does not exist - or at least not in the way that we'll introduce here.

For the sake of continuity, we'll fit the queueing theory's terminology into the process centric approach. Since the process centric view is much broader and more flexible, our approach in using processes will be much more rigid and standardized here compared to what we'll show in practical simulation models.

Because queuing systems have a specific structure, there's a standard way of conceptualizing them into processes. In these systems, a simple process can represent how a resource helps the entities receive the required service. For example: when a passenger arrives at the check-in kiosk at an airport, the resource (or "server" in queueing system terminology) is the airline representative who receives the bag. If there are more passengers wanting to check-in their bags than there are representatives, only the amount of passengers equal to the free representatives will be able to be serviced; the rest must wait in a line/queue. Based on this, a simple process could be modeled with a combination of a queue and server(s) – a pairing known as a **service station**, **service center, service facility**, or **service** for short (Figure 2-1).

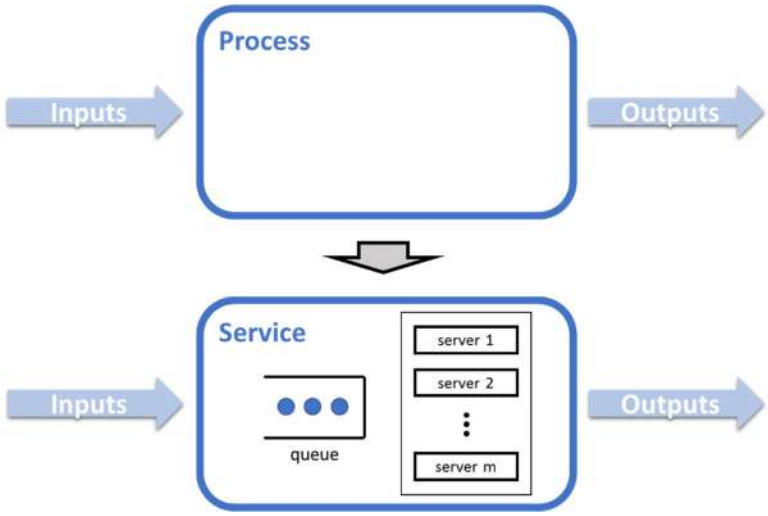


*Figure 2-1: One way a process can be modeled is as a **service** (one queue + one or more servers); note that the service is strictly limited to one queue, but it can have one or more servers*

The first step in the modeling of queueing systems is to identify the entity (for example, customers in a bank or patients in a hospital). The second step after identifying the entities is to identify all the processes in the system based on the chosen level of abstraction and model them with service stations. Since each process is modeled with a service, we can look for places in the flow that entities will look for resources before they can move forward; if they cannot immediately find a resource, then they must wait in a queue.

Think about a car wash. We can build a *high abstraction* model in which the entire flow of cars (entities) is one process. As shown in Figure 2-2, this process will be a model with a service that has one server (assuming the car wash can only wash one car at a time).

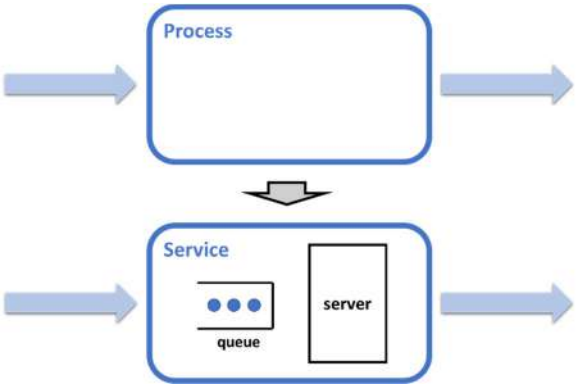


Figure 2-2: Car wash operation as one process (one service station)

If we want to model the flow at a lower abstraction level, we must identify the detailed processes that represent the various service stations. In our car wash example, we can identify three processes (payment, washing, and drying) that meet this standard (Figure 2-3).

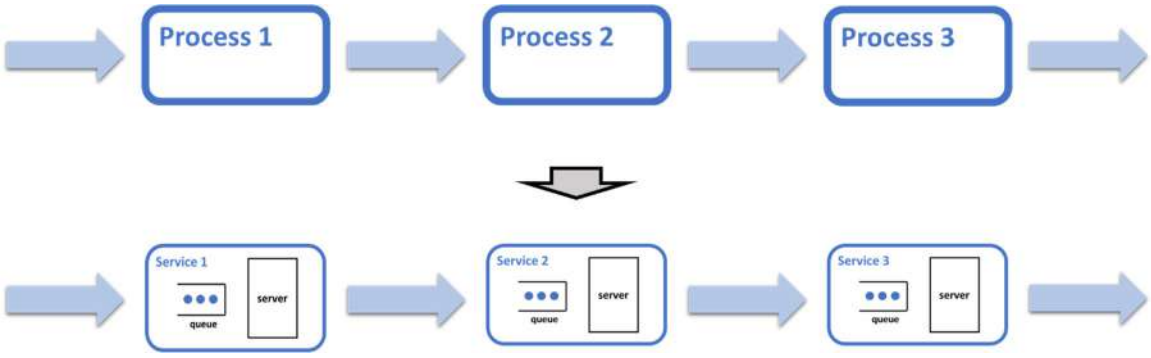


Figure 2-3: Car wash operation as three serial processes (three serial service stations)

In queuing theory terminology, whenever we have more than one queue (and thus more than one service station) connected to each other, we call that queuing system a **queuing network**. In these networks, each service station is a **node** and the connections between the nodes are referred to as the **customer (entity) routing**. The car wash example (Figure 2-3) is a queueing network because it has three service stations connected in tandem.

To summarize: the abstraction steps for a queuing system is like any other flow system. It starts with identifying the flow system and processes that conduct the flow. However, since the inner workings of a process in a queuing system are limited to services, the flow system cannot have anything other than one or more sets of queue and server/servers (Figure 2-4). Thus, the easiest way to determine the processes is to identify the location of queues; these locations are found in the places that entities need some resource(s) to help them move forward in the process. Queues that form within a system are a byproduct of lack of available resources.

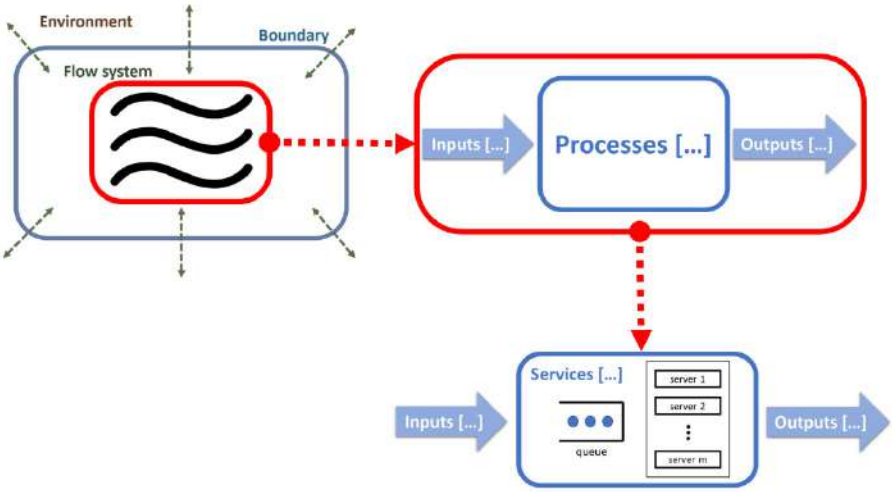


Figure 2-4: Steps in abstracting a flow system with a queueing system

Mathematical models based on queueing theory are complex; finding a solution for a basic queueing network can be a huge undertaking! Fortunately, performance metrics of these systems can be easily estimated with simulation models.

Technique 1: Modeling service stations in AnyLogic

As we discussed in the previous section, we could model an entire process at the highest level of abstraction with one queue and one or more servers. In queueing theory terminology, we call this combination a **service station**, or **service** for short. More than one service (that is, a combination of several service stations) is known as a **queueing network**.

If we abstract the entire process (the system of flow) as one service station, the input is the entities' arrivals at the service station. The output is the entites' departures from the service station (Figure 2-5).

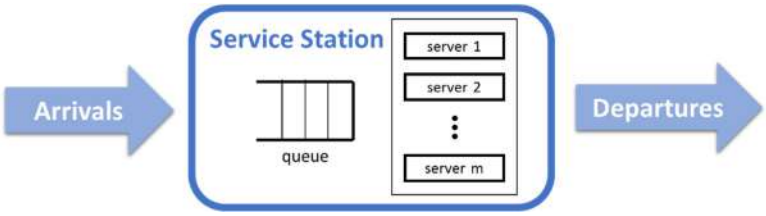


Figure 2-5: A generic service station (arrivals + queue + server(s) + departures)

AnyLogic allows us to model a single service station to be modeled in three ways in AnyLogic:

Approach 1: Having a queue and a delay as the main building blocks with no explicit server(s)/resource unit(s) as shown in Figure 2-6. In this approach, we use the following blocks from the PML:

- **Source**: represents the arrival process (for example, arrivals that are a Poisson process); the starting point in the process
- **Queue**: the space that lets the entities amass if there's no immediate server available to start serving them. Queue has a capacity that sets the maximum number of entities that can be inside the queue at any moment
- **Delay**: represents the server(s) of the service station by having the entity pause for a certain amount of time. Delay time will be analogous to the service time of each server. Capacity of the delay is analogous to the number of parallel servers in the service station. For example, if capacity of delay is set to 5, then a maximum of 5 entities can be served in parallel)
- **Sink**: represents the moment that the entities leave the process, and thus we stop following them

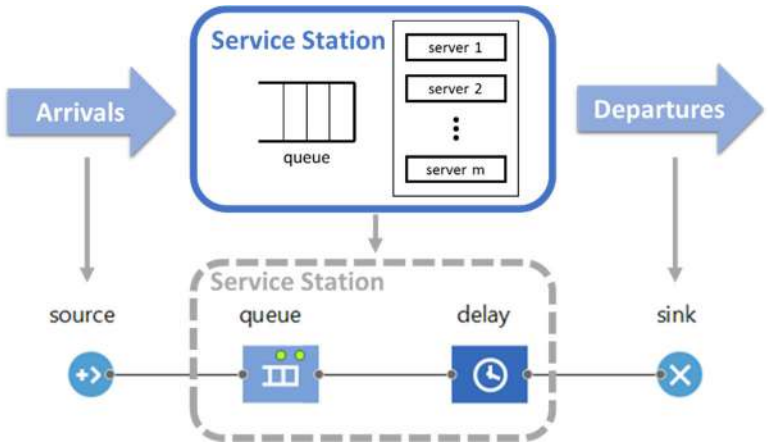


Figure 2-6: Approach 1: Delay represents the server/servers, without explicit modeling of resources

In this approach, we do not model resource units explicitly, which means that although some form of resource is helping to serve the entities, we're *only* concerned about the time associated with the service (and not time related to downtimes or calculation of resource utilization)

Approach 2: Explicitly modeling resources and using relevant AnyLogic blocks to describe the process in detail as shown in Figure 2-7. In this approach:

- **Source**: Used in the same way as in approach 1; represents the arrival process (e.g. arrivals that are a Poisson process)
- **Seize**: Internally is composed of a **Queue** block, and a mechanism that requests and grabs available resources so the service could start. Like the **Queue** block, you can set the internal queue's capacity to limit the number of entities inside.
- **Delay**: Strictly represents the time each server needs to serve the entity
    - Since the **Seize** block (in combination with **ResourcePool**) handles which and how many servers are used, the capacity of the **Delay** block *MUST* be set to infinity (maximum capacity)
    - This means that the delay does not represent the number of servers; the **ResourcePool**'s capacity (number of resource units) is the controlling constraint that represents the number of available servers
- **Release**: Disassociates the resource unit and the entity; frees the resource unit and returns it back to the available resource pool
- **Sink**: Used in the same way as in approach 1; represents the moment that the entities leave the process, and thus we stop following them afterwards.
- **ResourcePool**: represents a pool of resource units (or servers) available to the incoming entities.
    - One or more units are taken from here by entities in the **Seize** block, and later returned in **Release** blocks

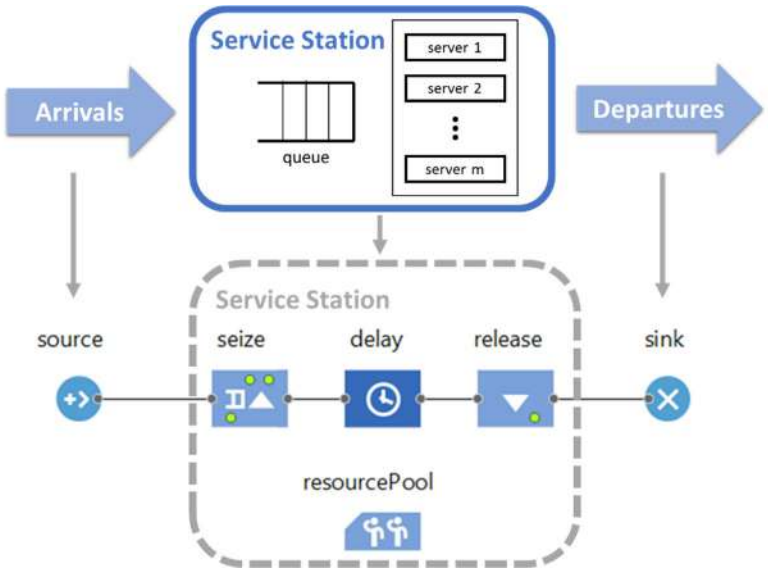


Figure 2-7: Approach 2: Seize, Delay, Release blocks in combination with a ResourcePool

Approach 3: Explicitly modeling resources through a single AnyLogic block as shown in Figure 2-8. In this approach the following blocks are used:

- **Source**: Used in the same way as in approach 1 and 2; represents the arrival process (e.g. arrivals that are a Poisson process).
- **Service**: a compact representation of an entire service station
    - A combination of a **Seize**, **Delay**, and **Release** block that is used in the same way as in approach 2.
    - Like in the previous approach, the internal **Delay** has a maximum capacity since the number of servers is set by the capacity in the **ResourcePool**.
- **Sink**: Used in the same way as in approaches 1 and 2; represents the moment that the entities leave the process, and thus we stop following them afterwards.
- **ResourcePool**: Used in the same way as in approach 2; represents a pool of resource units (or servers) available to the incoming entities.

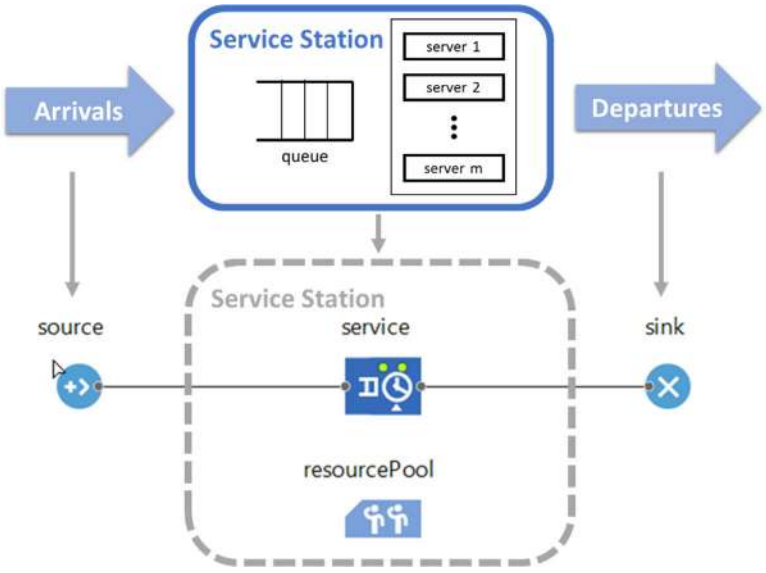


Figure 2-8: Approach 3: Service block in combination with a ResourcePool

In general, the concise approach to model service stations in AnyLogic is the simplest approach that does not sacrifice model accuracy - typically, this is done using approach #3 (that is, using a **Service** block with a **ResourcePool**). Depending on the service we're modeling, we may need to take other approaches. For example, queueing systems models that we'll cover in the following sections of this chapter need detailed recording of their performance metrics. That's why we'll use approach 2.

Furthermore, **queueing networks**, a combination of several service stations, can easily be modeled using several **Service** blocks and their associated **ResourcePools**.

Further discussion is continued in **TECHNIQUE 2**, which explains in more details how the **Seize** and **Release** blocks work in any process that uses resources, including what was shown in approaches 2 and 3.

Technique 2: Use of resources in a process

Almost all processes are reliant on resources that help them to happen. In previous sections, we've focused on queueing systems, a subset of process centric models, which only consisted of service stations. This technique goes over how an entity can seize, use, and eventually release a resource as it passes through the flowchart. In later chapters, you will see that this is one of the core techniques extensively used in *all* types of process centric models.

Whenever an entity needs one or more resource units, it must seize the resource, complete the action, and then release the resource. In further detail, the logical steps in using resources are as follows:

1. The entity that needs a resource must wait in a physical or logical place for the next available resource. In AnyLogic's Process Modeling Library (PML), the **Seize** block has a built-in queue which keeps the entity(s) waiting in line until the needed resource becomes available. Once that happens, the next entity seizes it and can move forward with the next step in the process. The **Seize** block can only work in conjunction with a **ResourcePool** (Figure 2-9).

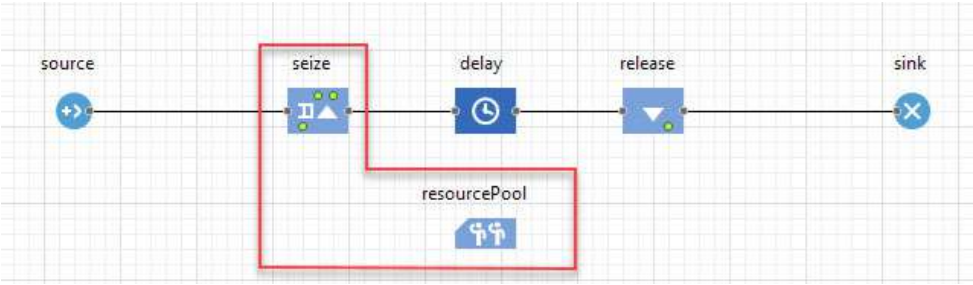


Figure 2-9: Seize grabs resource units from the associated ResourcePool

2. Once the entity successfully seized the resource unit(s) at the **Seize** block, it keeps utilizing the seized resource unit(s) as it passes through the subsequent blocks.
   - For example: in Figure 2-10, there's only one **Delay** block between the **Seize** and **Release** blocks. If the entity has not reached a **Release** block, it keeps using the seized unit(s). As a result, we can have more than one block in between **Seize** and **Release** blocks, such as when there are multiple delays needing to occur.

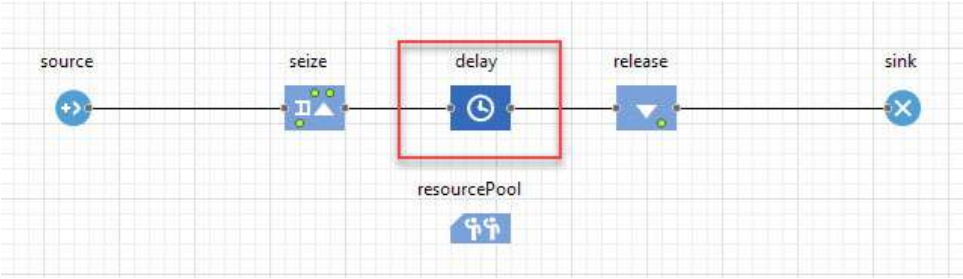


Figure 2-10: The process block(s) in between Seize and Release in which entities will keep using the seized resource unit(s)

3. After having gone through the points that the resource was needed for, we use a **Release** block to return the resource pool (Figure 2-11).

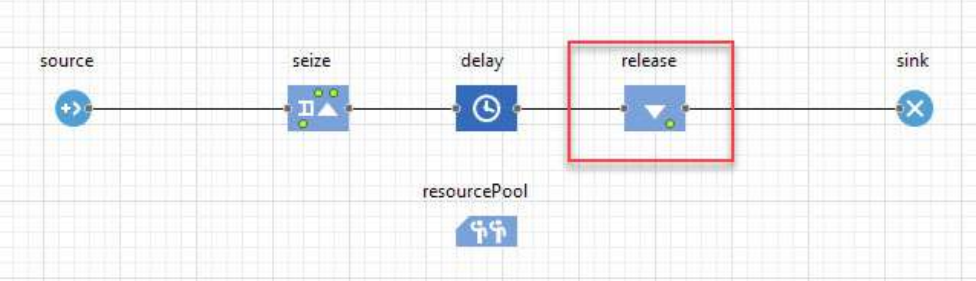


Figure 2-11: The Release block puts the resource unit(s) back into the resource pool

Since a combination of **Seize** + **Delay** (with max capacity) + **Release** is considered a service station, it can be compacted into a **Service** block, which is a combination of these three blocks (Figure 2-12). The combination of **Service** and **ResourcePool** blocks can be used to model service stations, as discussed in approach #3 of **TECHNIQUE 1**.

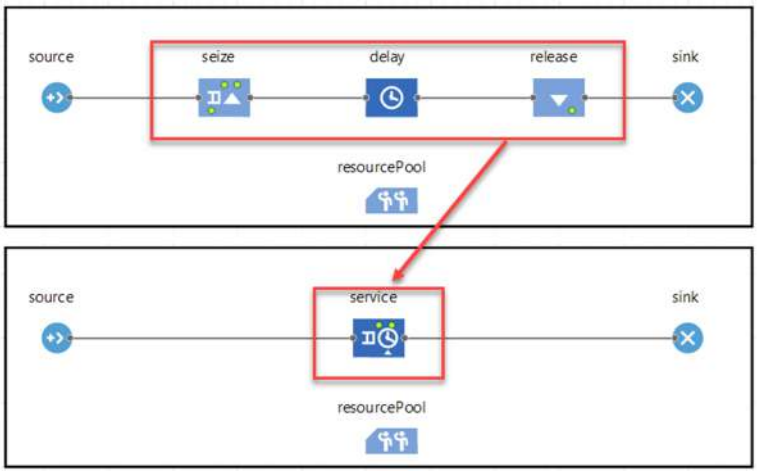


Figure 2-12: Substituting Seize + Delay (with max capacity) + Release blocks with a Service block

Example Model 1: A queue in front of a ticket booth

Prerequisites:

Model Building Blocks (Level 1): **Source**, **Queue**, **Delay**, **Sink**, **Seize**, **Release**, and **ResourcePool** blocks.

Math: Random variable, random variate, [Poisson process]

Problem statement

You work for a concert promoter, and your manager wants to hold an hour-long flash sale to sell the remaining seats for an upcoming concert. Now, you need to figure out the number of sales personnel you'll need. There's no space limitation in front of the ticket booth, so the queue length isn't a concern. Previous sales tell you an average of 2 customers will join the queue each second. However, the low customer satisfaction that long queues often cause leads your manager to decide there should be no more than 100 people in the queue at one time.

Building the flowchart process of one line and one seller (without an explicit resource)

We're going to build a model of a flow system that uses a **Queue** and a **Delay**. The queue is the space in front of the ticket booth, while the delay is the time each customer takes during their transaction with the ticket desk to receive a ticket.

As you can see in Figure 2-13, we're not explicitly modeling the salesperson (the resource); we're modeling the time it takes for a customer to pick up their ticket and depart. We'll do this by using **APPROACH 1** as shown in **TECHNIQUE 1** we described earlier. AnyLogic draws the time each customer waits from a triangular distribution.

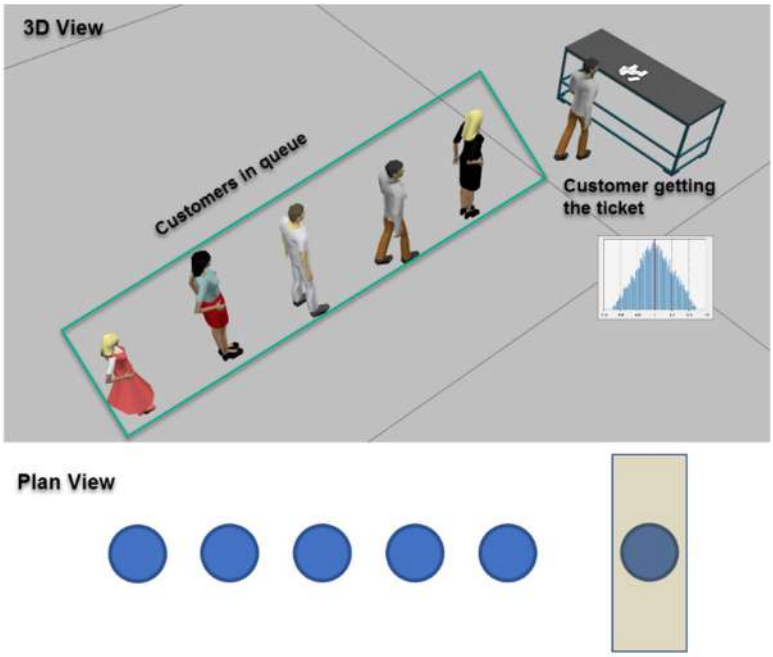


Figure 2-13: Customers waiting in a queue to pick up their tickets

Based on these assumptions, we can start building our first AnyLogic (AL) model:

1. On the **File** menu, click **New**.
2. Drag a **Source** block from the Process Modeling Library (PML) onto the Main graphical editor, and then release the mouse button.
3. On the **Properties** view, set the **Arrival rate** to 2 per second (Figure 2-14).

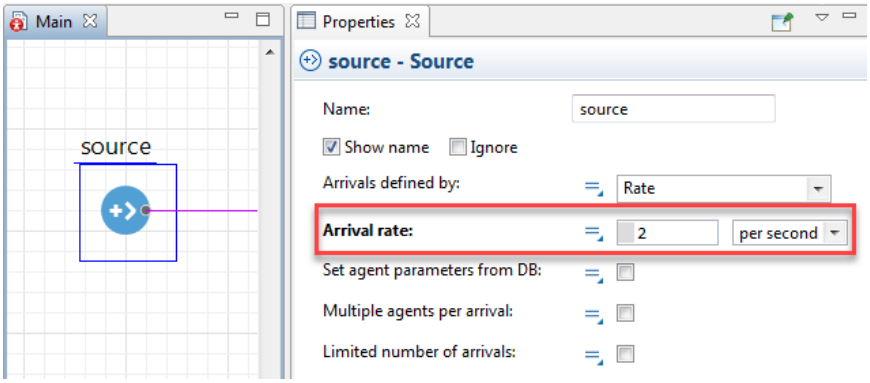


Figure 2-14: Add the Source block and set the arrival rate

4. Drag a **Queue** block from the PML on to the graphical editor and then release the mouse button so it is near the **Source** block (Figure 2-15). AnyLogic will automatically connect them.
5. On the Properties tab, select the **Maximum capacity** checkbox to set its capacity to infinite.

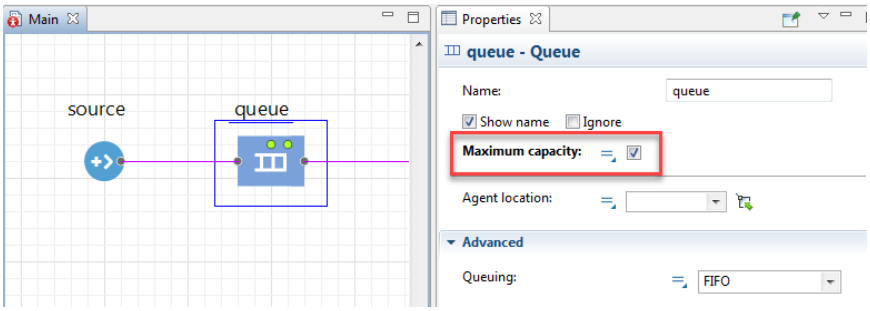


Figure 2-15: Add the Queue block and set its capacity to Maximum

6. Drag a **Delay** block from the PML onto the graphical editor, and then release the mouse button to connect it to the queue (Figure 2-16).

   We'll keep the default delay time distribution: a triangular distribution with a minimum of 0.5, maximum of 1.5 and mode of 1 second. The delay's default capacity is 1, which means we can only serve one customer at a time. This is the same as having one ticket seller in the booth.

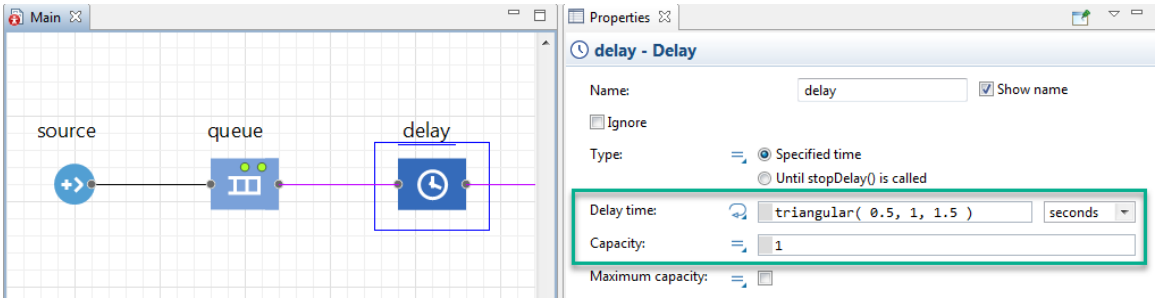


Figure 2-16: Add a Delay block and set its delay time

7. Drag a **Sink** block from the PML onto the graphical editor, and then release the mouse button to connect it to the delay (Figure 2-17).

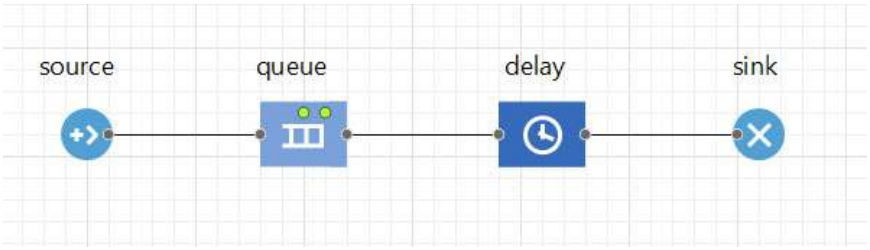


Figure 2-17: Add a Sink block to the end of a flowchart

8. In the **Projects** panel (on the left), select the **Simulation** experiment in your model.
9. In the **Properties** panel (default on the right) that displays, under **Model time**, set the **Stop time** to 3600 (one hour, in seconds) (Figure 2-18).

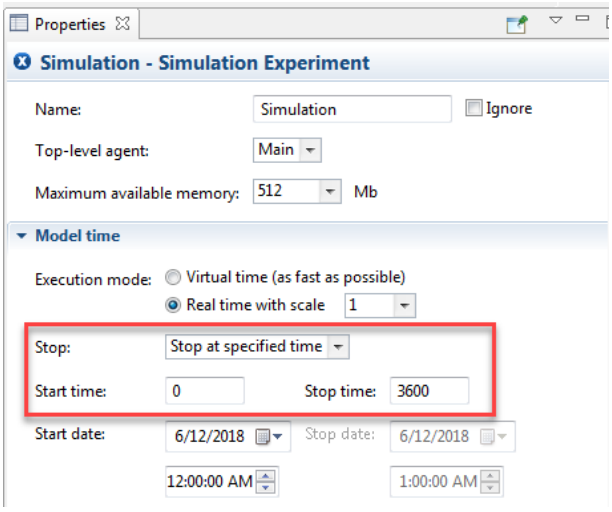


Figure 2-18: Setting the model stop time

10. On the **Properties** panel, in the **Randomness** area, make sure the random number generation (RNG) is set to **Fixed seed** (Figure 2-19).

    Don't make any changes unless you changed the default setting from **Fixed seed** to another value. In this example, we're using a fixed seed to ensure each run produces the same output.

Figure 2-19: Ensuring "Fixed seed" is set for reproducible runs (default)

11. Run the simulation experiment, setting the time speed to virtual mode; wait until the one-hour process (in simulation time) is complete. If necessary, you can use the Developer panel to check the model time.

12. Click the source, queue, and delay blocks to open their inspection windows (Figure 2-20).

    For each block, we can see the inspection window shows "*in*" (the number of people who entered the block), "*out*" (the number of people who exited the block) and other attributes such as the block's capacity and current contents.

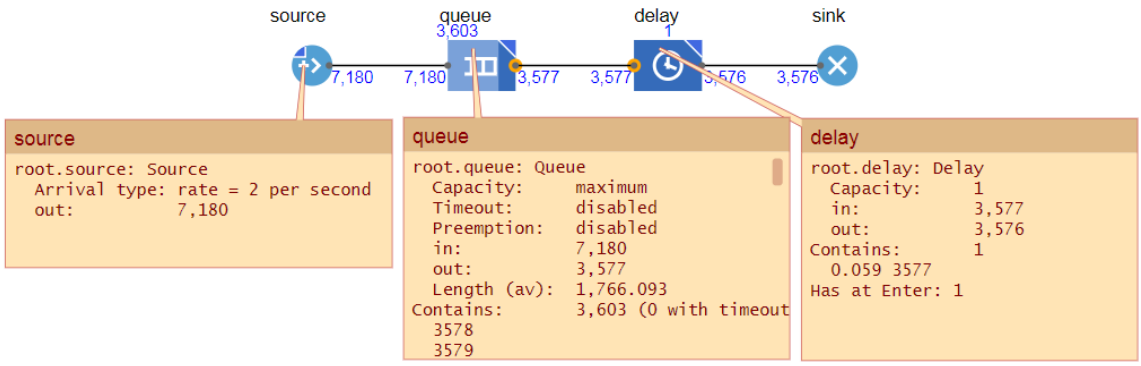


Figure 2-20: Opening the inspection window of queue and delay block in the run time

Our experiment's results confirm we'll need more than one seller if we want to keep the queue length under 100 people. The queue block shows us there were 3603 people in line after one hour of sales and the average queue length was 1766. The queue block also shows us 7180 people came to purchase a ticket, but only 3576 people received their ticket and departed.

In step 10, we made sure the random number generation (RNG) had a fixed seed. This setting ensured the random numbers AnyLogic generated for the arrival and the delay times would be the same for each run. This is helpful when we need reproducible data, but let's now change the **Random number generation** value to ensure each run produces a different result.

13. On the **Properties** panel, in the **Randomness** area, set the random number generation (RNG) to **Random seed** (Figure 2-21).

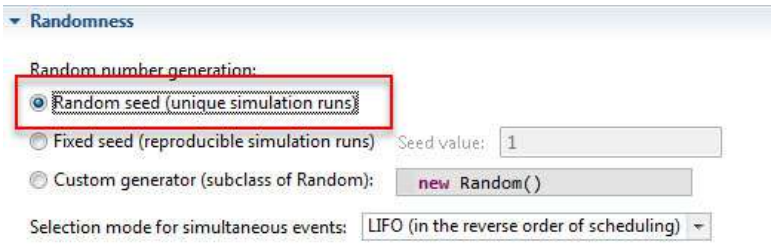
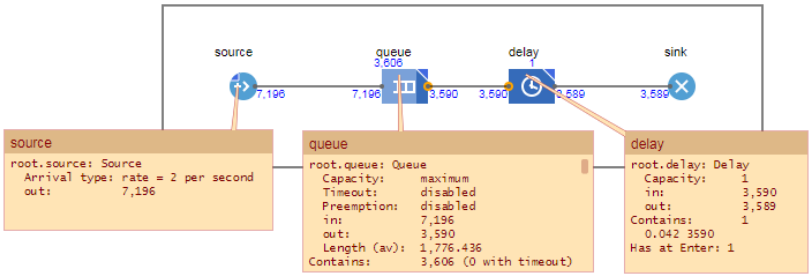
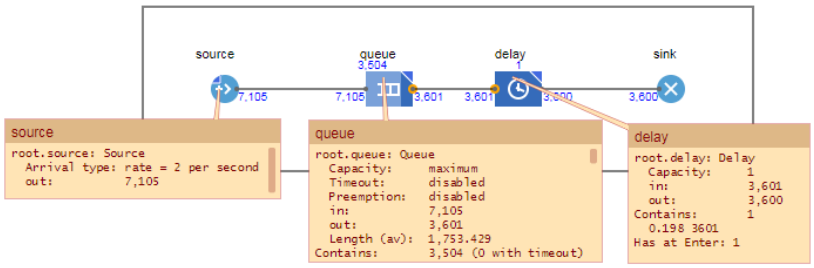
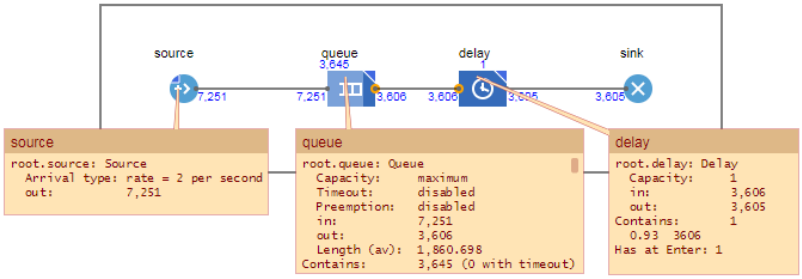


Figure 2-21: Setting the RNG to have a random seed

14. Run the simulation experiment again, set the time speed to virtual mode and then wait until the one-hour (in simulation time) process is complete.

As you can see in Table 2-1, changing the random number generation to **Random** seed produces a different result each time you run the simulation:

Table 2-1: Model results after three runs (with a random seed)

Run 1	
Run 2	
Run 3	

#	# arrivals	Queue length at time = 1 hour	Average queue length
Run 1	7196	3606	1776.436
Run 2	7105	3504	1753.429
Run 3	7251	3645	1860.698

As we look at our results, we need to focus on the "average queue length" metric. These values are a far cry from our manager's request for a queue length of under 100. However, we know queue length will vary: it might hover around 10 for the first 10 minutes and then increase to around 15 for the next 10 minutes. This variation explains why we want to look at the average queue length over the full hour.

Bear in mind it isn't easy to calculate the average queue length because the data samples (that is, the queue length) are time-persistent. This means they persist in continuous time and only change at discrete moments. In other words, the mean of the queue length is a time-weighted value. Later (in **TECHNIQUE 3**) we'll discuss how we can calculate the average queue length. For now, we'll use the value we received from our simulation.

By using the average queue length, we can make some back-of-the-envelope calculations. Let's start by calculating the expected number of arrivals. Our arrival rate for the hour averaged 2 persons per second:

$$Expected\ number\ of\ arrivals = expected\ arrival\ rate\ \left(\frac{unit}{time}\right) \times operation\ time\ (time)$$

$$= 2\ \left(\frac{customer}{second}\right) \times 3600\ (seconds)$$

$$= \mathbf{7200}\ \boldsymbol{customers}$$

When we compare this amount to the results of our three runs, we can see the simulation outputs (7196, 7105 and 7251) are all close to the expected value. We expected this variance: the randomness that is part of the arrival process causes each run's result to deviate from the expected value.

Now, let's see how many tickets we sold. Bear in mind AnyLogic drew the delay time from a triangular distribution with a minimum of 0.5, maximum of 1.5 and a mode of 1 second. And since our triangular distribution is symmetric, its mean and mode are the same (mean = mode = 1 second).

We'll start by making some important assumptions. Since the average arrival rate is double the average process time (2 per second vs. 1 per second), we can assume the ticket seller is always busy. Knowing they won't have any idle time, we can assume the seller works at his or her maximum capacity and never stops processing tickets.

On average, it takes one second to process a ticket. Assuming the seller continually processes tickets, this means his or her average throughput is $\left(\frac{1}{average\ processing\ time}\right) = \left(\frac{1}{1}\right) = \mathbf{1}$.

Let's calculate the number of tickets we expect to process in one hour:

$$Number\ of\ tickets\ sold\ =\ Expected\ throughput\ of\ a\ seller\ \left(\frac{ticket}{time}\right) \times\ operation\ time\ (time)$$

$$= 1\ \left(\frac{ticket}{second}\right) \times\ 3600\ (seconds) =\ \mathbf{3600}\ (\boldsymbol{tickets})$$

Our rough estimate is close to each of the simulation outputs throughout the three runs (3589, 3600, and 3605).

The difference between the number of customers that arrived and the number of customers who received their tickets -- plus the customer who may be waiting to get their ticket -- should be the number of customers waiting in the queue. In other words, the total arrivals is equal to the customers in the queue, those who are waiting for to get their ticket and those who left after receiving theirs. We can see this in the following example that uses the numbers from Run 1:

$$\begin{aligned} &Number\ of\ arrivals\ (7196) \\ &\quad = Customers\ in\ queue\ (3606) + Customer\ waiting\ to\ get\ the\ ticket\ (1) \\ &\quad + customers\ that\ got\ their\ tickets\ and\ left\ (3589) \end{aligned}$$

Now that we've manually confirmed the outputs, let's see if our initial settings met the requirement. We know our manager wanted to keep the queue length below 100. But with only one seller, we couldn't meet this standard. The average queue length at the end of the hour for each of the three runs was 1776.436, 1753.429 and 1860.698.

Since the average arrival rate of two people per second is twice the average service rate of one person per second, it's obvious our settings can't meet the demand. We'll need more than one seller to meet the requirement of a queue with a 100-customer maximum.

Increasing the number of sellers (without an explicit resource)

We now must figure out how to reduce the average queue length from 3600 to 100. If you look at the system, you'll see there are only two solutions: reduce the service time (increasing the service rate) or increase the sales staff (resources).

Unfortunately, our expected service time of one second doesn't leave us much room for improvement. That means we need to focus on the number of resources. Since the **Delay** block's capacity of 1 implicitly represents one concurrent server (one resource unit), we can try to improve our performance by increasing the block's capacity to two units (Figure 2-22).

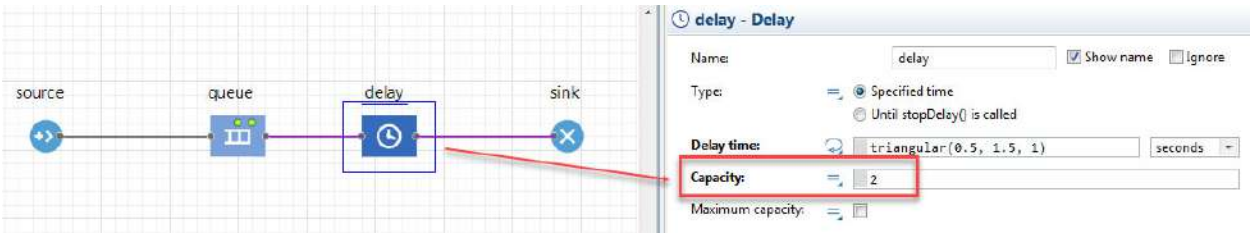
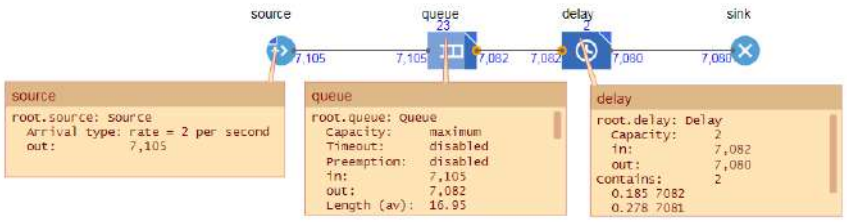
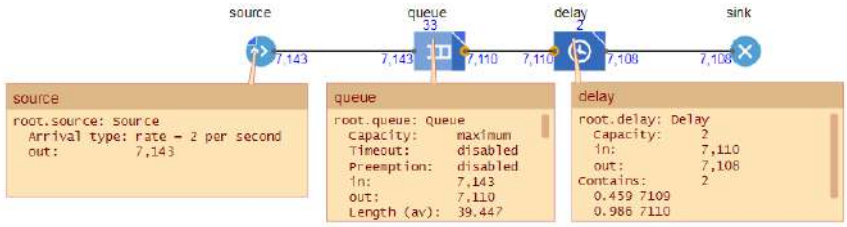
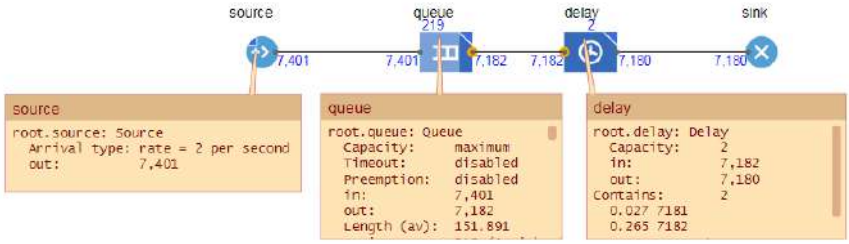


Figure 2-22: Increasing the capacity of the delay from 1 to 2 is correspondent to adding a seller

After we make this change, we have two servers (implying two sellers) and we're ready to run the model with this new value. Table 2-2 shows the results of our three runs, each with a unique, randomly generated seed.

Table 2-2: Three runs, each with a unique, randomly generated seed

Run 1	
Run 2	
Run 3	

Looking at the "*Length (av)*" values from each run, we can see the average queue length has decreased to 16.95, 39.447, and 151.89 people in run 1, run 2, and run 3. These results matched our expectations. Based on our previous calculations, we expect the throughput of two sellers to be twice as one seller, as shown below:

$$\begin{aligned} &Expected\ throughput\ of\ all\ sellers\ per\ second \\ &\quad = Number\ of\ sellers \times Throughput\ of\ one\ seller\ per\ second \\ &\quad = 2\ (sellers) \times 1\left(\frac{customer}{second}\right) = 2\left(\frac{customer}{second}\right) \end{aligned}$$

In this second set of runs, the expected arrival rate is equal to the expected service rate (2 customers/second). But if the arrival rate that regulates the inflow of customers is equal to the service rate (which regulates the outflow of customers), why is our average queue length greater than zero?

We can find our answer in the *expected* qualifier before arrival rate and service rates. We know the expected arrival rate is 2 customer/second. But the actual number may not match the expected value. For example, after 100 seconds, the expected number of arrivals is 200. However, the actual (realized) value might be 198, 199, 200, 205, or 210. The same is true for the service time. While each seller (or server) takes *on average* 1 second to process a ticket; the actual value for each transaction might differ.

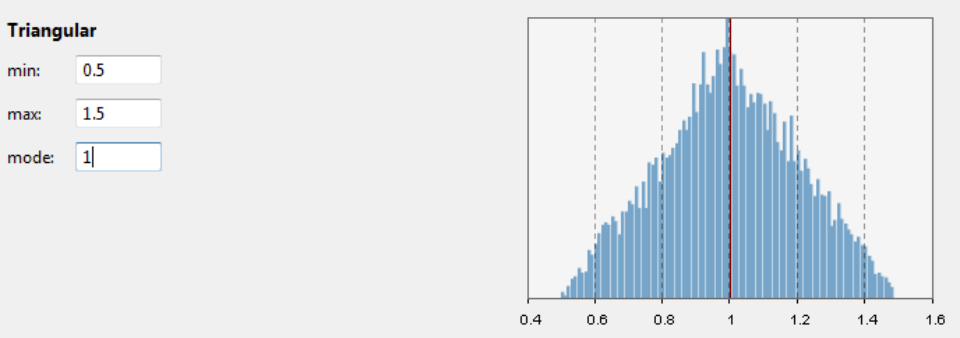


Figure 2-23: Histogram of samples generated from a triangular distribution

As you can see in Figure 2-23, the outputs from the **Delay** block's `triangular(0.5, 1.5, 1)` distribution show there are many service times above and below that one second average. Variability in the arrivals and service times almost always has a negative effect on the system's throughput.

Here's another interesting question: Why is there a significant difference between the average queue length and the number of entities in the queue when the simulation ends? (compared in the Table 2-3).

Table 2-3: Outputs from three one-hour long runs

Run	Average queue length	Queue length at the end (1 hour)
1	16.95	23
2	39.44	33
3	151.89	219

The answer depends on the time the delay overflows into the queue. If the buildup begins toward the beginning of the simulation, the effect will be greater than if it began toward the end. Since the arrival rate and service time are randomly distributed, if more entities arrive at the beginning of the hour than we expect and service times are higher than usual, the system will need extra time to process them closer to the beginning. This means the buildup will start early and accumulation increases throughout the simulation.

If you compare our results to our original setup that had one seller, you'll see the runs with one seller are more consistent. This is because the inflow with one teller is so much larger than the processing power there's a high probability the blockage will occur near the simulation's start.

With one server, the queue grew and the ticket sellers (servers) were always busy. Now that we have two sellers and a reasonable average queue size, let's look the utilization of our resources (sellers). We define the **utilization** of a server (a resource unit) as the percentage of time the unit was busy. In a simulation that runs for 100 minutes, the utilization of a resource unit which was busy for 35 minutes is 0.35 or 35%.

When we have more than one resource unit (a resource pool), utilization is the mean of the resource units' utilizations inside the pool. To track our sellers' utilization, we need to change our modeling approach. One option is to use the second approach we introduced in **TECHNIQUE 1** by using **Seize**, **Release** and **ResourcePool** blocks.

Modifying the process flowchart (two explicit resource units)

Before we modify the model, you may want to keep your original version intact. To do this, click the **File** menu and then click **Save As** to save a copy with a different name.

To modify the model (as shown in Figure 2-24), do the following:

1. Delete the **Queue** block.
2. Drag a **Seize** block from the PML on to the graphical editor, and then release the mouse button to place it before the **Delay** block.
3. Add a **Seize** block before and a **Release** block after the **Delay**.
4. Add a **ResourcePool** and set its capacity to 2 to represent two sellers.
5. Click the **Seize** block and do the following:
    a. Change **Seize policy** to **units of the same pool**.
    b. Select the **resourcePool** block in the resource pool drop-down menu.
    c. Check the **Maximum queue capacity** checkbox to set the embedded queue's capacity to infinity.
6. Click the **Delay** block and check the **Maximum queue capacity** checkbox to make sure the delay doesn't limit the entities the resources can serve in parallel.

    This setting makes the **ResourcePool** capacity the controlling constraint and sets the number of parallel servers. Please refer to **TECHNIQUE 1** to read more about this setting's importance.

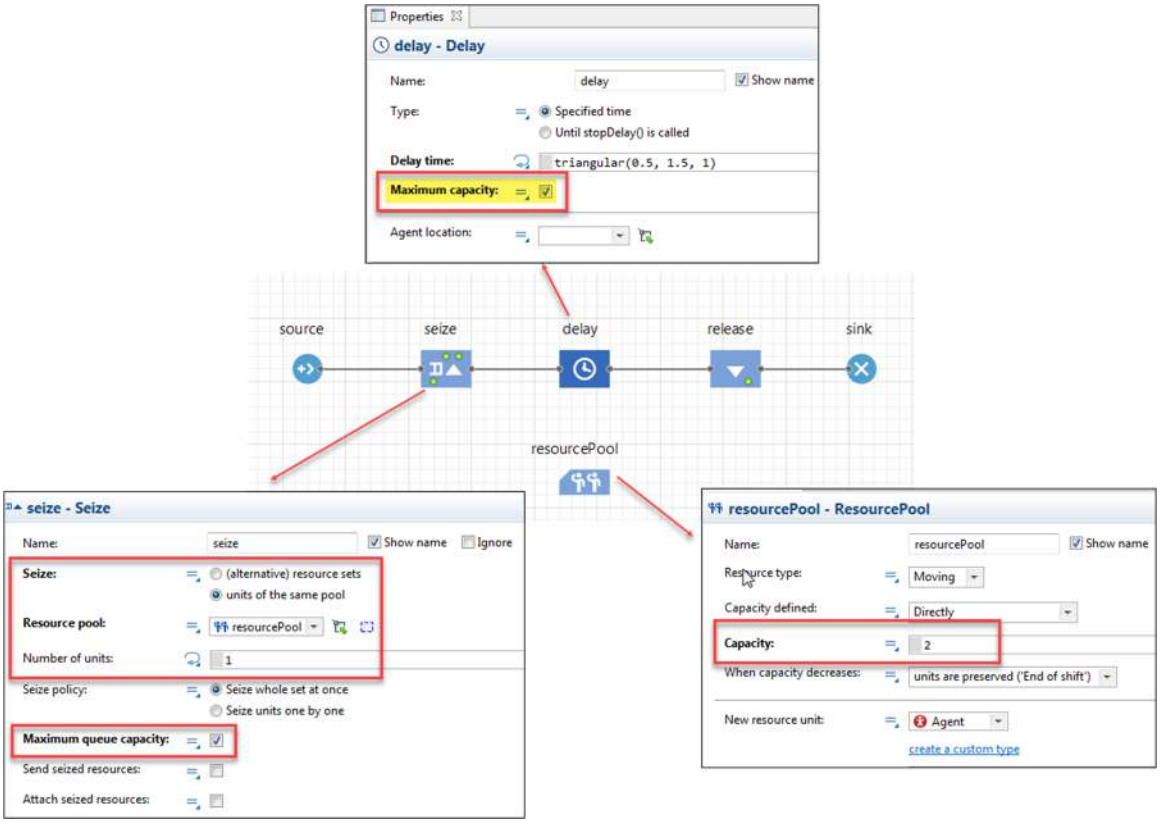


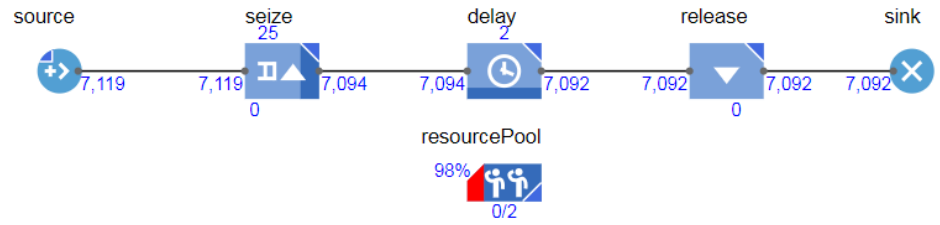
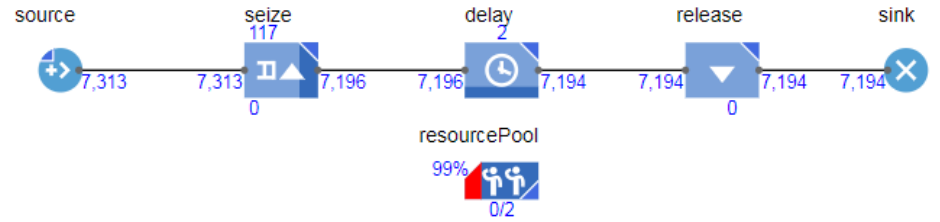
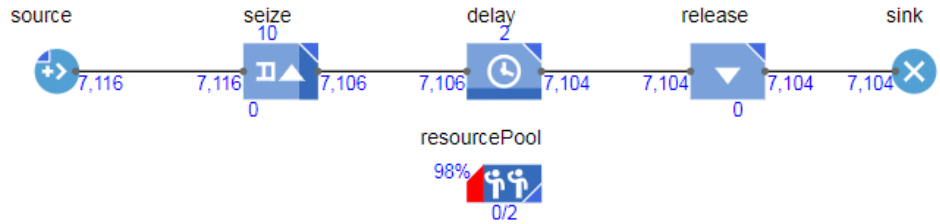
Figure 2-24: Modified model with explicit resources

We're ready to run the modified model and compare the outputs. The modified model is the same system as the original model that had 2 capacities for its delay block. The only difference is how we model the resources.

In the original model, we modeled the servers implicitly. In the modified version, we modeled them explicitly as resource units (inside a resource pool). The original model's **Queue** block has an internal statistics object that automatically records the queue's length. However, the embedded queue inside the **Seize** block doesn't have an internal statistics object. This means we can't monitor the average queue length without manually adding it to the model.

In **TECHNIQUE 3**, we'll calculate the average number of entities in a block or multiple blocks. In the modified model, we have resource units (in a resource pool) and we can see their utilization. Table 2-4 shows mean utilization of two sellers were 98%, 99%, and 98% by the end of the one-hour simulation time for runs 1, 2, and 3, respectively.

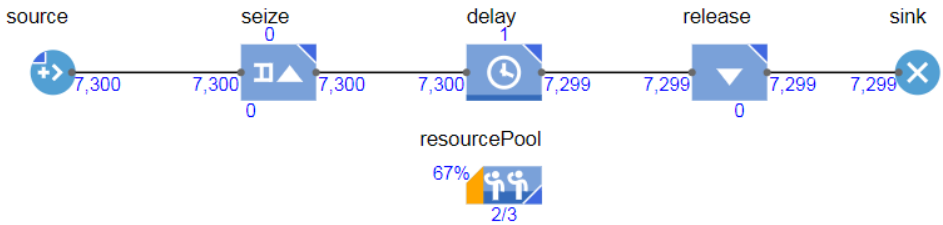
Table 2-4: Outcome of modified model with 2 explicit resources

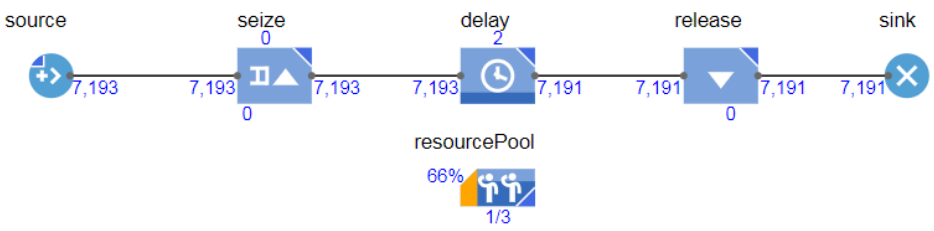
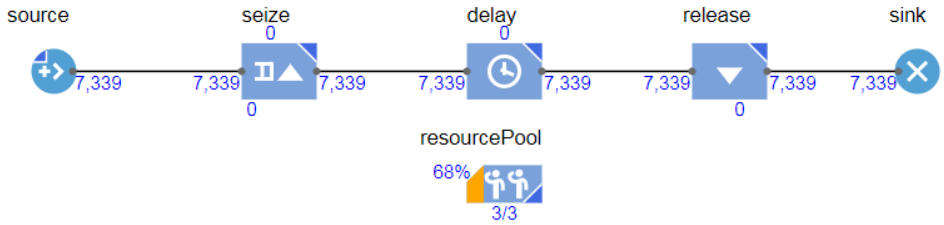
Run 1	
Run 2	
Run 3	

A persistent high utilization of a resource is a sign the resource is near its capacity. There's a good chance the lack of resources will slow or block the system. These blocked or slow points -- we call them **bottlenecks --** limit the flow of entities in a process. They usually occur when resources can't keep up with arrivals, possibly due to a slow serving speed or insufficient quantities.

Since the sellers are working at their maximum capacity (only having 1-2% idle time), one solution is to add another resource (seller). Table 2-5 shows the results of the three runs with an additional seller. Utilization dropped below 70% and people were served with almost no line!

Table 2-5: Outcome of modified model with 3 explicit resources

Run 1	

Run 2	
Run 3	

While we solved the problem, our understanding of the process' behavior is still superficial. We haven't answered fundamental questions about the system such as:

- Can we predict the system's behavior before we run the simulation? Is there an analytical solution for our problem?
- How long should the simulation run? Will the model show similar behavior if it runs longer?
- Besides the average queue length, which metrics should we use to gauge system performance?
- How do some metrics influence others? Can we use one metric to estimate another? For example, can we use average queue to estimate average wait time?
- How should we handle the randomness in the simulation outputs? We saw each run's output was different. How can we be confident about our claims when the outputs are random? Were three runs enough to claim the other runs will be similar?
- Are there archetype queueing systems we can learn from?

Our next step is to look at archetypes of queuing systems which led to contemporary process-centric models. We'll start by classifying simulation models by their model time and review common performance metrics in queueing systems. We'll then review fundamental queueing system archetypes, their analytical solutions and their simulation model representations.

Terminating (finite-horizon) vs nonterminating simulations

In our definition of a system, we subjectively set a boundary to separate interrelated things of interest from the rest of the universe. We should also mention a system isn't set in one static moment – things are also connected throughout time (as shown in Figure 2-25).

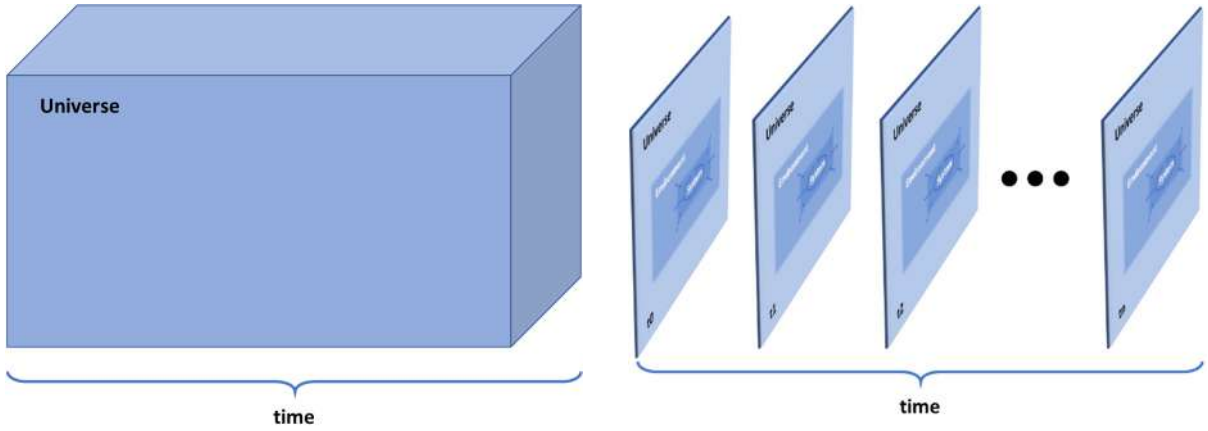


Figure 2-25: Universe existing both as one objects in space and time and as interconnected snapshots in space throughout time

This concept reminds us systems can span from the beginning of time to the end of time. Our decision to choose a time span that works well for our study means we define both the system's boundary and its time span (Figure 2-26).

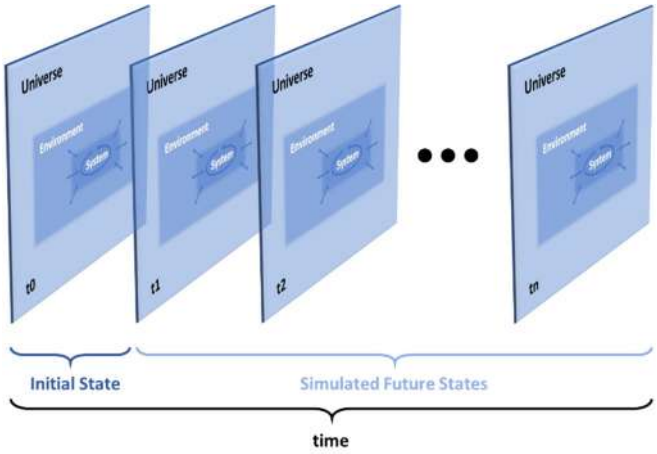


Figure 2-26: Progression of a system over time from an initial state to an end state

Let's start to clarify these concepts by imaging we own a shop that will hold a one-day sale to get rid of its inventory. A reasonable abstraction would include elements such as our customers, sales representatives and inventory as well as space for a queue within the shop. Setting the model's time span to 24 hours would allow time to simulate the sale's outcome.

We could also include more detailed factors to better simulate the sale's outcome. They might include the attitude of our sellers on a given day, the sales of nearby shops, and how our store's prices compare

to our competitors. We could even include the weather and the state of the economy. All these factors play a role in determining the success of our shop's sale.

A similar concept holds true for our abstraction's time span. In our model's simplest implementation, we would set the simulation's time span to 24 hours. However, interconnectivity among things doesn't occur during one moment; It occurs throughout time. Think of what determines our flash sale's results. It's more than passing customers who happen to visit; customers who enjoyed their last visit to our store might bring a friend to the sale. You can also imagine the effects of external forces such as the weather or economic conditions that began some time ago and will continue for the foreseeable future.

Even in a simple example, it can be difficult to choose our system's boundary and time span. Does this interconnectivity and continuity make it impossible for us to build representative models? In theory, the answer is yes. Practically speaking, the answer is no. The interconnectedness and continuity of systems lessens as we move away from their boundary and peak manifestation. This means we can use our experience and mental models to produce abstractions that are practical and useful.

At its core, a simulation model initializes a system at a known state and then simulates the future states. In our abstraction of a system, we clearly understand the time our study (and our simulation model) starts. But the system's initial state may be unclear. The reason? Everything that took place before the start time contributes to the system's **initial state**.

> *If we can produce a reasonable initial state, we'll incorporate the effect of all prior influential interactions and events into the model. The initial state doesn't mean we only have scaler data related to that moment; it could include the accumulated historical knowledge up to that point. In other words, full information about the initial state includes all past knowledge and information about the system before the model's start time.*

When we aren't sure about the initial state, we can define probable scenarios for inputs and then run several simulation models. We'll discuss these types of analysis – what we call **experiments** – in later chapters. For now, we should use our knowledge of the initial state to help us choose when to start the model. That's a good start, but it's also hard to know when we should end the model. In fact, many systems have no clear end. Many others have an end that we can't estimate.

We can classify simulation models – and especially process-centric models – into two major groups by the way they end and how we analyze their outputs.

- A **terminating simulation** ends at an explicit point in time or when a specific event takes place or when the model satisfies a condition. We usually associate the stop time with a conclusion in the real system we're studying. But our model's stop time may not be the same as the real system's final state. It might just be the last point of time we deem useful.

  If we wanted to simulate a box office's ticket sales for a concert, we would start our simulation weeks or even months before the concert. But once the concert starts, there's no reason to continue our analysis of the ticket-selling process. If our model looks at the day tickets for the concert went on sale, it wouldn't make sense to continue it beyond that day.

- In theory, a **nonterminating simulation** never ends. This means there's no specific end that can be foreseen for the system; in practice, we're interested in the system's long-run outputs. We usually associate "long-run" with **steady state** simulations where the outputs converge into **stable**

values. "Stable values" doesn't mean the values are the same: it means they'll have the same distribution. This means we measure the outputs of a simulation that has reached a steady state by averaging the random outputs we observe at the same point in time during different runs (estimated expected value).

Comparing warm-up period to steady state

As mentioned above, we're interested in the steady state of a nonterminating simulation. However, a nonterminating simulation often starts with a warm up period before it enters a steady state. Let's review the difference between the warm-up period and the steady state.

Before a flow system starts, the flow channel is empty and all resources are idle. When entities start to arrive and fill the flowchart, resources move them forward. Think of a warehouse, where each workday begins with empty storage racks and idle forklifts. When trucks bring pallets to the unloading docks, forklifts move them to the storage racks. Assuming those trucks arrive at a consistent rate, the flow of products will reach a steady state.

We call this first unstable period -- one that ends once the system reaches a steady state -- the **initial transient period** or **warm-up period**. In a terminating simulation, the initial condition has a significant effect on the outcome. This means the initial condition should represent the actual system (Figure 2-27).

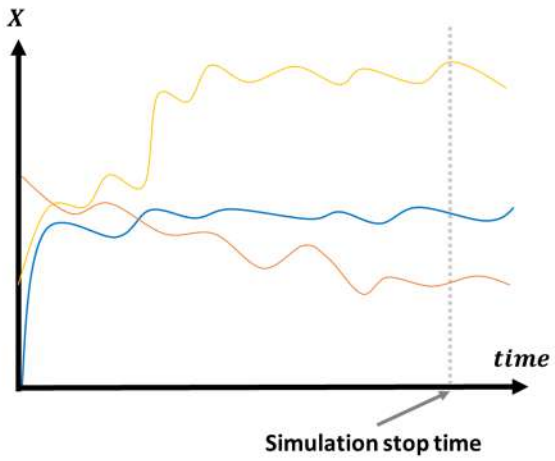


Figure 2-27: Several sample paths of a random process as the output of a terminating simulation

In contrast, the steady state in non-terminating models is independent of the initial values, although the initial state affects the convergence rate. Figure 2-28 (left) shows us that when we start the model with a specific initial value, each realization (that is, each simulation run for a specific initial state) during the warm-up period can be different. However, Figure 2-28 (right) shows us that once the model reaches the steady state, the ensemble's expected values (that is, the values from several runs) start to converge. In the end, the steady state value isn't sensitive to the initial condition.

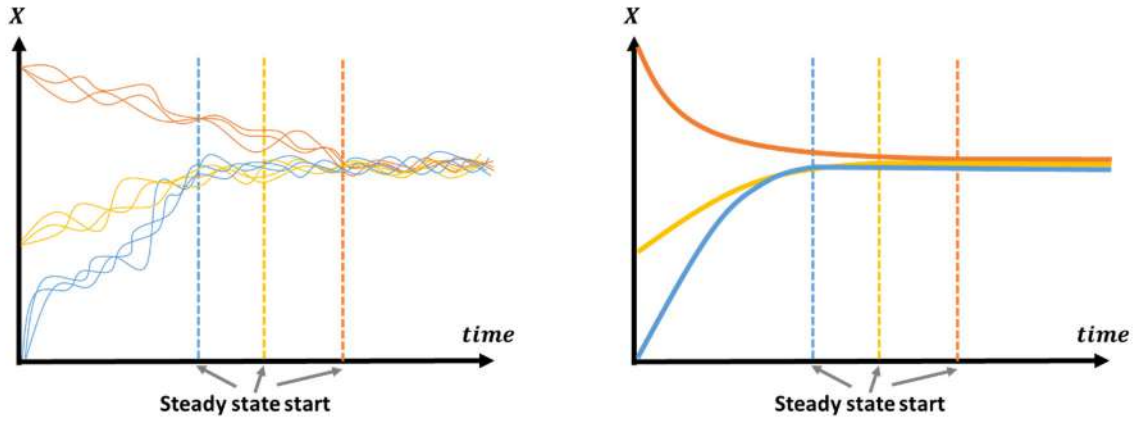


Figure 2-28: Output of simulation runs with different initial conditions (Left: sample paths, Right: Ensemble values)

With that in mind, why don't we start the simulation with the steady-state values (stationary distributions)? Those values are the emergent outcome: If we knew them, we wouldn't need the simulation.

The most important implication of knowing a flow system is in a steady *state* (which occurs when a system shows a consistent behavioral pattern) is the system's overall arrival rate and departure rate must be equal. This is because the number of entities in the system is the difference between the entities that entered the system and the entities that left it. When we have a stream of entities flowing through a system and the expected number of entities in the system is constant, the system's arrival and departure rates are equal (what comes in, gets out).

Remember that only stable simulation models reach the steady state. While we'll discuss how you can check the stability of simple queuing systems, most present-day simulation models include other factors and rules. They can include learning capability, adaptivity, complex algorithms, interactivity and real-time inputs. This means any convergence into a steady state (assuming a convergence is possible) might take place after an extended time or in cycles with long periods. It makes more sense to run complex models that don't have a specific end time for a *relatively* long period and treat them as terminating simulations in the output analysis phase.

The terms we introduce in this section apply to any simulation model (including agent-based models). In later sections, we'll return to simple process-centric models that have mathematical solutions with the help of queueing theory. These mathematical solutions work well for steady states and long-run output values – although they all start with empty and idle states and go through an initial transient period. If we want to know the initial transient period's outputs, a simulation may be our best option.

Process measurements and analysis

The pioneers of process-centric simulation models identified the key features and performance indicators we find in all processes. They used this information to develop a simulation methodology they could apply to a broad range of systems of flows.

As we mentioned, a process-centric simulation has three fundamental components:

1. **Entities**: These items that flow through the system are the system's main actors and the reason the system exists.
    - We can separate the time an entity spends in a process into two categories. *Necessary* time is the time the entity spends getting a service or completing a task with the help of resources. *Wasted* time is the time the entity spends in queues when there aren't enough resources.
2. **Resources**: The second most important actors in flow systems, resources help the process occur. Decision makers can use them to tweak the process toward a desired performance.
    - Since a lack of resources is the most common cause of unnecessary delays and wait times in a process, more resources are often better. However, we don't want more resources than we need because of the associated cost. Aside from cost, there might be hard constraints on the availability of resources. In these cases, decision makers should strategically allocate these limited resources.
3. **Flowcharts**: Think of a flowchart as a static channel that guides entities. When a simulation begins, entities flow through the flowchart much like fluid flows through a welded pipeline. You can see the flowchart's structure - the blocks that guide the entities - even before you execute the model.

Together, we can use these three principal components of process-centric models to model a wide range of flow systems across many industries and domains. But what might be most interesting is the performance measures for process centric models, including queuing systems, are mostly the same.

Classic measures of queueing system performance

We'll now focus on classic measures of performance in queueing systems. Since process-centric models inherited most of their logic from queueing systems, these measures are transferable. According to Shortle et al. (2018), we should look at three main categories of system outputs. The first two are related to entities and the third is related to resources:

1. Some measure of waiting time a typical entity might endure. We measure the time an entity spends in the queue and the time an entity spends in the system (that is, queue plus service). It's important to differentiate them: time with servers is necessary but time in the queue is wasted. Think of how you might wait two hours at the Department of Motor Vehicles to complete a process that only takes five minutes.
2. Some measure of the number of entities that may accumulate in the queue or the entire system. Like the first category, we must distinguish between the number of entities in the queue and the number of entities receiving a service. A production line that can only hold 50 cars can't expand its capacity, no matter how many cars are in the queue or are being worked on.

One way to measure this number is to use the associated space (physical or virtual) needed, particularly for waiting entities. For example, the space we need in front of a ticketing booth or the maximum number of people that might be on a waiting list.

3. Some idle-service measures can tell us the percentage of time a resource unit (server) may be idle or busy. In other words, they track the utilization of resource units (or servers). Contrary to the first two metrics that are related to entities, this metric helps us understand if available (or assigned) resources can support the flow of entities. While utilization is related to resources, maximum or high utilization levels lead to long queues and increased waiting times.

A decision maker who has these metrics can build a simulation model of a queuing system and gauge its as-is performance. Assuming the arrival rate of entities is an exogenous input and out of the decision maker's control, the next step is to find a balanced level of resources. The goal is to allocate the ideal amount of resources to ensure a smooth flow that minimizes the time entities wait. We also want to avoid the costs that occur when we overallocate resources.

As we discussed, we can classify simulation models into two major groups based on the way they end: Terminating and Non-terminating. If a non-terminating queueing system satisfies a certain condition (ergodicity), its output metrics will reach a steady state. The metrics we'll introduce in this section are for ergodic queuing systems at steady state (Figure 2-29).

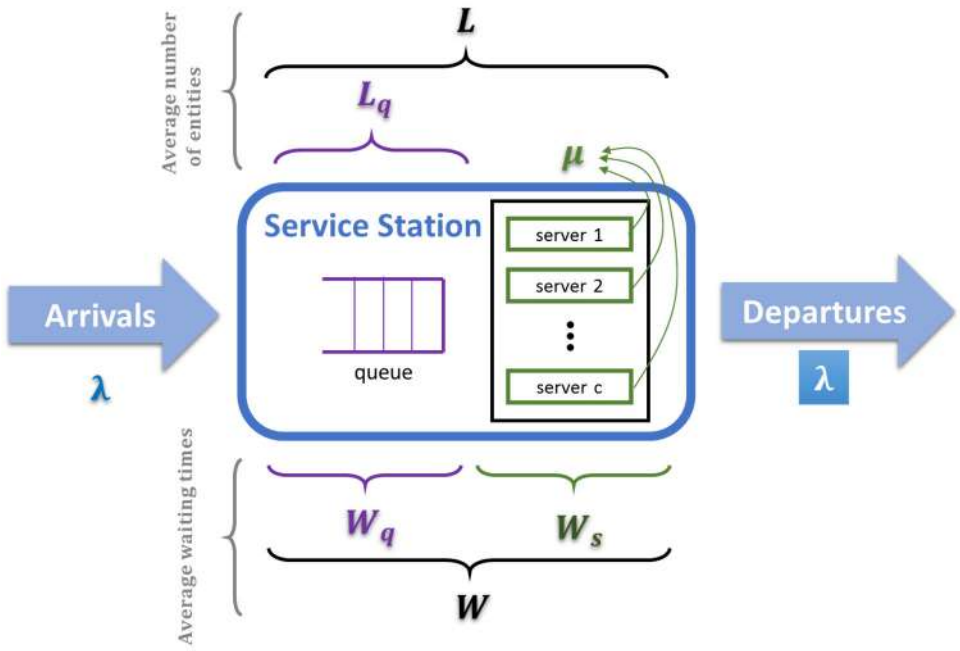


Figure 2-29: Measures of system performance

Knowing that

- $E[.]$ denotes the expectation operator
- $\equiv$ denotes definition
- $=$ denotes equality

Then we have the following definitions:

- A is the random interarrival *time*
- S is the random service *time*
- λ is the *average* arrival *rate*
- μ is the *average* service *rate* of one server
- c is the number of servers
- $r \equiv \frac{\lambda}{\mu}$, where r is the *average* number of busy servers
- $\rho \equiv \frac{\lambda}{c\mu}$, where ρ is utilization (or fraction of time) the server(s) is busy
- W is the *average* time an entity spends in the *system*
- W_q is the *average* time an entity spends in the *queue*
- W_s is the *average* time an entity spends with the *server*
- L is the time *average* number of entities in the *system*
- L_q is the time *average* number of entities in the *queue*

The condition of **ergodicity** is that average rate of arrivals must be strictly less than the *system's* maximum average service rate:

- $\rho < 1 \quad or \quad \lambda < c\mu$

Prerequisite formulas and relationships:

- $\lambda = \frac{1}{E[Interarrival\ time]} = \frac{1}{E[A]}$, average arrival rate based on random interarrival time
- $\mu = \frac{1}{E[service\ time]} = \frac{1}{E[S]}$, average service rate based on random service time
- $W_s = E[S]$, average time an entity spends in the server based on random service time
- $L = \lambda \times W$, Little's Law for the system
- $L_q = \lambda \times W_q$, Little' s Law for the queue

The most important prerequisite formula is Little's Law -- $L = \lambda \times W$ -- which we'll discuss in the next section. For now, we only need to know that Little's Law addresses the fundamental relationship among three key parameters we find in queueing systems: the average number of entities, their average waiting time, and their average arrival rate.

We're going to derive the other formulas from the definitions and prerequisite formulas we introduced above. At the risk of stating the obvious, the time an entity spends in the system is equal to the time it spends in the queue and in service ($W = W_q + W_s$). This means:

$$W = W_q + W_s \rightarrow W = W_q + E[S] \rightarrow W = W_q + \frac{1}{\mu}$$

We can substitute the above formula for W in Little's formula:

$$L = \lambda \times W \rightarrow L = \lambda\left(W_q + \frac{1}{\mu}\right) \rightarrow L = \lambda W_q + \frac{\lambda}{\mu}$$

$$L = L_q + r \rightarrow r = L - L_q$$

Remember that r is the average number of busy servers or equivalently the expected number of customers in service: $r = L - L_q$

In summary, the formulas we introduced are relationships between the performance metrics of a queueing system at steady state:

$if\ \rho < 1 \quad (or \quad \lambda < c\mu)\ then$:

- $L = \lambda \times W$ (Little's Law for the system)
- $L_q = \lambda \times W_q$(Little's Law for the queue)
- $W = W_q + \frac{1}{\mu}$
- $L = \lambda\left(W_q + \frac{1}{\mu}\right)$
- $L = L_q + r$

Since simulation models often include random variables (A and S) in their distributions and we're dealing with long-run values, we can substitute the expected values of these random variables in the above formulas:

$if\ \boldsymbol{E[A]} < \dfrac{\boldsymbol{E[S]}}{\boldsymbol{c}}\ then$:

- A is the random interarrival time
- S is the random service time
- $L = \frac{1}{E[A]} \times W$ (Little's Law for the system)
- $L_q = \frac{1}{E[A]} \times W_q$ (Little's Law for the queue)
- $W = W_q + E[S]$
- $L = \frac{1}{E[A]} \times \left(W_q + E[S]\right)$
- $L = L_q + \frac{E[S]}{E[A]}$

All these formulas are only valid for stable systems (ergodic). We consider a queueing system to be **stable** when the number of customers in the system remains finite and won't grow indefinitely. A stable system's main characteristic at a steady state is its long-run average departure rate is equal to the average arrival rate, as shown in Figure 2-29.

If we're dealing with an unstable system that doesn't have enough resources to handle the arrivals, the system's throughput (that is, the average departure rate) will be equal to its maximum ($c\mu$):

$if\ \boldsymbol{\rho < 1}\ (stable\ system)\ then$:	$if\ \boldsymbol{\rho \geq 1}\ (unstable\ system)\ then$:
• $utilization = \boldsymbol{\rho}$ • $throughput\ = \lambda$	• $utilization = \mathbf{1}$ • $throughput\ = c\mu$

Little's Law

Every process we model as a queuing system has a similar flow: entities arrive, they pass through one or many servers – possibly after they wait in a queue if there aren't enough resources– and then they depart. Figure 2-30 is a good example:

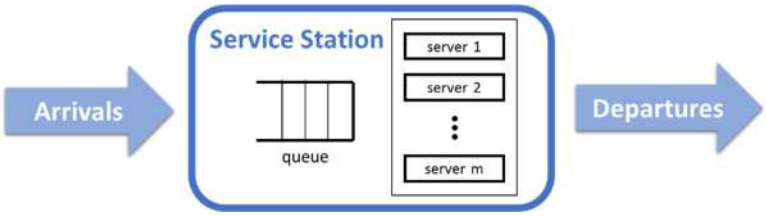


Figure 2-30: Generic queuing system

More than fifty years ago, John Little published the proof of a formula we now call Little's Law. It explains the relationships among three key parameters we find in a queueing system: the average number of items in the system, the average waiting time (or flow time) in the system, and the average arrival rate of items to the system. Little's Law applies to all types of flow systems including queueing systems with single queues and queuing networks; we can also apply it to all types of queueing discipline. Technically, the law doesn't require a "queue" - it simply needs a black box flow system where entities arrive and depart.

Little's Law has two main types of proofs: **sample path** and **stationary**. Sample path proofs tend to be more practical and generic, while stationary proofs focus on a more mathematical method. In this section, we present sample path proofs that apply to both transient and steady states. We first review two versions (version 1 & 2) of the law that apply to terminating systems. These versions focus on a specific realization of a queueing system in *finite* time periods.

When we analyze a sample path, we're working with realized values and averages over that sample path. In other words, versions 1 & 2 are associated with cases where a scenario for the queueing system has been realized (has happened) during a finite period and we have the historical data. Little's Law helps us establish a relationship among the scenario's three fundamental quantities. And since we're analyzing the system for a finite period, the queuing system doesn't have to be stable.

In the third version of the law, we'll discuss Little's Law and its applicability for queuing systems that are at a steady state. We can apply this version to any queuing system where the underlying stochastic processes are stationary (statistical properties of the process don't change). Please refer to math section to read more about the formal mathematical description of stationary stochastic processes.

Version 1. Little's Law for an empty system at times 0 and T

If $n(t)$ denotes the entities in the system at time t and we have a queueing system that is empty at the beginning and end – $n(0) = 0$ & $n(T) = 0$ - then let:

- $\lambda(T)$: average arrival rate in $[0, T]$ (entities/time unit)
- $W(T)$: average waiting time of an item during $[0, T]$ (time units)
- $L(T)$: the average number of items in the system during $[0, T]$

There's a general relationship among three quantities in such systems: $L(T) = \lambda(T) \times W(T)$

Theorem LL (Version 1): For a queuing system observed over $[0, T]$ that is empty at 0 and T, with $0 < T < \infty$, the formula $L(T) = \lambda(T) \times W(T)$ holds.

Assumption: System is empty at the beginning and end $(n(0) = 0 \ \& \ n(T) = 0)$,

Applicability: Holds under nonstationary conditions and is independent of queue discipline.

Little's Law (LL) tells us that if we know two out of three quantities, we can infer the third. This is useful because in many cases one of the three quantities is important to know but difficult to measure. Let's review a simple example that help us understand this concept.

Example determining number of groups waiting for service at a restaurant (Version 1)

A popular restaurant that opens at 5:00 pm does not let new groups enter after 6:00 pm. Since the staff must serve orders before 7:00 pm, the entire process takes just two hours.

The many groups that wait for a table convince the restaurant's manager to consider expansion. She soon learns the restaurant's pagers log the time a customer receives the pager. They also log the time the pager receives the signal to buzz. The manager wants to use this information to calculate the average number of groups waiting for a table. She first looks at the day's data and soon finds the 20 sample waiting times in Table 2-6.

Sample No.	Duration	Sample No.	Duration
1	26.298	11	36.196
2	23.492	12	23.439
3	15.189	13	15.194
4	27.871	14	42.976
5	36.420	15	20.074
6	11.037	16	48.329
7	38.704	17	43.187
8	31.622	18	15.116
9	29.296	19	38.378
10	13.858	20	37.863

Table 2-6: 20 samples of time spent waiting

$$\lambda(T) = average\ rate\ of\ arrivals = \frac{total\ number\ of\ arrivals}{time\ period\ in\ minutes} = \frac{20}{120} \cong 0.166\ \frac{groups}{minute}$$

$$W(T) = average\ time\ spent\ waiting\ = \frac{sum\ of\ duration\ of\ all\ samples}{number\ of\ samples} = \frac{\sum_{i=0}^{20} w_i}{20} = \frac{574.539}{20}$$
$$\cong 28.727\ minutes$$

$$L = \lambda \times W = 0.166 \times 28.727 = 4.769\ (5)\ groups\ waiting$$

As you can see in this example, arrivals are greater than what the restaurant can sustainably process (input rate $= 0.166$, average throughput = $\frac{1}{28.727} = 0.035$). This means we're dealing with an unstable system;

if the restaurant increased their service time, there's a high probability the queue would keep accumulating and eventually approach infinity.

In general, queueing theory focuses on systems that are stable over the long run. However, the two versions of Little's Law that address finite periods (type 1 & 2) can apply to any system - including unstable ones.

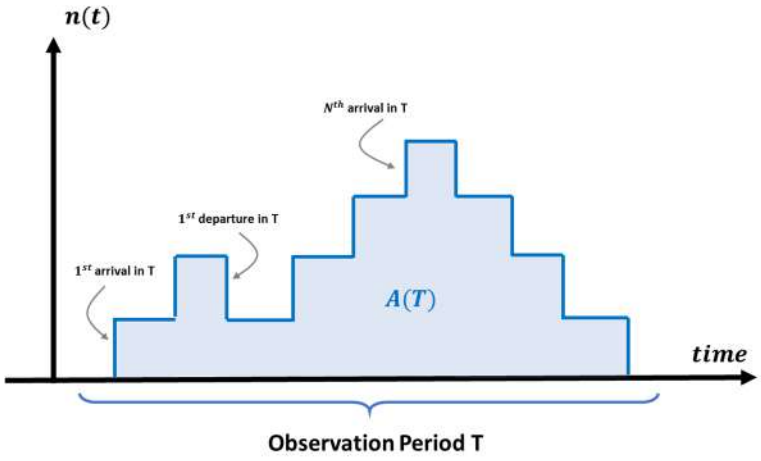


Figure 2-31: Version 1 of Little's Law: n(t) entities in system in period T, $n(0) = 0$ & $n(T) = 0$

Now that we've seen Little's Law in use, it's time to look at how we mathematically derive it. As shown in Figure 2-31, to show the precise mathematical statement of Little's Law let:

- $n(t)$: the number of entities in the system at time t;
- T: a relatively long period of time
- $A(T)$: the area under the curve $n(t)$ over the time T
- $N(T)$: the number of arrivals in the time T
- $\lambda(T) = \frac{N(T)}{T}$: arrival rate during time T
- $L(T) = \frac{A(T)}{T} = \frac{\int_o^T n(t)\,dt}{T}$: average queue length during time T
- $W(T) = \frac{A(T)}{N(T)}$: average waiting time in the system per arrival during time T

Proof: Using the notation above:

$$L(T) = \frac{A(T)}{T}$$

$$\lambda(T) = \frac{N(T)}{T}$$

$$W(T) = \frac{A(T)}{N(T)}$$

$$L(T) = \frac{A(T)}{T} = \frac{A(T)}{T} \times \frac{N(T)}{N(T)} = \frac{N(T)}{T} \times \frac{A(T)}{N(T)} = \lambda(T) \times W(T)$$

We arrived at $L(T) = \lambda(T) \times W(T)$ - which is the Little's Law formula for a finite period.

Version 2. Little's Law with permissible initial and final queues in $[0, T]$

If $n(t)$ denotes the number of entities in the system at time t and we have a queueing system that may not be empty at the beginning and end ($n(0) = n_0$ & $n(T) = n_T$), Let:

- $\lambda(T)$: average arrival rate in $[0, T]$ (entities per time unit)
- $W(T)$: average waiting time of an item during $[0, T]$ (time units)
- $L(T)$: average number of items in the system during $[0, T]$

Among these three, there's a general relationship in such systems:

$$L(T) = \lambda(T) \times W(T)$$

Theorem LL (2): For a queuing system observed over $[0, T]$ that has $0 < T < \infty$, the formula $L(T) = \lambda(T) \times W(T)$ holds.

Applicability: Holds under nonstationary conditions, is independent of queue discipline, and holds exactly even if the queuing system is never empty.

This is a more general form of version 1, which assumes an empty system at the beginning and end of time period T(Figure 2-31). In contrast, with version 2 we didn't assume the system starts or finishes empty (Figure 2-32).

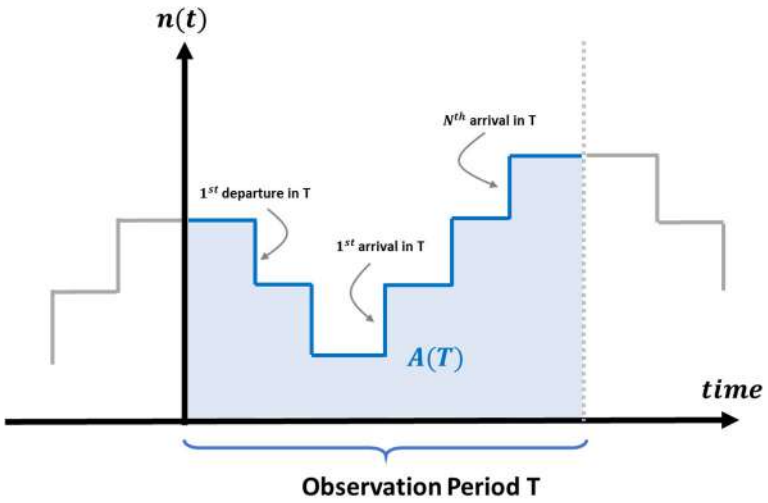


Figure 2-32: Version 2 of Little's Law: n(t) entities in system in period T, $n(0) = n_0$ & $n(T) = n_T$

Proof: Using the notation provided before and shown as in Figure 2-32:

Let,

- $n(t)$: the number of entities in the system at time t;
- T: a relatively long period
- $A(T)$: the area under the curve $n(t)$ over the time T
- $S(T)$: cumulative number of items in the system in the time T
- $S(0) = n(0) > 0$ *and* $n(T) > 0$

$$L(T) = \frac{A(T)}{T}$$

$$\lambda(T) = \frac{S(T)}{T}$$

$$W(T) = \frac{A(T)}{S(T)}$$

$$L(T) = \frac{A(T)}{T} = \frac{A(T)}{T} \times \frac{S(T)}{S(T)} = \frac{S(T)}{T} \times \frac{A(T)}{S(T)} = \lambda(T) \times W(T)$$

Online course example (Version 2)

A renowned university offers an online degree. When a student finishes their self-paced coursework and graduates, the university removes them from the program and closes their account.

The program has been active for the past 3 years, but we only have a record of last year's students. During the last academic year, 51300 students enrolled in the course and 23880 students graduated. We also have a table (not part of this example) that shows the length of enrollment for each student that graduated during the last year. This means we can calculate the $\mathrm{W(T)}$ *or the* average time to finish the course for those students.

The number of active students increases when students enroll and decreases when students graduate. Our goal is to calculate the average students enrolled during the last academic year. This is important because the university will use this number to allocate resources.

We can use last years' data to calculate $\lambda(\mathrm{T})$ and $\mathrm{W(T)}$:

$$\lambda(\mathrm{T}) = \text{average rate of new enrollements} = \frac{\text{all students enrolled during the year}}{\text{time period in days (a year)}} = \frac{51{,}300}{365}$$

$$\cong 140.55 \ \frac{\text{student}}{\text{day}}$$

$$\mathrm{W(T)} = \text{average time to finish the course}$$

$$= \frac{\text{sum of duration of enrollement for all students graduated last year}}{\text{number of graduated students}} = \frac{\sum_{\mathrm{i}=0}^{23880} \mathrm{w_i}}{23880}$$

$$\cong 62 \text{ days}$$

We know there were active students in the program when last year began. We also know there were enrolled students who were in the middle of their program at the end of last year. This means we can use version 2 of Little's Law to calculate the average number of actively enrolled students:

$$L(T) = average\ number\ of\ active\ students\ (unknown)$$

$$T = 365\ day$$

$$\lambda(T) = average\ rate\ of\ new\ enrollements = 140.55\ \frac{student}{day}$$

$$W(T) = average\ time\ to\ finish\ the\ course = 62\ days$$

$$L(T) = \lambda(T) \times W(T) = 140.55 \times 62 \cong 8714\ active\ students\ last\ year\ (average)$$

Version 3. Little's Law in steady state systems ($T \to \infty$)

The third version of Little's Law is for stable systems that run over a long time. Since their underlying stochastic processes are stationary, they will reach a steady state. Little's Law states that under steady state conditions, the average number of items in a queuing system equals the average rate at which items arrive, multiplied by the average time an item spends in the system.

Below is the third version of Little's Law based on the sample path viewpoint, but specific for steady states queuing systems:

$\lambda \stackrel{\text{def}}{=} \lim_{T\to\infty} \lambda(T)$

$W \stackrel{\text{def}}{=} \lim_{T\to\infty} W(T)$

$L \stackrel{\text{def}}{=} \lim_{T\to\infty} L(T)$

Theorem [Little's Law]*:* If the limits λ and W in the above formulas exist and are finite, the limit L exists and $L = \lambda \times W$

Assumption: Steady-state conditions (stationary conditions)

Applicability: Holds independent of queue discipline.

On the surface, this is like version 2 of the law - the only difference is the time interval T is no longer finite. When $T \to \infty$, we're dealing with long-run averages we see in the system's steady state; this requires a stable steady state to exist (that is, limits of λ and W exist and are finite).

However, version 3 of the law also differs from the others. The first two versions only hold for a finite period and are based on one real outcome (sample path) of the system in a specific period. Version 3 applies to mathematical solutions of long-run averages in queuing systems. In these mathematical solutions, the law holds both for a single sample path at the long-run or for the cross-sectional averages across many possible sample paths. For more information about this concept, refer to Ergodicity section).

This means each time we work with steady state mathematical solutions involving L , λ, or W and we know two of the three values, we can use Little's Law to calculate the exact true value of the third value in the steady state: $L = \lambda \times W$

Pilots at O'Hare Airport example (Version 3)

You're designing a new lounge for pilots of flights that depart from Chicago's O'Hare airport. You know 2400 flights leave O'Hare on an average day, which means 4800 pilots pass through the airport. Since pilots arrive two hours before takeoff, we want to calculate the average number of pilots at the airport to ensure the lounge can accommodate them.

In this example, we're not focusing on the historical data of a specific time span. Instead, our focus is on steady values in an ongoing operation. This means we're assuming the airport operation is in a steady state and there's a smooth inflow and outflow of flights. While the airport operation may vary, we can assume selecting the unit of time to be a day (24 hours) ensures the daily arrival and departure of pilots is close. We know this because each pilot who arrives at the airport will depart within the next 24 hours.

In a steady state system of flow, arrival and departure rates are the same. Since we know these pilots spend an average of two hours at the airport, we can calculate the daily average number of pilots in the airport:

$$L = \lambda \times W$$

$$L = 4800 \left[\frac{pilots}{day}\right] \times \frac{2}{24} [day] = 400 \; [pilots]$$

Alternative Version 3. Little's Law in manufacturing context (steady state systems)

Many manufacturing-related references use the third version of Little's Law (for systems in steady state). The law matches our previous descriptions, but the formula has more contextual labels.

Little's Law (manufacturing context): In the long run, the average work in process (WIP) is equal to the average throughput (TH) multiplied by the average cycle time (CT):

$$WIP = TH \times CT$$

- Work in Process (WIP): Number of items in the system (items)
- Throughput (TH): Rate at which items are processed (items/time)
- Cycle Time (CT): Time required for a unit to traverse the system (time).

Assumption*:* steady state conditions (stationary conditions).

Applicability*:* holds independent of queue discipline.

In the above formula, the following labels replaced the originals:

- $L \longrightarrow WIP$
- $\lambda \longrightarrow TH$
- $W \longrightarrow CT$

We should mention that contrary to λ (arrival rate), which is the system input, throughput (TH) is the system output. However, since we use version 3 of the law for systems that have reached a steady state, the long-run average arrival and departure rates should be interchangeable.

Applying Little's Law in queueing systems

We can apply Little's Law to any number of flow systems, including the queueing system in Figure 2-33. In this section, we assume our system is a single service station with a queue and server(s). If you apply Little's Law to the entire system, L will be the sum of the average number of entities in the queue and the server(s).

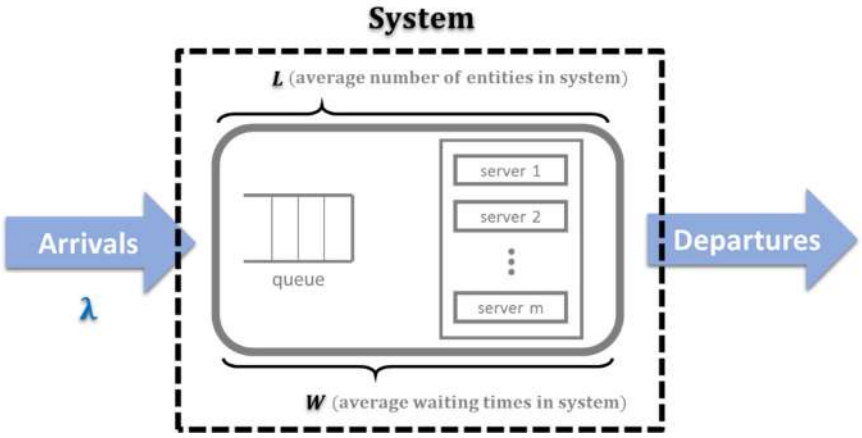


Figure 2-33: Little's Law applied to a system (queue + service)

We mentioned we can apply Little's Law to any black box system where entities arrive and depart. This means we can apply Little's Law just to the station's queue: $L_q = \lambda \times W_q$. Figure 2-34 shows this application.

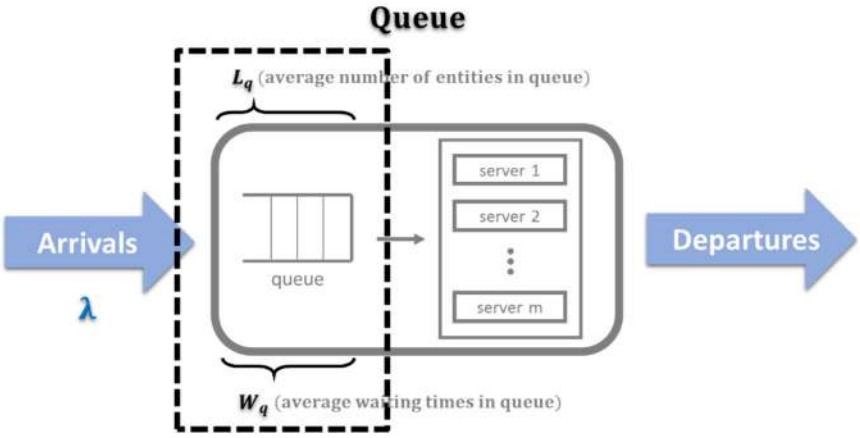


Figure 2-34: Little's Law applied to only the queue

For terminating simulation models of queuing systems, when the version 3's steady-state criterion isn't satisfied, the values of one terminating simulation run (a sample path) will still fall into version 1 or 2's definition. This is acceptable since versions 1 & 2 of the law uphold the relationship among the three key parameters under nonstationary conditions.

Version 3 of the law only applies to queuing systems in a steady state. This means the underlying stochastic processes should be stationary (having consistent statistical properties). This condition wasn't necessary for the first two versions.

For a queueing system to have a steady-state result, we must have the **ergodicity condition** where $\rho = \lambda/c\mu < 1$. If the system is ergodic $\left(\frac{\lambda}{c\mu} < 1\right)$ then during the transient period, the queue will fill until it reaches a steady state. When the system enters the steady state (stationary period), all the performance metrics (L, L_q, W, W_q) will become stable. From this moment, the average arrival rate and average departure rate will be the same, which means version 3 of Little's Law applies.

For simulation models of queuing systems to reach a steady state, technically we're supposed to run them infinitely. Since running an infinite simulation model is impossible, in practice, we run the model for a relatively long time. This ensures we gather many samples after the transient state which will be good estimates of the long-run values. In a simulation model, we associate "long-run" with steady state simulations where the outputs converge into stable values. Here, we use version 3 of Little's Law to estimate any of the three values based on the two others.

The important thing to remember is that the Little's Law is general. It holds for terminating and non-terminating simulations.

Queuing notation

Kendall (1953) established a notation that summarizes all attributes of a queueing system in a compact, easy-to-read format. We're using the abridged, five-letter version of this notation shown in Table 2-7.

Table 2-7: Kendall notation and definitions

$A/B/C/D/E$		
A	Interarrival time distribution	**M**: exponential or **M**arkovian (**M**emoryless) **D**: constant or **D**eterministic **G**: arbitrary or **G**eneral with known mean and variance
B	Service time distribution	
C	Number of parallel servers	Number of resources assigned to the service
D	System capacity	Number of entities that can be in the system – both waiting places in the queue and the number of servers; can be omitted if infinite – indicating that the process does not have a capacity limitation
E	Size of calling population	The population of all potential entities that at some point might enter the system; can be omitted if infinite – indicating a large population

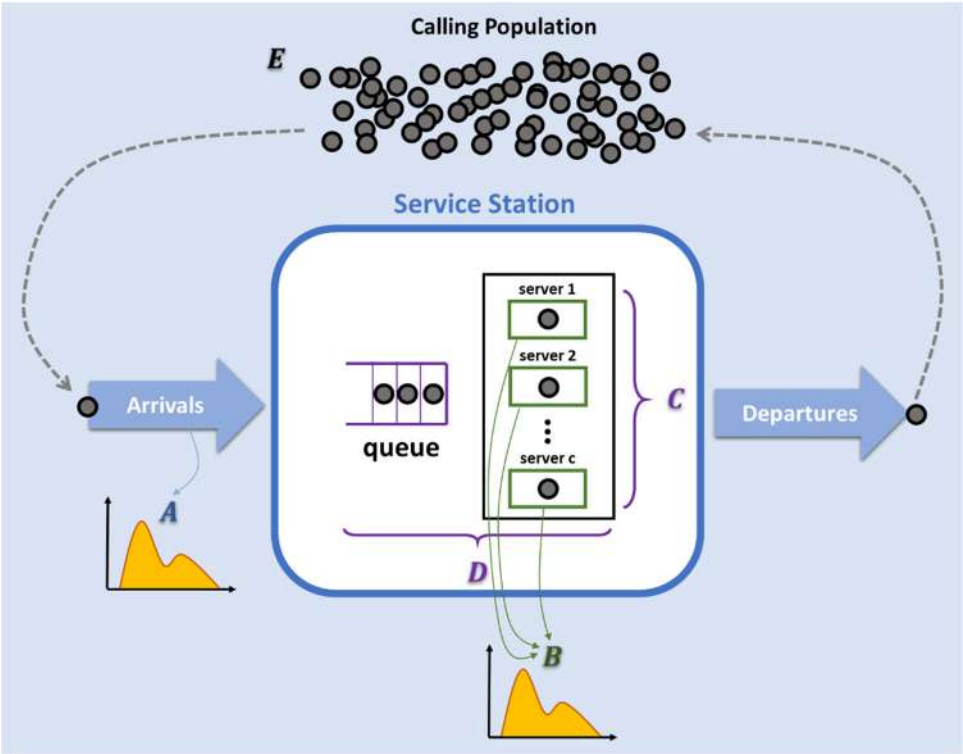


Figure 2-35: Kendall notation illustrated, E is the total number of entities (circles) in and out of the service station

In this book, we'll discuss some of the most important queueing systems, all of which we see in Table 2-8 below. We'll also review each system's **probability models** - closed form solutions of their performance metrics in steady state - and build their associated simulation models. We show you the mathematical

solutions of these archetype systems for your reference, but your focus should be on the simulation models that represent them.

Table 2-8: Archetype queueing systems covered in this chapter

Notation	Interarrival time	Service time	Number of parallel servers	System capacity	Calling population
	A	***B***	***C***	***D***	***E***
M/M/1/∞/∞	exponential	exponential	**1**	infinite queue	infinite
M/M/c/∞/∞	exponential	exponential	**c**	infinite queue	infinite
M/M/c/k/∞	exponential	exponential	**c**	**k**	infinite
M/M/c/c/∞	exponential	exponential	**c**	**c**	infinite
M/M/∞/∞/∞	exponential	exponential	Infinite / no resource constraint	infinite queue	infinite
M/M/c/∞/d	exponential	exponential	**c**	infinite queue	**d**
G/G/c/∞/∞	arbitrary with known mean and variance	arbitrary with known mean and variance	**c**	infinite queue	infinite
* Lowercase letters show the assigned integer values to specify the system					

Since these systems are easily modeled with modern simulation packages, don't treat them as a one-off model that you build once and then set aside. If this is your first exposure to queueing models, it's worth your time to look at each model and try to master their specifications and behaviors.

You can also use these archetypes as building blocks for more complex systems. We adapted nearly all the formulas in this chapter from the "Fundamentals of Queuing Theory" book by John F. Shortle et. al. If you want to learn more about the mathematical solutions and their proofs, that book will provide a wealth of information.

Anatomy of a queuing system model

Since common queuing system models have a generic structure, we'll first discuss the basic structure of a process centric model that adequately models a queuing system. The following information simply introduces some important ideas. We'll discuss step-by-step instructions for building each specific queueing system archetype later in this chapter.

Figure 2-36 shows the main flowchart we use to model the queuing system. It has a **Source** that generates entities, a **Seize** block with an internal queue that also reserves resource units from the **ResourcePool**, a **Delay** block that models serving times, a **Release** block that returns the resource units to the pool, and a **Sink** block that removes the entity from the flowchart.

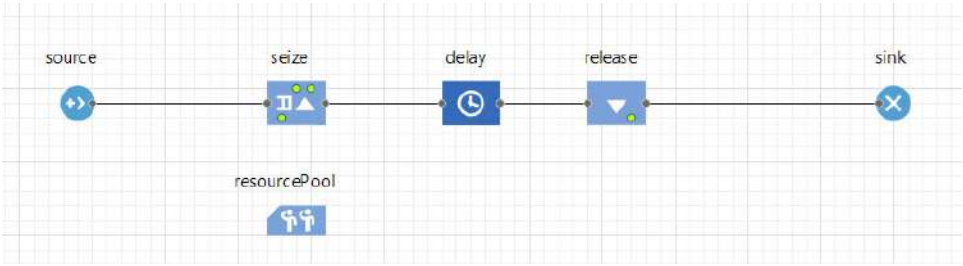


Figure 2-36: Basic structure of a queuing system model

a) Arrival process

Source (Figure 2-37) defines the arrival process by using a specified **Arrival Rate** or **Interarrival Time** to generate the entities. The arrival rate option models arrivals that follow a **Poisson process** (where the time between each arrival is drawn from an exponential distribution). Setting interarrival time provides a more generic option that lets you use any specified number or distribution (including exponential, as shown in Figure 2-37) to generate the times between each arrival.

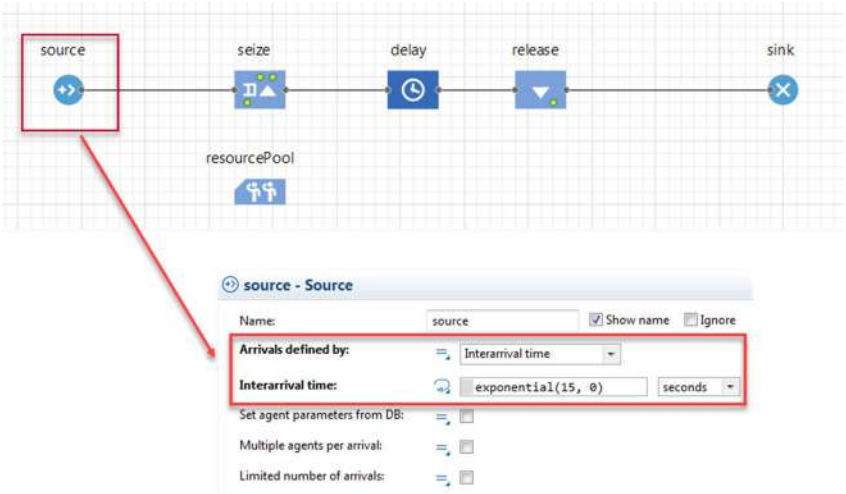


Figure 2-37: Source block using an exponential interarrival time with min = 0, shape = 15

Source also lets you inject multiple entities on each arrival or limit the number of arrivals (if the calling population is not infinite).

b) Queue and servers

In addition to being able to seize resource units from a pool, the **Seize** block's inner queue is the system's queue. By default, this inner queue has a FIFO queueing policy. You can set its capacity to the desired number or check the **Maximum queue capacity** checkbox to set an infinite capacity as shown in Figure 2-38.

ResourcePool's capacity sets the number of parallel servers in the queueing system. System capacity, which is the maximum number of entities that can be in the system at any point in time (in the queue and servers), is the sum of the **Seize** block's queue capacity and the capacity of the associated resource pool. **ResourcePool** also gives us utilization, which is an important system performance metric. We can access this by calling the `utilization()` method on the specified resource pool – using the model in the proceeding screenshot it would be called like so: `resourcePool.utilization()`.

During the model execution, you can click the resource pool block to monitor its utilization. If we only have one server (capacity = 1), utilization shows that server's utilization. If we have more than one server (parallel servers) then utilization shows the average utilization of all resource units/servers.

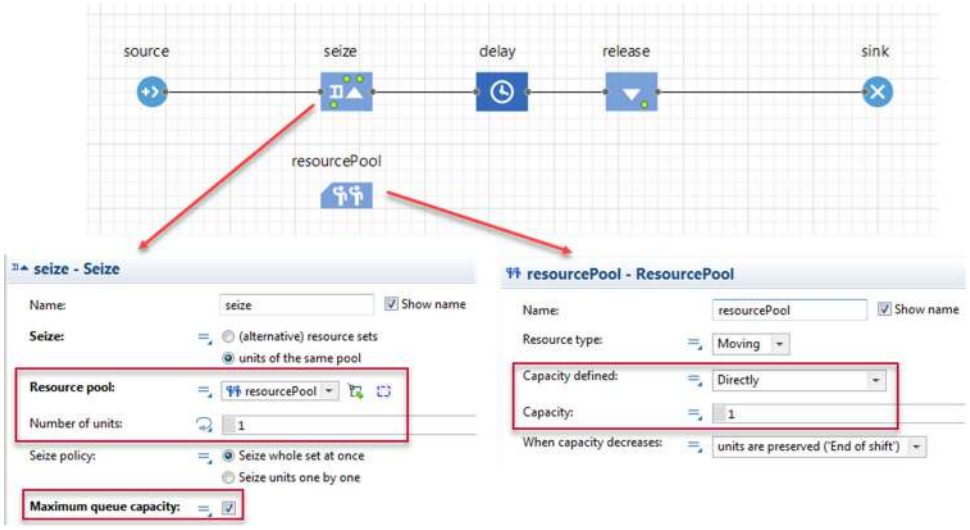


Figure 2-38: Example model with Seize and ResourcePool properties shown

c) Service time

Delay is responsible for modeling the service time associated with servers. You can assign any distribution to the delay, which all servers will then use for their service times. Figure 2-39 shows an exponentially distributed delay time with a minimum of 0 and the shape parameter set to 20.

When you use a **ResourcePool**, as shown below, always set the delay's capacity to maximum since the resource pool units should control the number of servers.

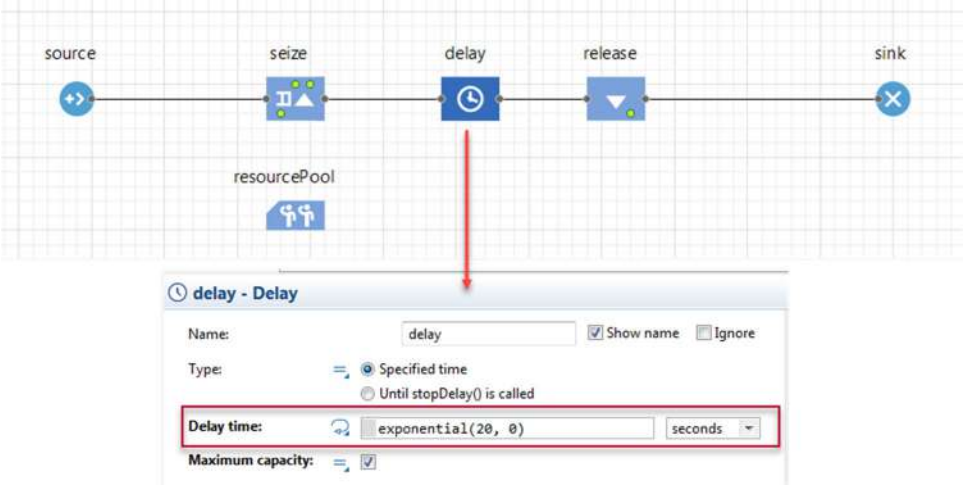


Figure 2-39: Example model with properties shown for the Delay block

As mentioned, the **Release** block is releasing the servers associated with entities. This is followed by the **Sink** block which removes entities from the flowchart.

d) Long-run average time spent in the system per entity (W)

To calculate the average time an entity spends in the system, we first need to calculate the time each entity spent in the system. In AnyLogic, we can use a pair of blocks -- **TimeMeasureStart** and **TimeMeasureEnd** -- to record the time an entity needs to traverse the process between them.

To calculate the time the entity spends in the system, we can add a **TimeMeasureStart** immediately after the **Source** block to measure when an entity first enters the system. We should also add a **TimeMeasureEnd** immediately before the **Sink** block to measure when an entity leaves the system.

The **TimeMeasureEnd** block's settings include an option of which **TimeMeasureStart** block you want it use to record the elapsed time. In our case, **tmE** is tied with **tmS**. For the **Dataset capacity** field in the **TimeMeasureEnd** block, we need to allocate space for the inner dataset that records each entity's travel time. We also need to make sure the capacity is large enough to record the travel time of *all* entities that pass through the flowchart – based on the expected incoming rate and the time or event that will stop the simulation.

In our example, we'll also add a histogram chart to show the distribution of travel times for the entities that reached the **TimeMeasureEnd** block. You can see these settings in Figure 2-40.

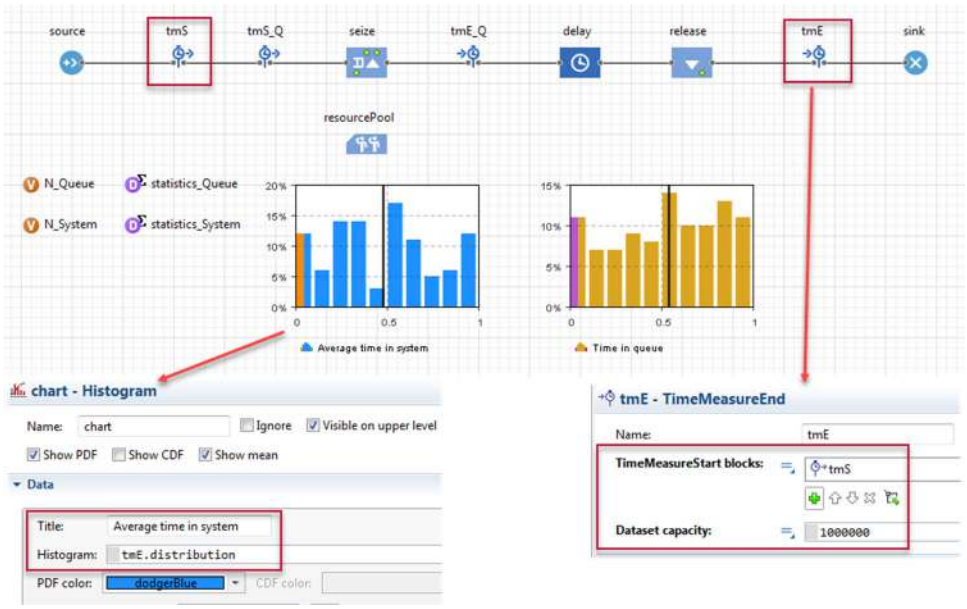


Figure 2-40: Example model with properties shown for Histogram and TimeMeasureEnd

e) Long-run average time spent in queue per entity (W_Q)

This is like calculating the average time spent in the system. The difference is that we're solely focusing on the time spent in the queue (which is in the **Seize** block). We've added a **TimeMeasureStart** (**tmS_Q**) and a **TimeMeasureEnd** (**tmE_Q**) around the seize block. We also coupled the **tmE_Q** block with **tmS_Q** and added another histogram to show the distribution of time spent in the queue (Figure 2-41).

Figure 2-41: Example model with properties shown for second Histogram and other TimeMeasureEnd

f) Long-run average number of entities in the system (L)

We need a variable to track the number of entities in the system. In our example model, it is an integer named **N_System**. We also need to update this variable's value whenever an entity enters the flow chart (**On exit** of **source**) or leaves the flow chart (**On enter** of **sink**).

To calculate the average number of entities across time, we need a **Statistics** object, set for continuous data. This object – you can find it under the AnyLogic palette's **Analysis** panel – can record data samples that persist in continuous time, which might change at discrete time moments. In our example, we named it **statistics_System**.

When the number of entities in the system is updated, we'll need to increment or decrement the **N_System** variable depending on if an entity has arrived to or departed from the system (Figure 2-42). The statistics object will also be updated by passing in the new value of **N_System** and the time the update took place (retrieved by calling `time()`). After the simulation ends, we can use the statistics object to get the mean value of **N_System**, which is the average number of entities in the system.

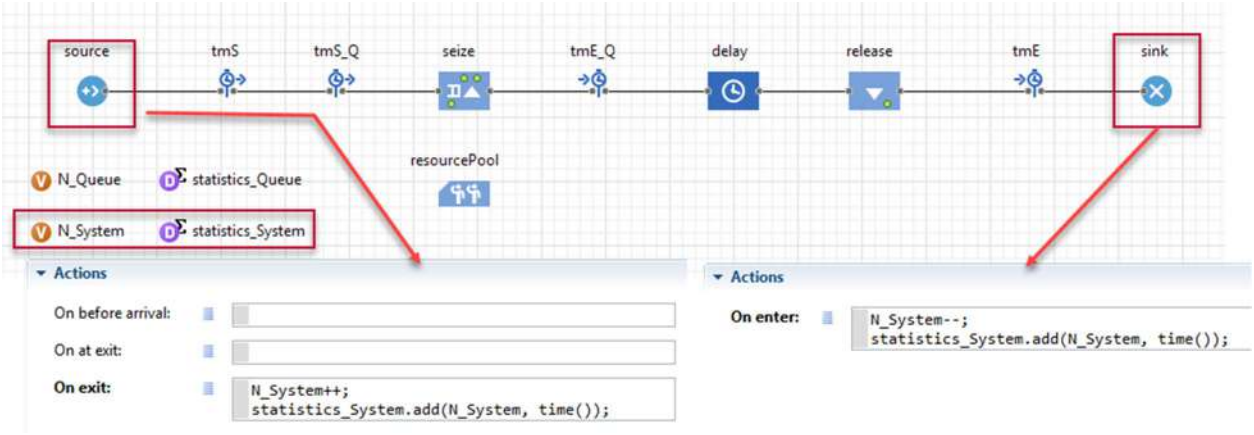


Figure 2-42: Example model with properties shown for the Source and Sink blocks

g) Long-run average number of entities in queue (L_Q)

This is like finding the average number of entities in the system across time. The difference is we're focusing solely on the entities in the queue.

We apply the same design to the queue as we did in the overall system: adding a variable (**N_Queue**) to track the number of entities in the queue, and a statistics object (**statistics_Queue**) to track the variable's time-average values. These changes occur when an entity enters or leaves the **Seize** block (*On enter* and **On exit**, respectively) as seen in Figure 2-43.

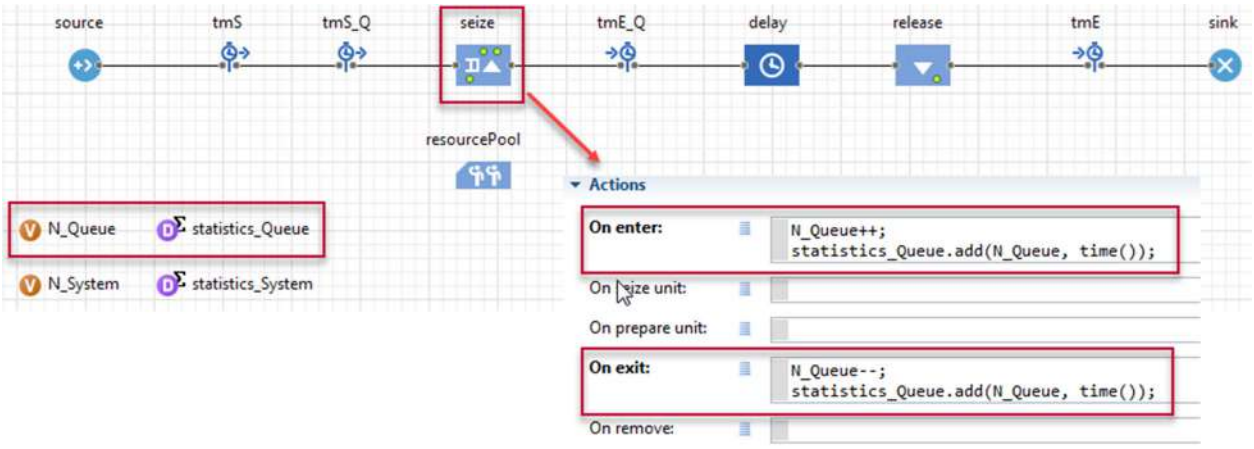


Figure 2-43: Example model with properties shown for the Seize block

Now that we understand the generic queueing system model, we're ready to build a M/M/1/∞/∞ system and modify it for other common queuing system archetypes. But before that, we need to explain how to calculate time-average for time persistent values such as the number of entities in the system.

Technique 3: Average number of entities in a system

Time average number of agents in the system is one of the metrics we can use to measure the performance of a process. It's easy to calculate the number of entities in the system at any time. One option is to subtract the number of entities that exited by the "*in*" port of **sink** from the number of agents that entered the model by the "*out*" port of **source**:

```
source.out.count() - sink.in.count()
```

You can also directly call the count() method directly for **Source** and **Sink** blocks:

```
source.count() - sink.count()
```

We could use this method for any Process Modeling Library (PML) block. In general, if you want to know the number of entities between two blocks, you can use:

{start_block_name}`.out.count() –` *{end_block_name}*`.in.count()`

In the arbitrary model shown in Figure 2-44, you can find the number of entities in **delay** and **service** by taking the number that left **queue** and subtracting the number that entered **service1**.

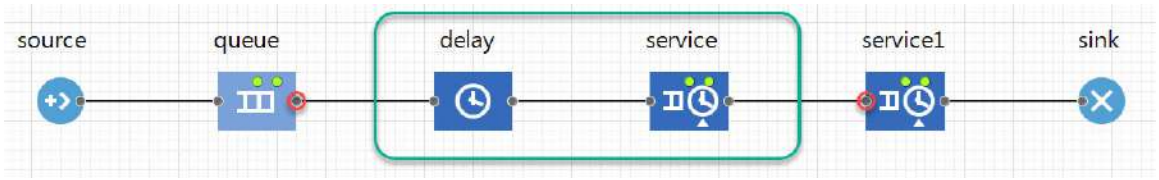


Figure 2-44: Calculating the total number of entities in delay and service

However, this method only calculates how many entities are in the system (NIS) *at a specific time*. As we know, that number will fluctuate. To determine the average, we must calculate a **time average**.

To visually explain this concept, take the example shown in Figure 2-45. It depicts some operation with time on the X-axis, ranging from zero to eight, and the number of entities in the system on the Y-axis. The chart tells us two entities entered the system at time 1, two entities joined them at time 5, and they all left the system at time 7. The system ran idly for another time unit before it ended at time 8.

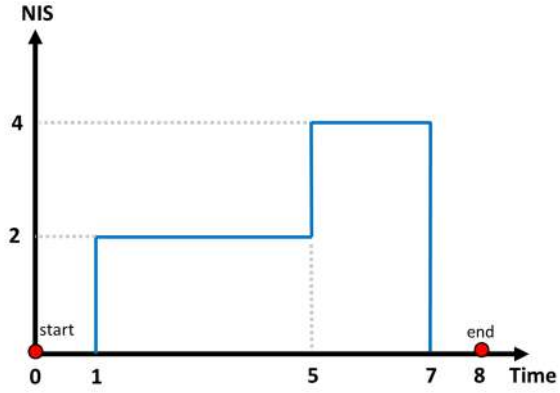


Figure 2-45: Examples of NIS

To determine the time average, we could record samples of the NIS values at predefined intervals, including the value at the start and end points, and then take their average. However, this approach might be inaccurate if the specified time step misses a change in numbers. For example, a cyclic sampling with a period of 2 will record 5 samples (including the end at time 8). From that we can get an average of these samples, as shown in Table 2-9.

Table 2-9: Five samples from the NIS

time	value
0	0
2	1
4	1
6	4
8	0
Average	**1.5**

By monitoring the changes, we can see we missed the NIS change at times 1, 5, and 7. This gave us an inaccurate average.

The correct method would be to calculate a weighted average of the NIS. To do so, we can calculate the area under the graph and divide the value by the time interval. In our example, the result will be $\frac{(4\times2)+(2\times4)}{8} = 2$.

In AnyLogic, the **Statistics** object from the Analysis palette can help us calculate the time average for the NIS. This object calculates several types of summary statistics (mean value, minimum, maximum, etc.) on a series of data samples for both discrete and continuous data. For our purposes, we would use the continuous option since it has a time duration.

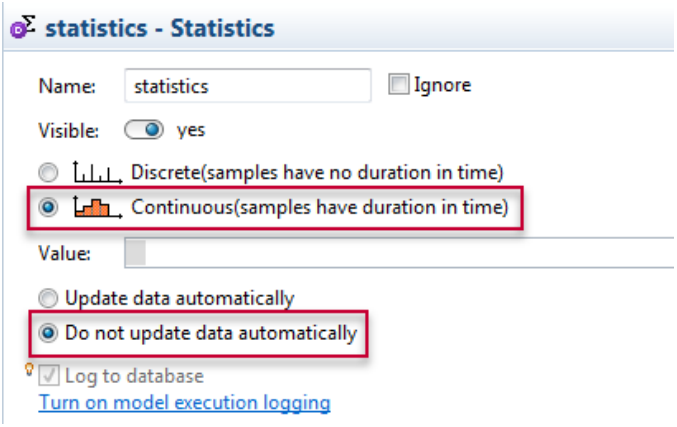


Figure 2-46: Statistics object properties set to 'Continuous' and 'Do not update data automatically'

To calculate the time average for the NIS, we'll manually add the data points at specific moments when the NIS changes. This means we need to disable automatic data updates which record samples at prespecified recurrence times as shown in Figure 2-46. As mentioned, if a small recurrence time is set which records most important changes, the automatic sampling method would be an approximation. If

the NIS doesn't change, automatic sampling would continually record unnecessary samples. This would increase the model's computational overhead without improving its accuracy.

With the continuous option, we add data samples and their timestamp to the **Statistics** object. To do so, we call the `add()` method of the statistics object and pass the value and time. For example, to pass the value of 5 with the current time of the simulation model to a Statistics object, named "statistics_1": `statistics_1.add(5, time())`.

When it calculates a continuous value, AnyLogic assumes the data samples keep their value until the next change. A new sample could change the current value at discrete moments (piece-wise constant, like a step function). The samples *must* be added to the statistics with ascending timestamps.

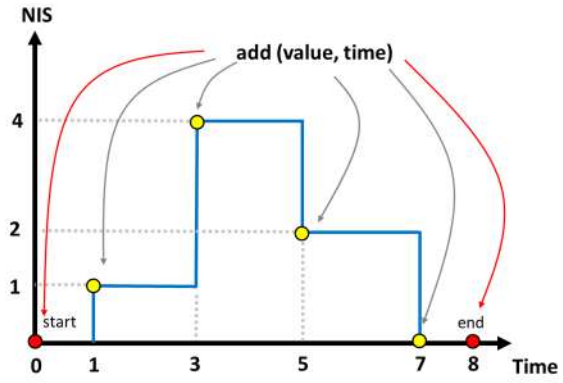


Figure 2-47: Adding data samples to the Statistics object with add({value}, {time}) method

One important caveat is to add the start and end values and times to the Statistics object (shown with red points in Figure 2-47). If we don't add them, the results will have errors. We might see models with short run times and boundary values that are significantly different from their closest point. For these, the result will be significantly different from the actual values.

In Figure 2-47 above, we can calculate the correct result manually by summing the NIS values multiplied by the time spent at that value, then dividing by the total time: $\frac{(0\times1)+(2\times1)+(2\times4)+(2\times2)+(0\times1)}{8} = 1.75.$

In this example, if we don't add the start and the end, the time average value will be calculated for the incorrect period of 6 units: $\frac{(2\times1)+(2\times4)+(2\times2)}{6} = 2.33.$

> *In long-run simulations where the start and end points are distant (timewise) and their values are like their closest non-zero point, we could ignore the effect of the boundary points. In our calculation of the long-run time averages (the number of entities spend in the system and queue), we did not add the boundary points, as we knew the differences would be negligible.*

To use the **Statistics** object to calculate the time average NIS value, do the following (shown in Figure 2-48):

1. Add a variable that records the current value of the Number in System (NIS) – it will be updated as the NIS changes during the simulation execution.

2. Whenever the NIS value changes, add the updated value and it's time to the continuous statistics with a line of code like this: `continuous_statistics_name.add(value, time());`

   To automate this process, you can add this piece of code in a callback of a block in which the change in NIS happens. For example, if we want to know the time average NIS of the entire flowchart, we need to update the **NIS_count** variable, **On exit** callback of source block and **On enter** field of sink block. Immediately after we update the variable, we add the updated value with its associated time stamp to the statistics object (**NIS_statistics** in this example). As you can see, the updated value of NIS will be added whenever an entity leaves the source or enters **sink**.

3. You must add the NIS values with their time stamps to the Statistics object until the simulation ends. At that time, we can call the `mean()` method of Statistics object to get the time average NIS. In this example, the code will be `NIS_statistics.mean()`.

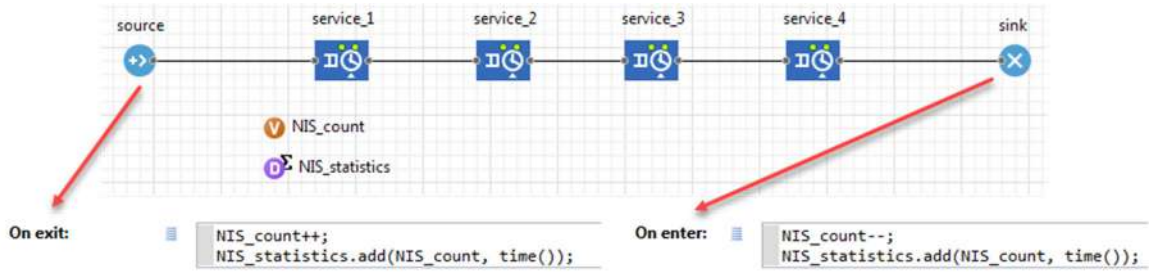


Figure 2-48: Snippets of code that update the Statistics object

In Figure 2-49, we run an example model for 1000 time steps and plot a time plot of NIS values at each time unit. There's also a time plot of the mean of Statistics object (**NIS_statistics**) which is the time average of NIS. You can see the time average starts to converge to a stable value as the model passes its first transient period. Although the long-run average approaches a steady state, transient period incorporation makes this a biased value. We'll discuss potential remedies in later chapters.

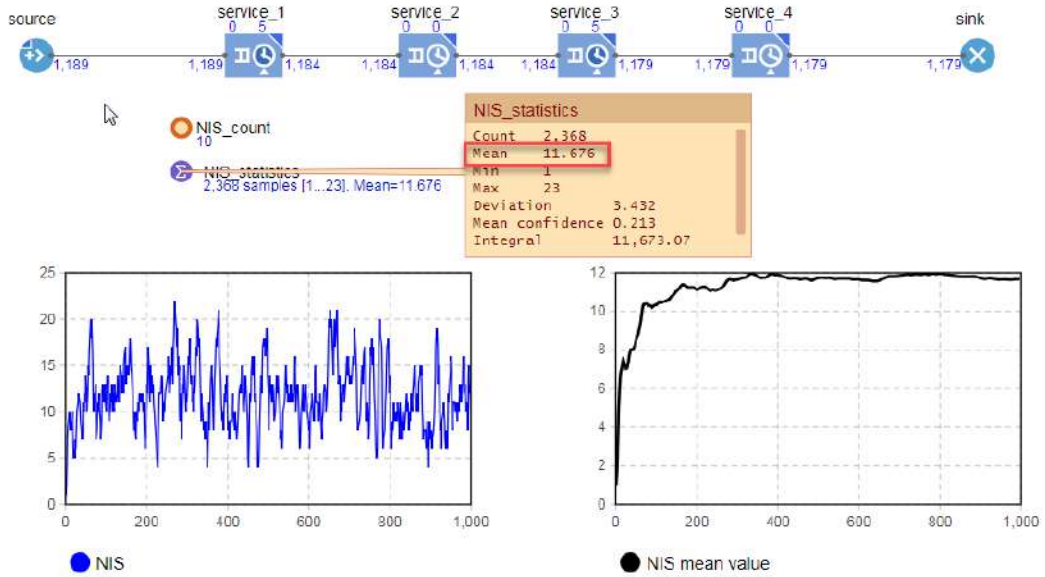


Figure 2-49: Time average of NIS (mean value of the continuous statistics)

M/M/1/∞/∞ (single-server queues)

As we mentioned, when the system capacity and size of calling population are infinite, we can omit them from the queuing notation - M/M/1/∞/∞ being the same as M/M/1. That said, we'll use the more complete notation throughout this manuscript. As a reminder, the M/M/1/∞/∞ notation means this queuing system (Figure 2-50) has the following characteristics:

- Interarrival time = exponential
- Service time = exponential
- Number of parallel servers = 1
- System capacity = ∞
- Calling population = ∞

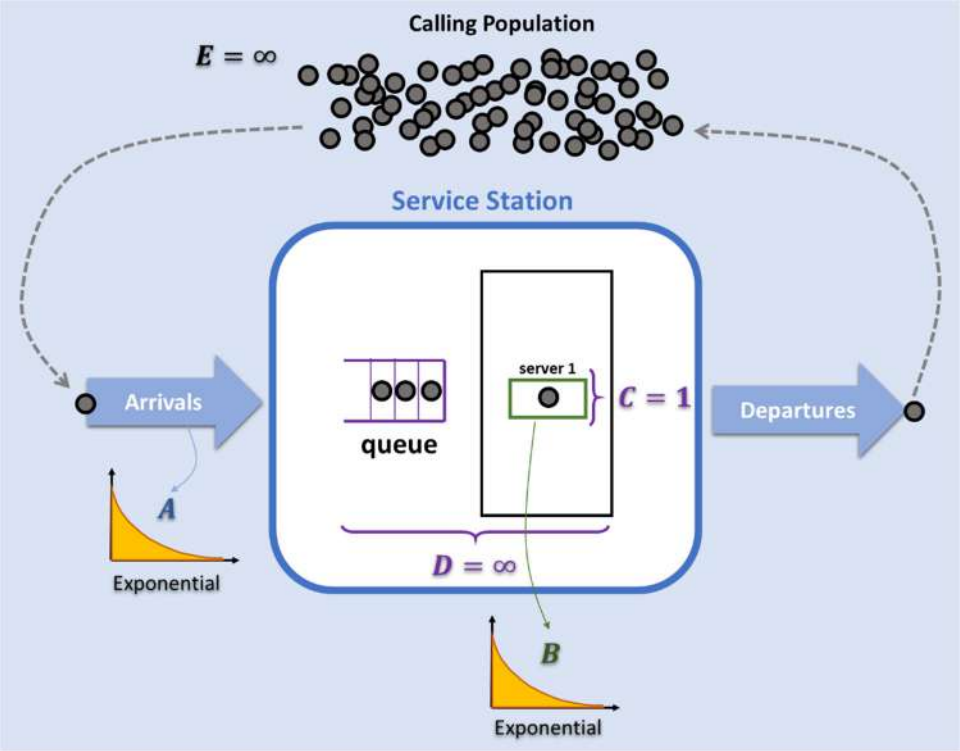


Figure 2-50: Illustration of M/M/1/∞/∞ (Single-Server Queues)

Examples of an M/M/1/∞/∞ queueing system:

- A line of passengers waiting to have their boarding pass scanned before boarding a plane
- A single ATM outside a grocery store
- A dentist office with a single receptionist to check in patients

Formulas for an M/M/1/∞/∞ queuing system:

$$\rho = \frac{\lambda}{\mu}$$

$$L_q = \frac{\rho^2}{1-\rho} \qquad W_q = \frac{\rho}{\mu(1-\rho)}$$

$$L = \frac{\rho}{1-\rho} \qquad W = \frac{1}{\mu(1-\rho)}$$

For a detailed explanation of these metrics (ρ, L, L_q, W, W_q), please refer to the **PROCESS MEASUREMENTS AND ANALYSIS** section. We included these formulas to show how you can calculate exact solutions, with the goal to compare them with simulation outputs. If you're interested in the mathematical proofs, you can learn more about them in references such as Shortle et. al, 2018 and Kleinrock, 1975.

In an M/M/1/∞/∞ system, arrival rates and service times are a **Poisson stream** with interarrival times that follow the exponential distribution (Figure 2-51). The main characteristic of a Poisson arrival rate is each arrival is independent and doesn't influence the arrival time of another entity; this applies to many real scenarios and is a useful feature. For a more thorough description of the Poisson process and its applications, please refer to the **POISSON PROCESS** section in the math **APPENDIX**.

In contrast, using exponential distribution for service time isn't realistic. The main reason we use exponential distribution for service times in legacy queuing systems is its nice mathematical properties help to find a closed form solution. Like arrivals, service times from an exponential distribution are memoryless (next state is independent of the current state). This means the time it takes for an entity to pass through the server doesn't depend on the server's current state.

Let's look at an example. If an expected service time is 10 seconds ($\lambda = 0.1$), and we follow an entity that has entered the service, our knowledge it has spent t time units in the server does not give us any information about when it will leave the server. In other words, knowing the entity has spent 3, 5, or even 11 seconds in the server does not give us extra information. The exponential distribution is the only memoryless continuous random distribution where the past has no bearing on its future behavior.

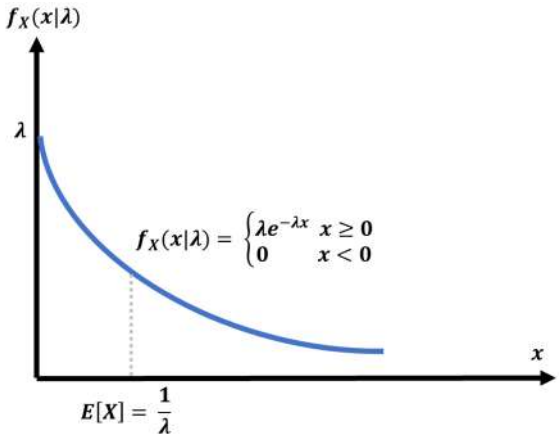


Figure 2-51: Exponential distribution with rate of λ

M/M/1/∞/∞ queue example (mathematical solution)

Assumptions for this example:

- $\lambda = \ the\ arrival\ rate = 15\ \left(\frac{entity}{second}\right)$
- $\mu = service\ rate\ of\ an\ individual\ server = 20\ \left(\frac{entity}{second}\right)$
- $c = \ number\ of\ parallel\ servers\ = 1\ \leftarrow\ (M/M/\mathbf{1})$
- $Queueing\ discipline\ is\ FIFO\ (First\ in\ First\ Out)$

Calculations of exact values:

$$\rho = \frac{\lambda}{\mu} = \frac{15}{20} = 0.75$$

$$L = \frac{\rho}{1-\rho} = \frac{0.75}{1-0.75} = 3$$

$$L_q = \frac{\rho^2}{1-\rho} = \frac{0.75^2}{1-0.75} = 2.25$$

$$W = \frac{1}{\mu(1-\rho)} = \frac{1}{20\times(1-0.75)} = 0.2$$

$$W_q = \frac{\rho}{\mu(1-\rho)} = \frac{0.75}{20\times(1-0.75)} = 0.15$$

M/M/1/∞/∞ queue example (simulation model)

Prerequisites:

Model Building Blocks (Level 1): **Source**, **Service**, **Sink**, **TimeMeasureStart**, **TimeMeasureEnd**, and **ResourcePool** blocks.

Math: Random variable, Random variate, [Poisson process], Exponential Distribution

The model we'll build in the following example will be generic example that will serve as a baseline for modeling the other common queuing systems. We use the same assumption for this model as we did for the mathematical solution.

To build the M/M/1/∞/∞ system with the input data mentioned above, do the following:

1. Create a new model, keeping all options in the input dialog at their default values.
2. Add a **Source** from the Process Modeling Library (PML) and change the **Arrival rate** to 15 per second (Figure 2-52).

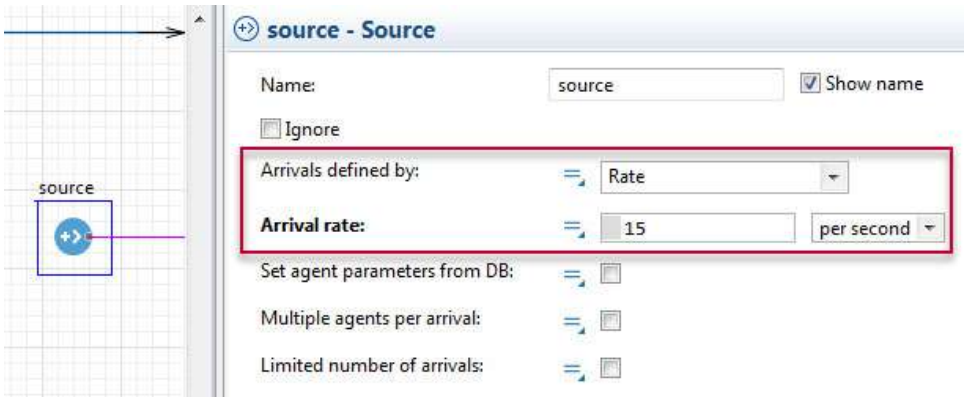


Figure 2-52: Arrival defined by Rate

In AnyLogic, arrivals defined by **Rate** means the arrival follows a Poisson process, meaning the interarrival times are exponentially distributed. As an alternative preferred for this example, we can define the arrivals by **Interarrival time** and explicitly use an exponential distribution.

- An important caveat: you must pass the rate or shape parameter (λ) to the exponential distribution – this being inverse of interarrival time (If the arrival rate/shape parameter is λ , then the expected interarrival time between entities is $\frac{1}{\lambda}$)
- With a minimum value of 0 and a shape parameter of 15, the function `exponential(15,0)` will define the interarrival time (Figure 2-53). To visually generate the probability distribution function, click the '  ' icon on the top bar and use the **Choose Probability Distribution** wizard.

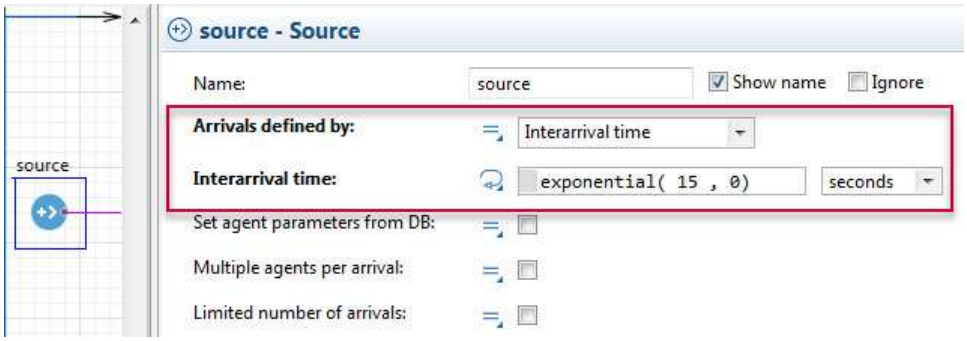


Figure 2-53: Arrival defined by Interarrival times (analogous to rate = 15 per second)

3. Add a **Seize** from the PML and connect it to the **Source** by dragging it slightly to the right of the exiting source block, where it will auto-connect. Click the **seize** block and select the **Maximum Capacity** checkbox to set its capacity to infinite (Figure 2-54). In Step 6, we'll return to assign a resource pool.

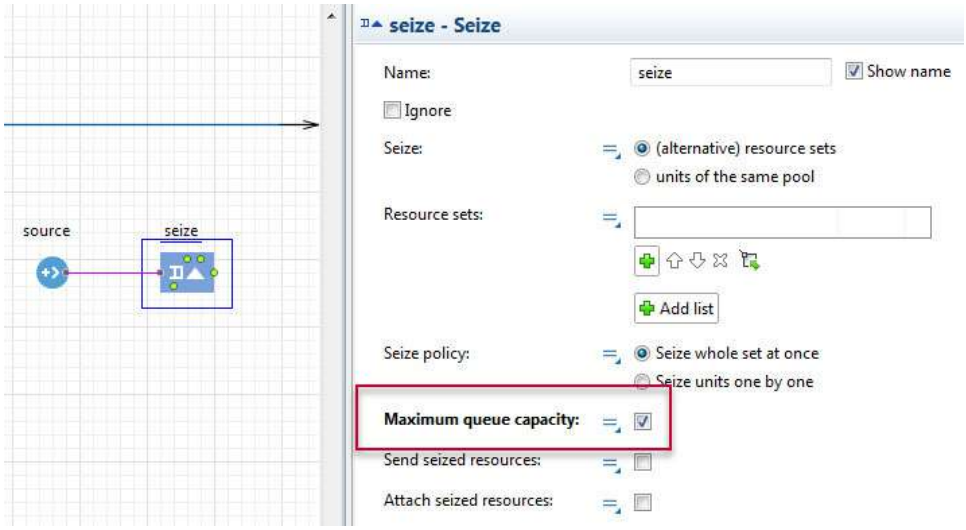


Figure 2-54: Maximum capacity for the Seize queue

4. Add a **Delay** from the PML and drag it next to the **seize** block to have it auto-connect (Figure 2-55).
    a. Change the **Delay time** distribution to `exponential(20,0)` to set the service rate to handle about 20 entities per second (that is, occurs for about 1/20th of a second).
    b. Select the **Maximum capacity** checkbox since the set number of resource units in the resource pool units will restrict the capacity.

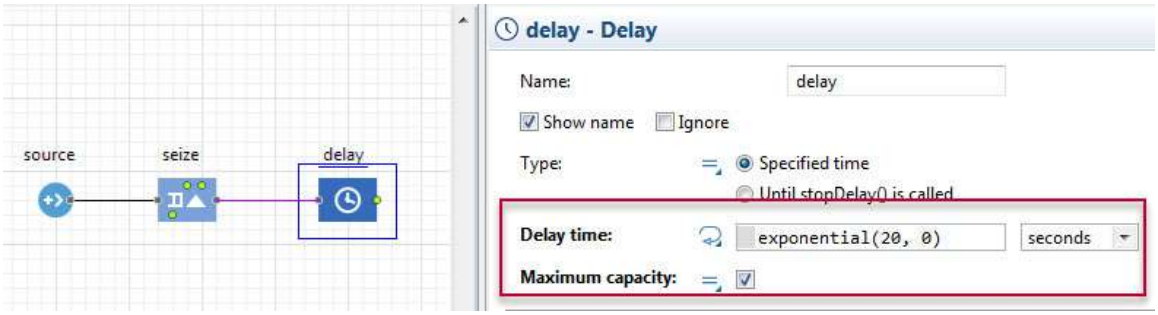


Figure 2-55: A delay time that draws its values from an exponential distribution

5. Add a **ResourcePool** and keep its default properties.
6. Click the **seize** block and change its **Seize** option to **units of the same pool**. Click the **Resource Pool** drop-down list and select the **resourcePool** you just added (Figure 2-56). You could also choose **resourcePool** from the graphical editor by clicking the graphical element chooser to the right of the drop-down menu.

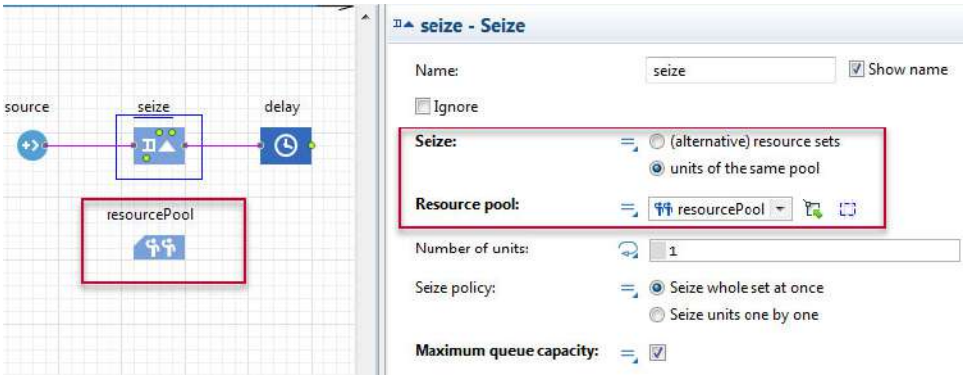


Figure 2-56: Assigning the resource pool to the seize block

7. Add a **Release** from the PML after the delay block; keep its default properties (Figure 2-57).

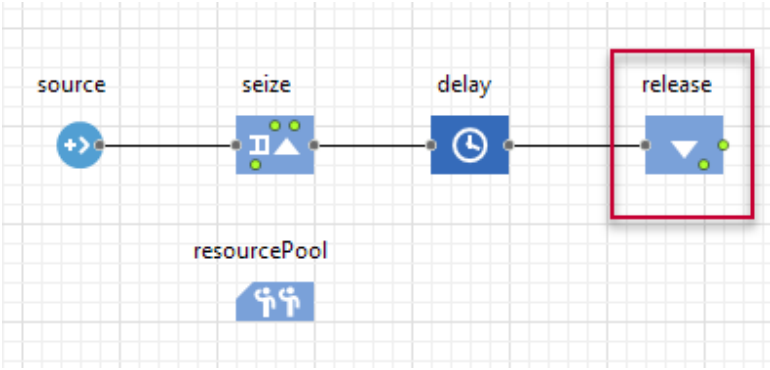


Figure 2-57: Release block that will release a seized resource unit

8. After the **release** block, add a **Sink** from the PML (Figure 2-58).

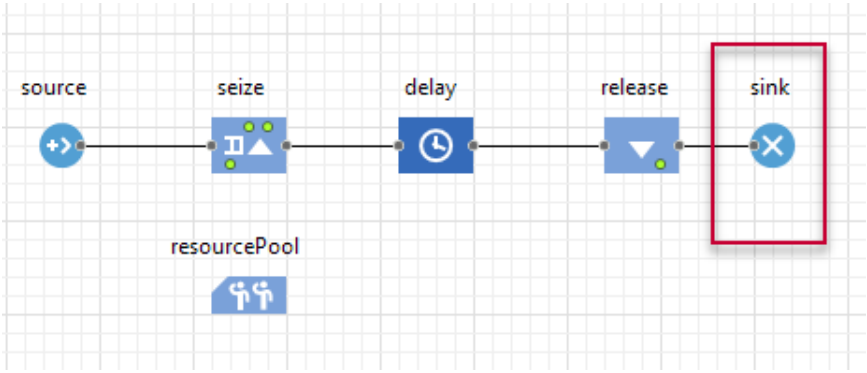


Figure 2-58: Sink block to remove the entities that reach to the end of the process

9. Drag a **TimeMeasureStart** from the PML and add it after the **source** block. Change its name to **tmS** (Figure 2-59). You may need to move the **source** to the left or the other blocks to the right to make room.

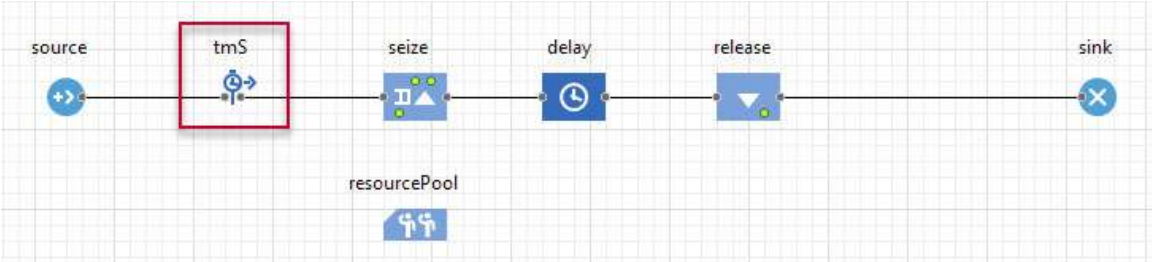


Figure 2-59: TimeMeasureStart which records the timestamp of entities that pass through

10. Drag a **TimeMeasureEnd** from the PML and add it after the release block. Change its name to **tmE** (Figure 2-60 and Figure 2-61).
    a. For the **TimeMeasureStart blocks** option, assign it to **tmS**.
    b. Set **Dataset capacity** to 1000000 (one million) to make sure we have enough storage to handle the entities that will pass through the process. We chose this number after we estimated the number of entities that will pass through the system. If you keep the default value of 100, the system will only keep the last 100 data points.

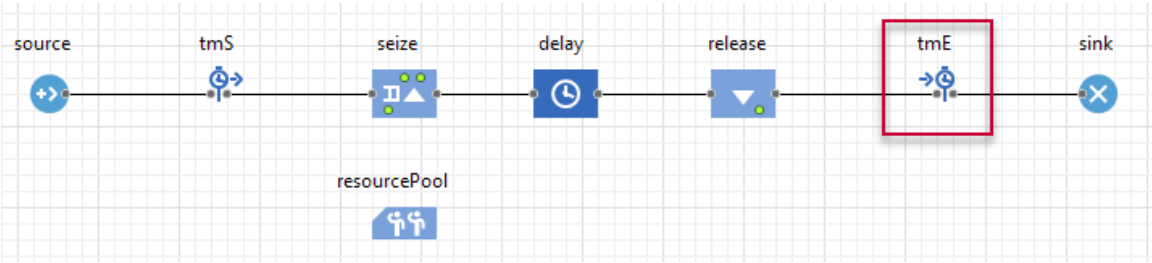


Figure 2-60: TimeMeasureEnd that calculates the elapsed time of an entity from the coupled TimeMeasureStart

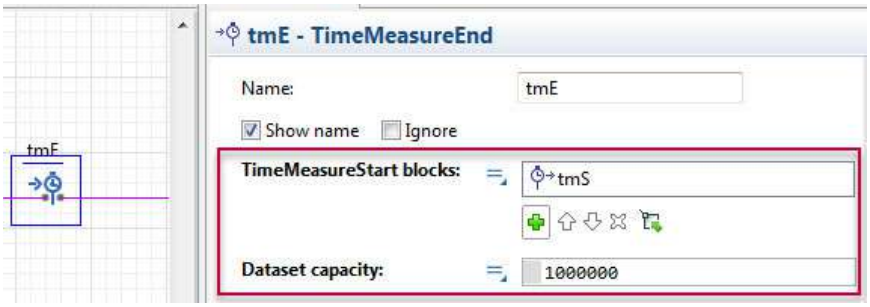


Figure 2-61: Associating a specific TimeMeasureStart and setting the capacity of its embedded dataset

11. To measure the time spent in the inner queue of the **seize** block, add a **TimeMeasureStart** before the seize block and a **TimeMeasureEnd** block after, naming them respectively **tmS_Q** and **tmE_Q** (Figure 2-62).

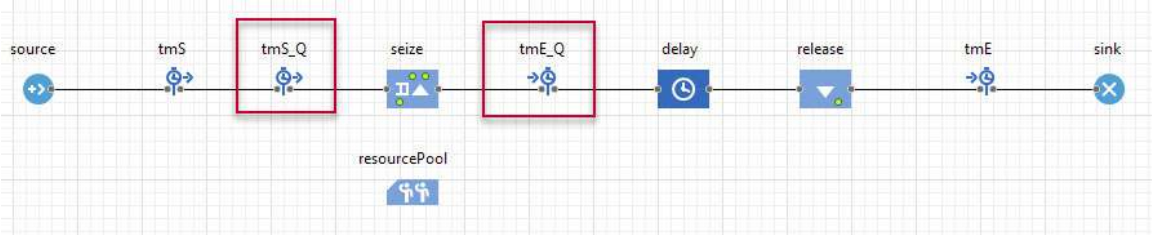


Figure 2-62: Adding another pair of time measure blocks

12. Click **tmE_Q**. For the **TimeMeasureStart blocks** setting, assign **tmS_Q**. Then, set the **Dataset capacity** to 1000000 (one million) (Figure 2-63).

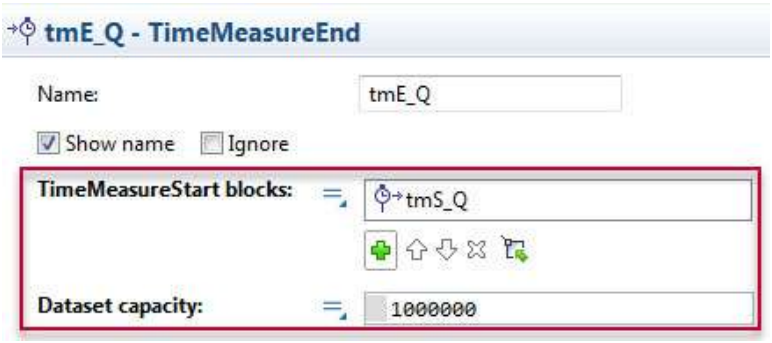


Figure 2-63: Associating a specific TimeMeasureStart and setting the capacity of its embedded dataset

13. Add a **Variable** object from the Agent palette, name it **N_Queue**, and change its type to *int*.
14. Duplicate **N_Queue** (by copy/paste or Ctrl-dragging) and rename the new variable to **N_System** (Figure 2-64).

    Copying an object also copies its properties, which serves as a time-saver in that we don't need to reset all the properties. These two integer variables will help us calculate the average number of entities in the system and the queue (embedded inside the seize block).

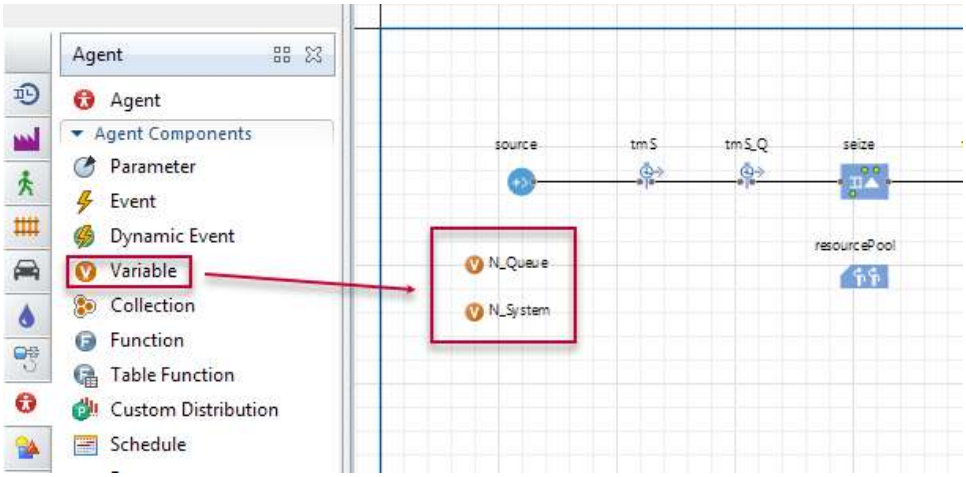


Figure 2-64: Adding variables to keep track of the number of entities in queue and system

15. Add a **Statistics** object from **Analysis** palette and name it **statistics_Queue** (Figure 2-65). In its properties, select the options for **Continuous (samples have duration in time)** and **Do not update data automatically** (Figure 2-66).

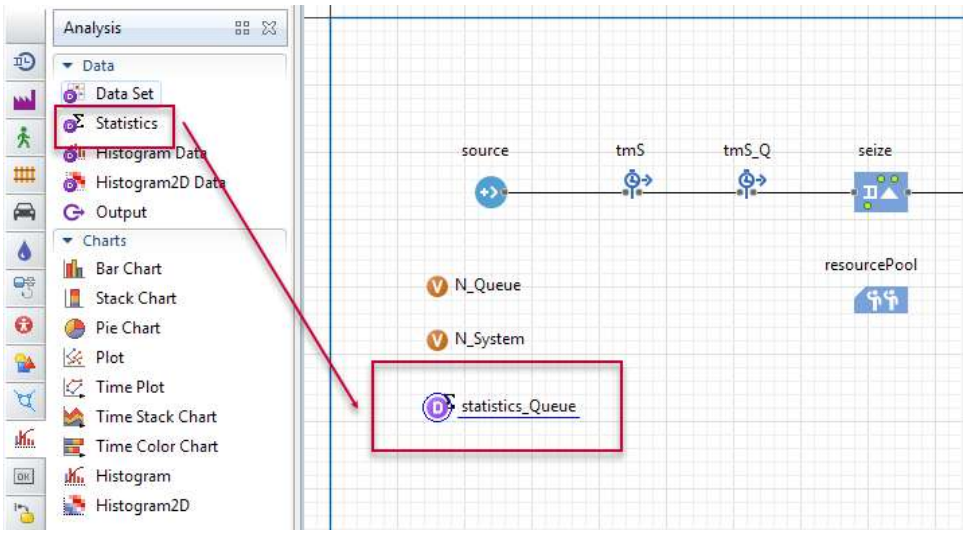


Figure 2-65: Adding a Statistics object from Analysis palette

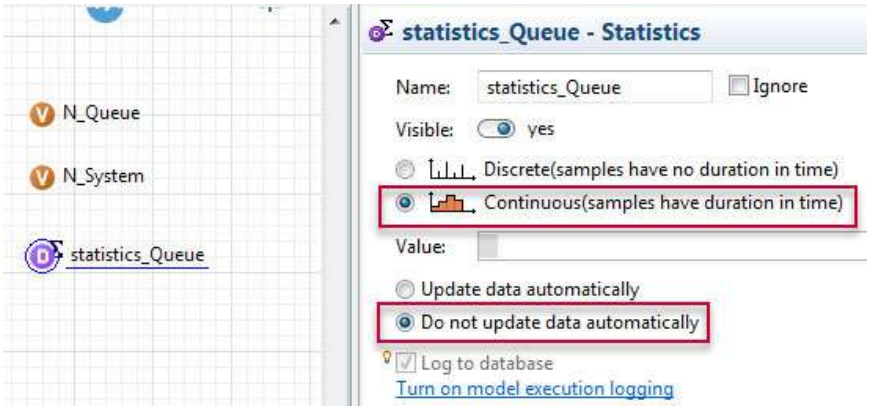


Figure 2-66: Statistics configuration

16. Duplicate the **Statistics** objects (in the same way stated in step 14 for the variable) and rename it **statistics_System** (Figure 2-67).

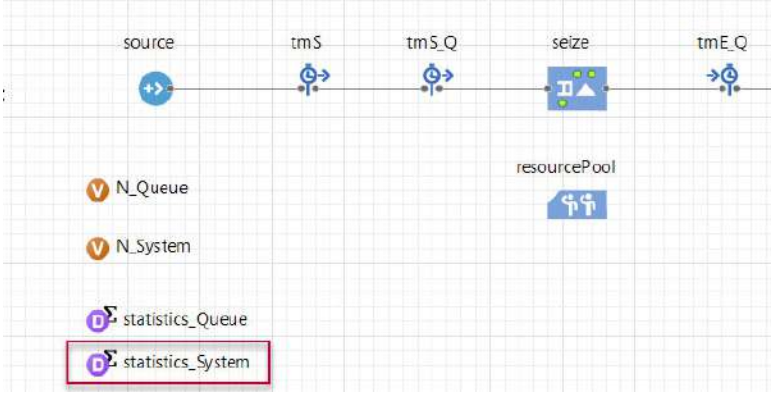


Figure 2-67: Adding a Statistics object to record the time average number of entities in the system

17. We must update **statistics_System** whenever a new entity enters the system (**On exit** of **source**) and whenever an entity leaves the system (**On enter** of **sink**). To do so, click the **source** and add the lines of code shown in the figure below to its **On exit** field (Figure 2-68).

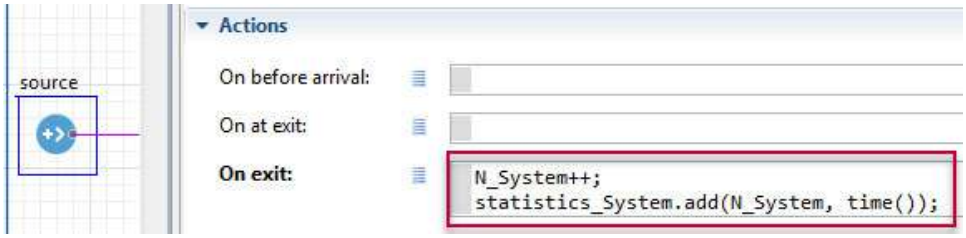


Figure 2-68: Updating the number of entities whenever a new one enters the system

The first line increments the **N_System** variable by one. The second line updates the value of the **statistics_System** with the new value and its time.

18. Click the **sink** block and add the lines of code shown in the figure below to its **On enter** field (Figure 2-69). The first line decrements **N_System** by one and then the second line of code assigns the updated value and the current time to **statistics_System**.

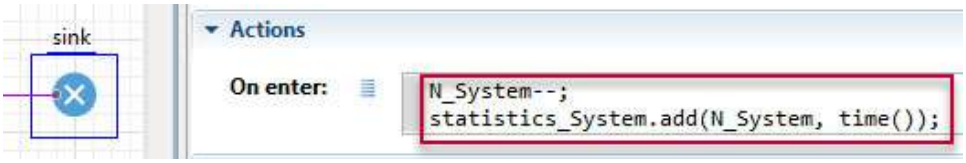


Figure 2-69: Updating the number of entities whenever an entity exits the system

19. We must add similar code shown in the previous steps to update **statistics_Queue** whenever an entity enters or leaves the inner queue of the **seize** block. In the properties of the **seize** block, write the code in the figure below in its **On enter** and **On exit** fields (Figure 2-70).

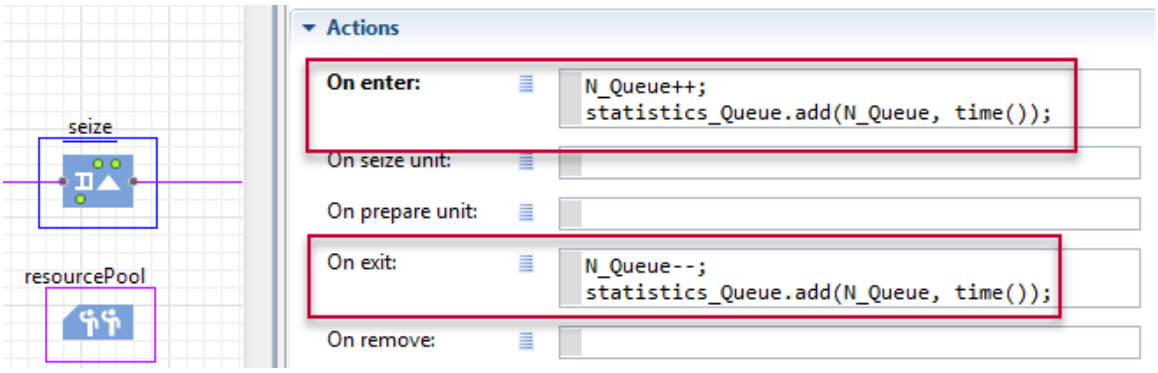


Figure 2-70: Updating the number of entities whenever an entity enters or exits the queue

20. Add the histogram showing the waiting time distribution in the overall system and the queue (more specifically, the inner queue of the **seize** block) (Figure 2-71).

    a. Right-click **tmE** and select **Create Chart** -> **Distribution**.

    b. Click the **chart** and change the title of the data to "Average time in the system".

    c. Check the **Show mean** checkbox.

    d. Move, resize, and customize the **chart** as you like.

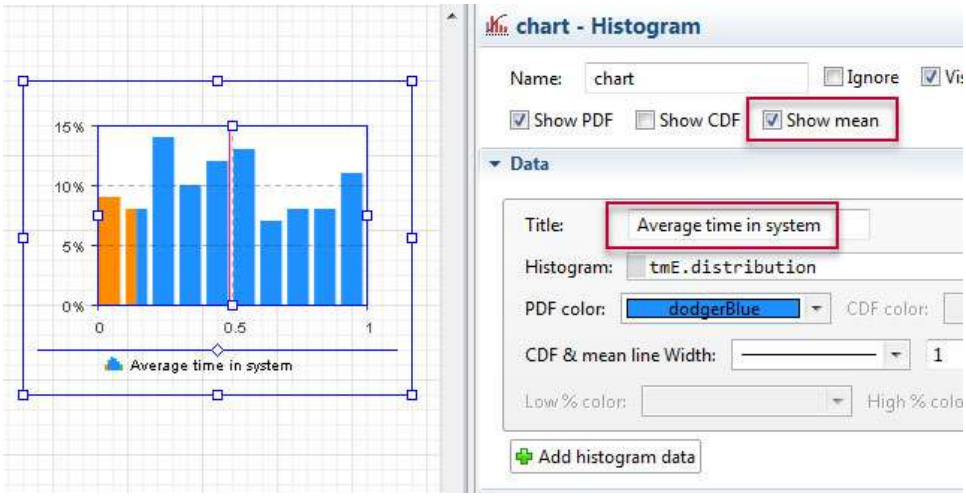


Figure 2-71: Adding a histogram to show the distribution of the entities' time in system

21. Repeat the same steps for the Average time in queue by right-clicking on **tmE_Q** and select **Create Chart** -> **Distribution**. Click the **chart** and change the title of the data to "Average time in queue". Then check the **Show mean** checkbox (Figure 2-72).

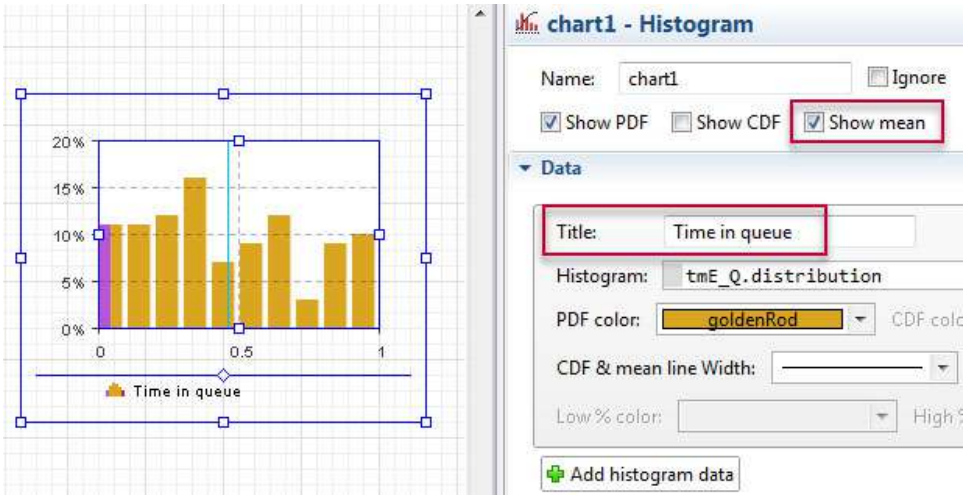


Figure 2-72: Adding a histogram to show the distribution of the entities' time in queue

22. Set the simulation stop time to 10000. This value is in seconds because the model time unit when you create the model was in seconds. (Figure 2-73).

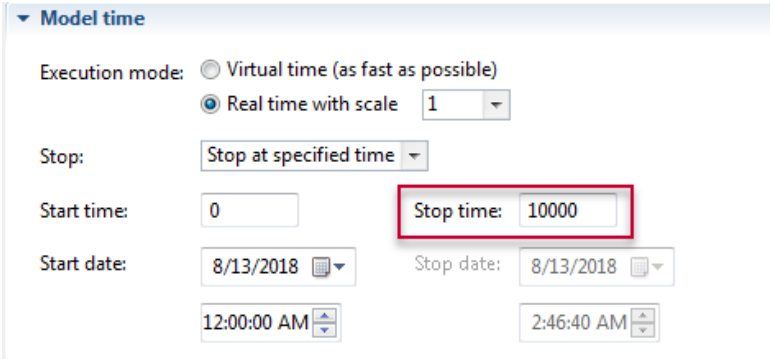


Figure 2-73: Setting the model to stop at 10000 time units (seconds)

23. Set the simulation random number generator to **Random seed** (Figure 2-74).

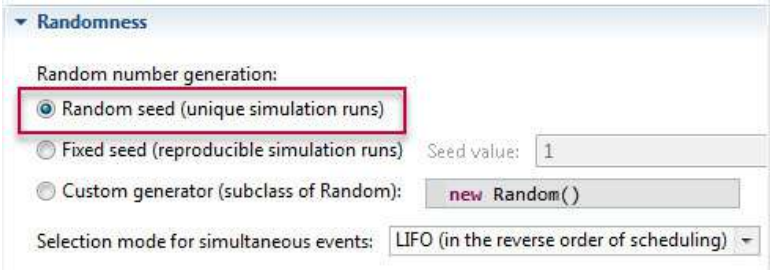


Figure 2-74: Set the model randomness setting to Random

24. In the **Projects** window, right-click your model's **Simulation** experiment and select **Run**. When the simulation window displays, click the **Play** button. You should run the simulation in virtual mode by speeding the model past x1000 or by clicking the '⏩' button.

If you're using AnyLogic PLE, the following error message appears: *The model has reached the maximum number of dynamically created agents (50000) for this version of AnyLogic* (Figure 2-75). Close the dialog box by clicking the **X** in its upper right corner.

The results shown in all the examples in this chapter are from runs we performed in the Professional license without any limitation on the number of agents (entities). This limitation will reduce the accuracy of performance metrics estimates, but 50000 samples will be enough for our example models.

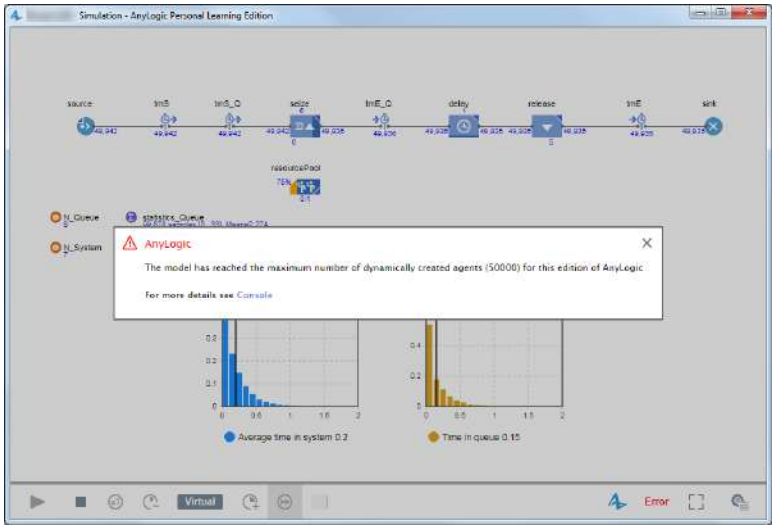


Figure 2-75: Error message regarding the 50000 limitation on the number entities in PLE version of AnyLogic

25. Click **statistics_Queue**, **statistics_System**, **resourcePool**, and both time-measure end blocks to open their inspection windows and see the results (Figure 2-76).

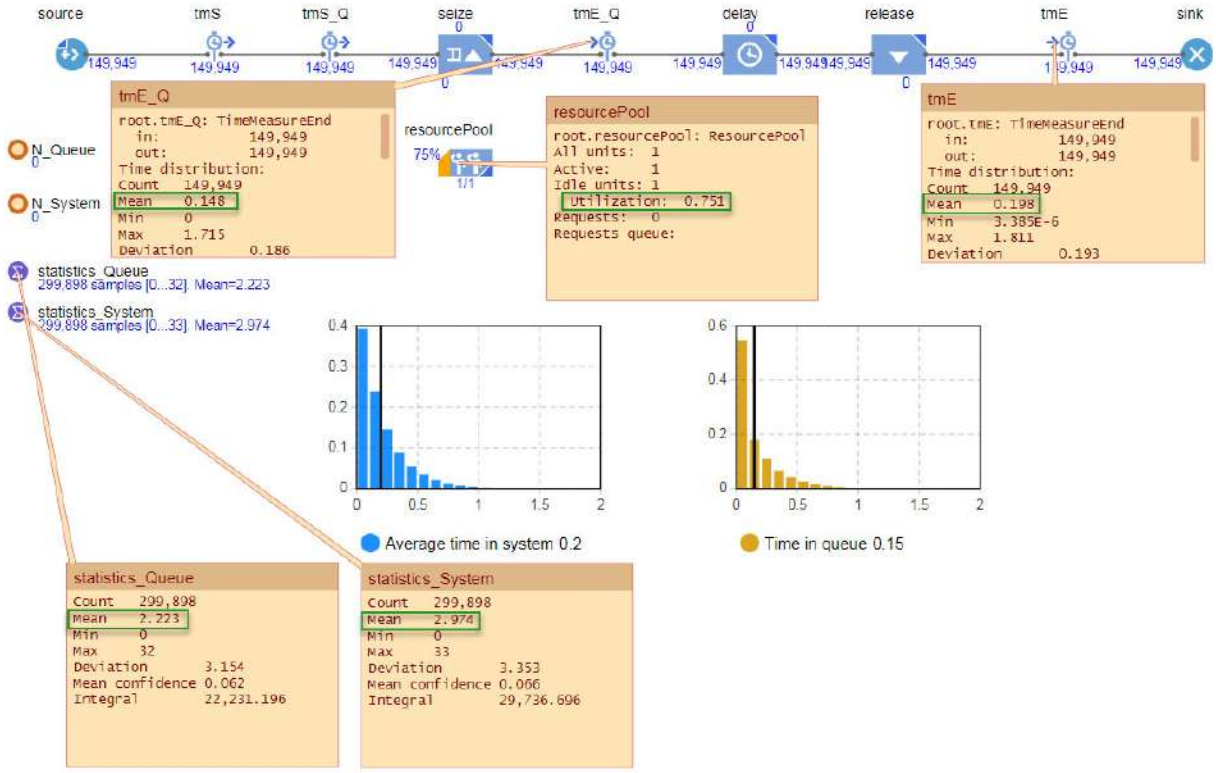


Figure 2-76: Set the model randomness setting to Random

Table 2-10: Comparison of exact and simulated values (M/M/1/∞/∞ example)

Description	From simulation	From calculations
Long-run average number of entities in system	$\hat{L} = 2.974$	$L = 3$
Long-run average number of entities in queue	$\hat{L}_q = 2.223$	$L_q = 2.25$
Long-run average time spent in system per entity	$\widehat{W} = 0.0.198$	$W = 0.2$
Long-run average time spent in queue per entity	$\widehat{W}_q = 0.148$	$W_q = 0.15$
Utilization of resource pool (1 server)	$\hat{\rho} = 0.751$	$\rho = 0.75$

As you can see in Table 2-10, our direct estimates from one long simulation run are reasonably close to the actual numbers. In the next section, we'll discuss how to indirectly estimate the performance metrics from direct estimates out of a simulation run. We'll also discuss the underlying reasons for the bias in our simulation estimations and introduce some remedies.

Our purpose in building simulation models for common queueing archetypes is not so much on the detailed analysis of the output. Instead, we want to learn the modeling techniques we need to build these queuing archetypes and then show how to gauge their performance. By doing this, we can get ideas for the more realistic, complex processes we'll see in later chapters.

Estimation of performance metrics from simulation results

Retrieving correct performance metrics from a simulation model is important. When we receive performance metrics from the simulation model, we're not getting closed form mathematical solutions but rather *estimates* of the metrics. We receive **direct estimates** directly from the simulation outputs. We calculate **indirect estimates** by plugging direct estimates into formulas that set up relationships between different metrics (for example, Little's Law).

In this section, we'll discuss how we can record and directly or indirectly estimate the performance metrics of terminating and non-terminating simulations.

Direct estimation of averages (initial transient state, terminating models, or unstable models)

If we run a simulation of a queuing system for some time T, where N entities have passed through the system, we'll have n samples to calculate the following values:

- T: length of simulation run-time
- $N(t)$: Number of entities in system at time t
- $N_q(t)$: Number of entities in queue at time t
- $\bar{L}$: time average number of entities in system
- $\bar{L}_q$: time average number of entities in queue
- $\bar{W}$: average time spent in system per entity
- $\bar{W}_q$: average time spent in queue per entity
- $\bar{W}_s$: average time spent in server per entity

$$\bar{W}_q = \frac{\sum_{i=0}^{i=n} W_{q_i}}{n} \qquad \bar{W}_s = \frac{\sum_{i=0}^{i=n} W_{s_i}}{n} \qquad \bar{W} = \frac{\sum_{i=0}^{i=n} W_i}{n}$$

$$\bar{L} = \frac{1}{T}\int_{i=0}^{T} N(t)\,dt \qquad \bar{L}_q = \frac{1}{T}\int_{i=0}^{T} N_q(t)\,dt$$

The metrics shown above with an overbar (the horizontal line drawn above a letter) show they're the averages of n samples recorded. We can calculate these averages for all types of systems, regardless of the simulation length or the simulation state (that is, transient or steady). However, this section on direct estimations of averages is mostly used for terminating simulations. As we'll see in the next section, calculating the long-run averages for directly estimating steady-state values is like the transient state but with some justification in the model run time and remedies for warm-up period bias.

Manually calculating $\bar{L}$ and $\bar{L}_q$ is complex; but as we've showed in **TECHNIQUE 3**, the AnyLogic **Statistics** object can record and calculate continuous statistics.

> *To summarize, you can use the formulas provided above to directly estimate the performance metrics of a terminating simulation (finite time). All you need to do is to run the simulation for a finite period, record some samples and calculate the averages. AnyLogic will automatically calculate these averages for you, either directly by the PML blocks or an added Statistics object.*

However, if you follow the instructions in the previous section for the M/M/1/∞/∞ queue, the direct estimates of W and W_q ($\bar{W}$ and $\bar{W}_q$) won't be correct for unstable systems or stable systems in the initial transient state (warm-up period). To show some of the intricacies of recording correct metrics, we need

to review some examples. Figure 2-77 below shows the result of the M/M/1/∞/∞ system in the previous section, but with two changes: the model stop time is changed from 10000 to 100 and the interarrival of the source is changed from `exponential(15,0)` to `exponential(30, 0)`. You can modify the previous example and test this scenario. The impact of these changes results in a shorter run time and unstable inputs (*arrival rate* > *service rate or* $\rho > 1$). Therefore, the results don't show steady state values.

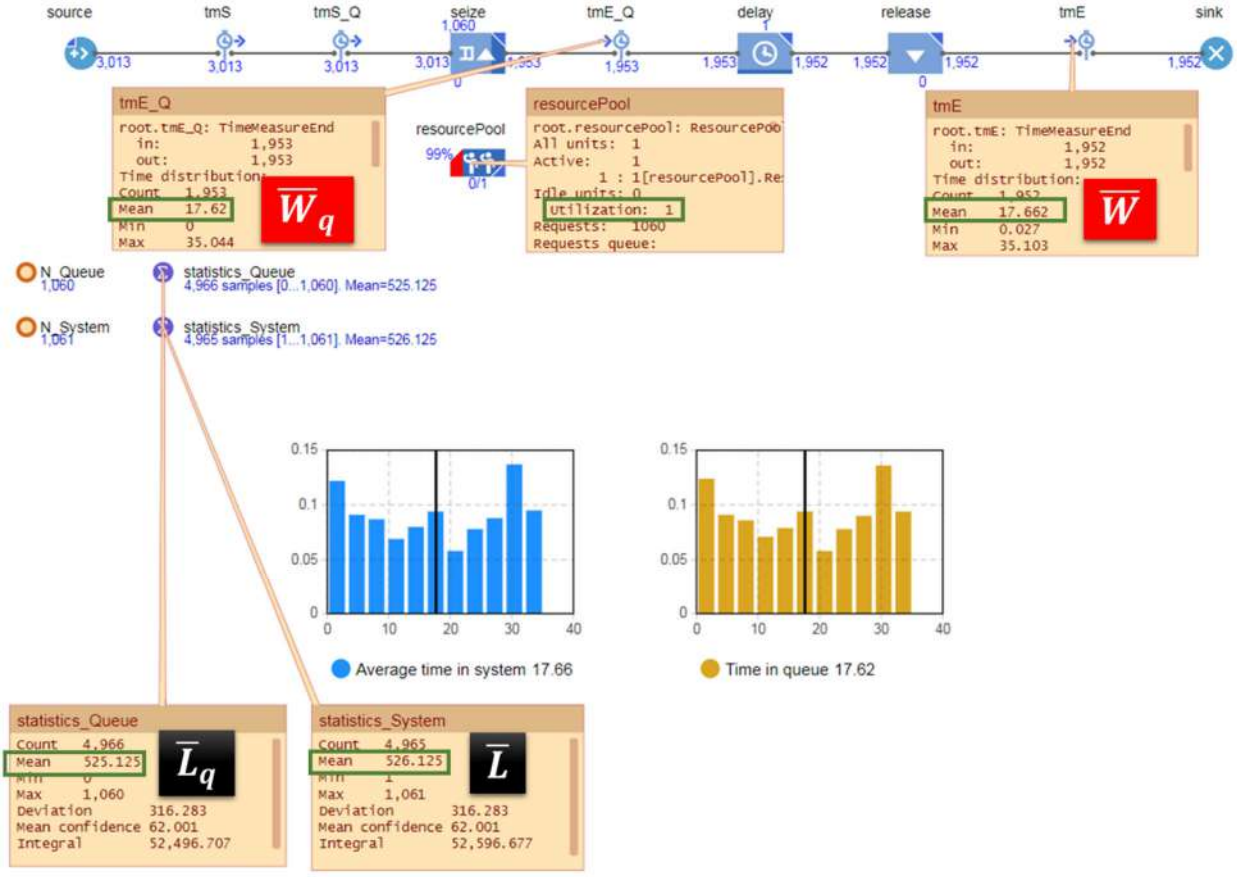


Figure 2-77: M/M/1/∞/∞, stop time = 100, λ=30, μ=20

Since this is a specific realization of the process (or one set of some specific actions taken), we should be able to check the sample path version of Little's Law (specifically version 2, since the queue has some entities in it at the model stop time). We do this to check the consistency of simulation estimated outputs. If our estimates are correct, plugging two out of three estimated metrics in Little's Law should result in the third:

$$\lambda(T) = \frac{N(T)}{T} = \frac{3013}{100} = 30.13$$

$$L(T) = \lambda(T) \times W(T) = 30.13 \times 17.662 = 531.979$$

As you can see, there's a discrepancy between the calculated output from Little's Law (532.1561) and the simulation's direct estimation of the third quantity ($\bar{L}$ = 526.125).

The source of this discrepancy occurs from using the value of $\overline{W}$ (colored red in Figure 2-77 above) when the simulation does not account for the people still in the queue (but our formula does). If you look at the model closely, you'll see the simulation concludes with 1060 entities in the seize (or more specifically, the inner queue of seize) and one entity is in the delay. **TimeMeasureEnd** blocks (**tmE_Q** and **tmE** blocks in this example) only record the waiting times when an entity reaches it. Since we have an unstable arrival rate and our service rate is lower than our arrival rate, most entities will have long waits in the queue. The 1061 work-in-progress entities (those in seize and delay) haven't passed through the **TimeMeasureEnd** blocks. This means our simulation does not account for their waiting times.

To do so, we need to change how we collect waiting times. To manually gather each entity's waiting time, we must understand several model building techniques we haven't covered. After you learn them, you can return to this example and change the model to calculate the exact terminating wait times. Conceptually, you need to make the following changes to record the exact waiting times:

- Add a timestamp to each entity, set to the time of arrival when an entity leaves the source
- When the entity exits the seize's inner queue, the difference between the simulation time at that moment and the recorded arrival time is that entity's waiting time in the queue. We can add these values to a dataset and get their average at the end of the simulation.
- Calculating the waiting time in the system follows the same process as the waiting time in queue, except the duration gets calculated when the entity reaches the sink.
- To be 100% accurate for $\overline{L}$ and $\overline{L}_q$, we need to add a sample of the number in queue and system to the continuous statistics object at the beginning and end of the simulation (please refer to **TECHNIQUE 3** for more information).

We won't show step-by-step instructions for this approach, but you'll find the modified model in this book's supplemental material. Figure 2-78 shows the result of the accurate recording of performance metrics from the modified model.

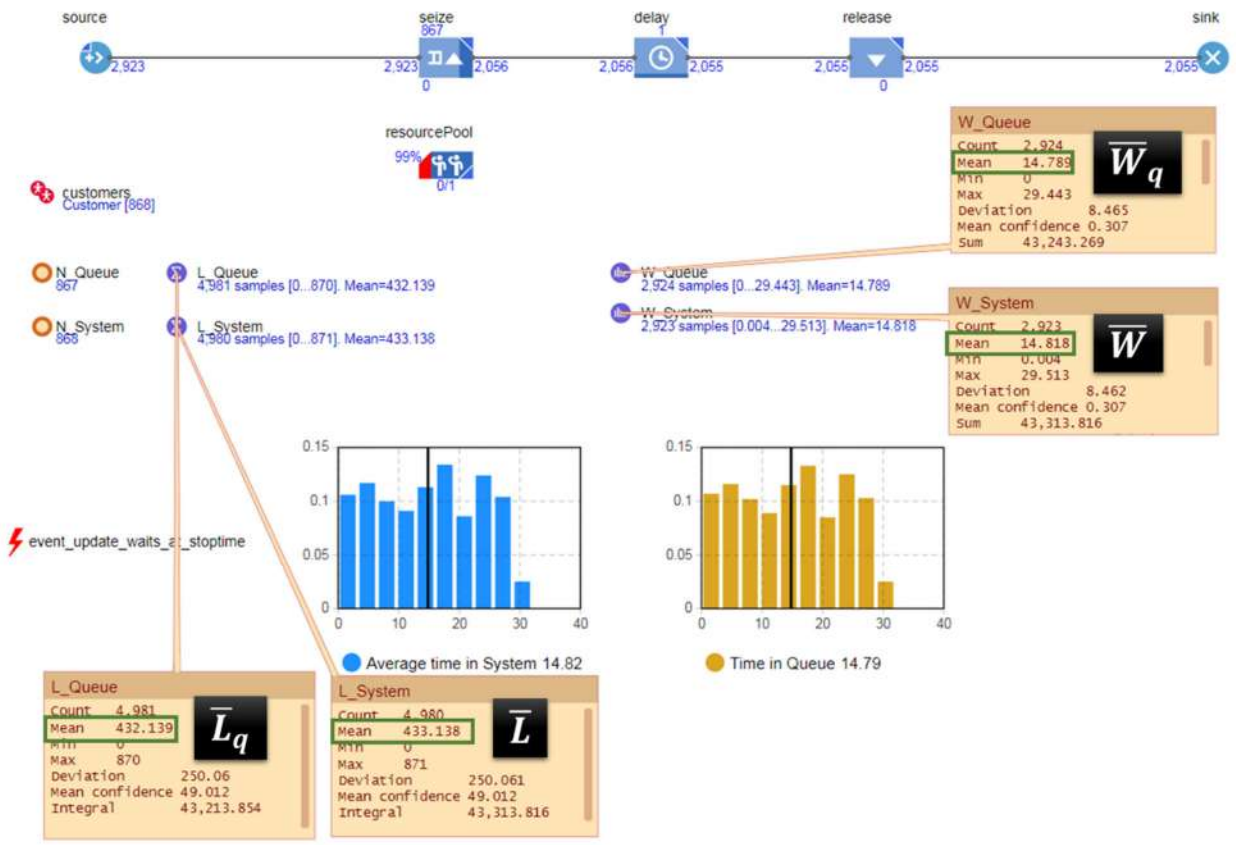


Figure 2-78: M/M/1/∞/∞, stop time = 100, λ=30,μ=20 with correct calculation of $\overline{W}$ and $\overline{W}_q$

We can plug the numbers from the simulation run shown in Figure 2-78 above and the check consistency of the results with Little's Law:

$$\lambda(T) = \frac{N(T)}{T} = \frac{2923}{100} = 29.23$$

$$L(T) = \lambda(T) \times W(T) = 29.23 \times 14.81827 = 433.138$$

Expectation based on Little's Law *does* match the direct estimate of the simulation: $433.138 == 433.138$

Direct estimation of long-run averages (steady state)

Theoretically, long-run averages are the result of a stable system over an infinite time. In practical terms, simulation models simply need to run long enough to reach to an acceptable estimate of the stable system's steady state. The formulas we use to calculate these estimates are the same as the previous averages, though you should gather your samples while the system is in steady state.

- $\hat{L}$: Estimate of long-run time average number of entities in system
- $\hat{L}_q$: Estimate of long-run time average number of entities in queue
- $\hat{W}$: Estimate of long-run average time spent in system per entity
- $\hat{W}_q$: Estimate of long-run average time spent in queue per entity
- $\hat{W}_s$: Estimate of long-run average time spent in server per entity

- $N(t)$: Number of entities in system at time t
- $N_q(t)$: Number of entities in queue at time t

$$\widehat{W}_q = \frac{\sum_{i=0}^{i=n} W_{q_i}}{n} \qquad \widehat{W}_s = \frac{\sum_{i=0}^{i=n} W_{s_i}}{n} \qquad \widehat{W} = \frac{\sum_{i=0}^{i=n} W_i}{n}$$

$$\hat{L} = \frac{1}{T}\int_{i=0}^{T} N(t)\,dt \qquad \hat{L}_q = \frac{1}{T}\int_{i=0}^{T} N_q(t)\,dt$$

$\hat{L}, \hat{L}_q, \widehat{W}, \widehat{W}_q, \widehat{W}_s$ are estimates of the true values of L, L_q, W, W_q, W_s respectively. The hat symbol placed on top of variables is to show they're estimates of **long-run averages**, instead of a bar symbol in the previous section which represented the simple average for a finite period.

Although we can run relatively long simulations to produce higher degrees of accuracy in the long-run averages, these estimates aren't perfect. These imperfections are due to the inherent randomness of the system (one run just gives us one realization of the system), insufficient simulation length, and bias from warmup period samples (if not removed from the samples). We'll discuss these issues in detail in an upcoming section.

Indirect estimation of long-run averages (steady state)

The two previous sections have shown us how simulation models of queuing systems can provide estimates of actual performance metrics: L, L_q, W, W_q, W_s (estimates are shown as $\hat{L}, \hat{L}_q, \widehat{W}, \widehat{W}_q, \widehat{W}_s$ respectively). However, we can only calculate these estimates if we record pertinent samples during the simulation execution. There might be cases we've recorded samples from one part of the system but not others and we're interested in **indirectly** estimating other metrics. Little's Law can help us indirectly estimate missing metrics, but we need to be aware of the implications.

Assuming:

- W: long-run average time spent in system per entity
- W_q: long-run average time spent in queue per entity
- W_s: long-run average time spent in server per entity

It is evident that:

$$W = W_q + W_s$$

Knowing $\widehat{W}, \widehat{W}_q, \widehat{W}_s$ are estimates of W, W_q, W_s respectively, we can write:

$$\widehat{W} = \widehat{W}_q + \widehat{W}_s$$

In the above formula, $\widehat{W}_s$ can be derived from:

$$\widehat{W}_s = \frac{\sum_{i=0}^{i=n_s} W_{s_i}}{n_s}$$

The formula for the actual long-run average time spent in the server per entity is like the estimate shown above; the only difference is that the number of entities approaches infinity.

$$W_s = \lim_{n_s \to \infty} \frac{\sum_{i=0}^{i=n_s} W_{s_i}}{n_s}$$

There's something exceptional about W_s compared to W_q and W - all instances of W_s (each W_{s_i}) are drawn from a random distribution, but W_q and W are the outcome of system performance. If we assume the random variable that represents the wait time is S, we can write:

$$W_s = E[S]$$

And therefore, we can substitute $\widehat{W}_s$ with the known value of W_s ($E[S]$):

$$\widetilde{\boldsymbol{W}} = \widehat{W}_q + E[S] \text{ (alternative estimate of W)}$$

Therefore, for a service station in a queuing system, we can estimate $\widehat{W}_q$ and add it to the expected service time to estimate W.

For example, in the M/M/1/∞/∞ model shown in the previous section:

- Results of mathematical solution for true values: $W = 0.2$, $W_q = 0.15$
- The expected value of service time (exponential distribution): $\mu = 20 \frac{entity}{sec}$ or $E[S] = 0.05\ sec$
- The output of simulation run estimating W and W_q: $\widehat{W} = 0.198$, $\widehat{W}_q = 0.148$

If we want to indirectly estimate $\widehat{W}$, we can use the alternative estimate: $\widetilde{W} = \widehat{W}_q + E[S] = 0.148 + 0.05 = 0.198$. This is almost identical to our direct estimate. Law (2015) argues that since the $E[S]$ is a fixed number, the alternative estimate ($\widetilde{W}$) is less variable than the direct estimate $\widehat{W}_s$.

We can also indirectly calculate other metrics. In a simulation model of a queueing system where we know λ and W_s:

- $A\ is\ the\ random\ interarrival\ time$
- $S\ is\ the\ random\ service\ time$
- $\lambda = \frac{1}{E[Interarrival\ time]} = \frac{1}{E[A]}$
- $W_s = E[S]$

Using Little's Law (Version 3) for both the system and queue, we can calculate all the other metrics after we retrieve the $\widehat{W}_q$ from the simulation model.

$$\tilde{\boldsymbol{L}}_{\boldsymbol{q}} = \lambda \times \widehat{W}_q$$

$$(\lambda = \frac{1}{E[A]}\ \&\ \widehat{W}_q\ from\ simulation\ output)$$

$$\tilde{\boldsymbol{L}} = \lambda \times \widetilde{W}$$

$$(\widetilde{\boldsymbol{W}} = \widehat{W}_q + E[S])$$

$\tilde{L}_q$ and $\tilde{L}$ that are alternative indirect estimates are generally more accurate than the direct estimates $\hat{L}_q$ and $\hat{L}$ from the simulation outputs. This is because $\lambda = \frac{1}{E[A]}$ and $W_s = E[S]$ are expected values and have less variance compared to their estimated counterparts.

To summarize our discussion about the indirect estimates, you can use the following formulas to indirectly estimate the metrics just by attaining $\widehat{W}_q$ (long-run direct estimates of W_q):

Known inputs to simulation: **A** (the random interarrival time) and **S** (the random service time)
Known direct estimate from the simulation is $\boldsymbol{\widehat{W}_q}$

1. Knowing the distribution of interarrival time and service time we can mathematically calculate $E[A]$ and $E[S]$.

2. We use $E[A]$ and $E[S]$ to calculate the λ and W_s:
   - $\lambda = \frac{1}{E[Interarrival\ time]} = \frac{1}{E[A]}$
   - $W_s = E[S]$

3. Using the $\widehat{W}_q$ (direct simulation estimate of W_q) we can calculate indirect estimate of W ($\widetilde{W}$) from the following formula:
   - $\boldsymbol{\widetilde{W}} = \widehat{W}_q + E[S]$

4. Now that we know the $\widehat{W}_q$, λ, $\widetilde{W}$, we can calculate indirect estimates of L and L_q ($\tilde{L}$ and $\tilde{L}_q$):
   - $\boldsymbol{\tilde{L}_q} = \lambda \times \widehat{W}_q$
   - $\boldsymbol{\tilde{L}} = \lambda \times \widetilde{W}$

In the following sections, we'll continue to review several important archetype queueing systems. We'll only show the long-run direct estimates of the simulation model for those models, but you can try to indirectly estimate the metrics using a single direct estimate of $\widehat{W}_q$ (the average time an entity spends in the queue).

Before we move to the other queueing archetypes, in the next section we'll discuss some of the nuanced assumptions we made in the simulation models of queueing systems of this chapter. These assumptions are mostly related to the way we gathered sample performance metric data (the direct estimates). It's important we're aware of these assumptions since some of them might affect our results.

Important caveats about the simulation output estimating the queueing system performance

In the previous example for a M/M/1/∞/∞, the simulation model perfectly represents the queuing system. However, the methods we used to record the sample and calculate the estimates aren't perfect. The following things need our attention:

1. In the previous section, we showed that at the stop time of the simulation, some entities may be waiting in the queue or delay (we call them undone entities). As stated, a correct waiting average should also account for the waiting times of these entities up to the model stop time.

   In a steady state scenario, we can assume the waiting time of undone entities should be like other entities that passed through the system in the steady state. This means the average waiting time we calculated for the entities that passed through the block(s) of interest should closely match the times for the entities that remain in the system.

   This assumption is consistent with our implementation of wait times recording in which we used **TimeMeasureStart** and **TimeMeasureEnd** blocks. This means we don't need to worry about the issue of waiting time for work-in-progress entities in the long-run estimates. In contrary to steady-state models, this issue could be a major problem in performance calculations of terminating models.

2. If the model starts in an empty and idle state, there will be a transient initial state where the waiting times differ from the steady state. Depending on the system configuration, this warm-up period might have a significant effect on the long-run estimates.

   In general, if the number of samples in the steady state is significantly larger than the number of samples from the warm-up period, the transient state's bias will be lower. In the examples shown in this chapter, we did not remove the samples from the warm-up period to help keep the models simple.

   Steady-state performance values are long-run averages. Although theoretically long-run means an infinite time, we can only run simulations with finite durations. This means the averages we calculate from the samples we gather during the finite run are estimates of the actual expected values.

   One important caveat: the model time does not tell us if it is long enough. If we run a simulation with the model time of 2 years but only 50 entities pass through the system, we may still be well in the transient state. We must make sure we gather enough samples in the steady state to ensure we have a relatively large number of samples available to estimate the averages.

3. We should be aware the performance metrics observed for both transient state and steady state (long-run averages) are just one realization of the system. If we run the same simulation, the random number generator that generates the underlying numerical values will generate a different stream of numbers. This means the average value of the performance metric across several runs is a more reliable estimate of these performance values.

   In the case of steady-state metric, a relatively big sample of metrics from one run could be considered equivalent to samples gathered from several runs - please refer to the ergodicity in

the math section for more information. In this chapter, all the metrics are outputs of one long run simulation (one realization of system outputs).

4. Lastly, a point on the way we calculate the time-average for the number of entities in the system or queue. According to **TECHNIQUE 3**, if we want the AnyLogic's **Statistics** object to perfectly calculate the time averages, we must add a sample of number in queue and system at the beginning and end of our simulation. In general, this small discrepancy should have a negligible effect on the "average time in queue" and "average time in system" values. That's why we ignored it for queueing models in this chapter.

M/M/c/∞/∞ (multi-server queues)

Characteristics of the M/M/c/∞/∞ queueing system (Figure 2-79):

- Interarrival time = exponential
- Service time = exponential
- Number of parallel servers = c
- System capacity = ∞
- Calling population = ∞

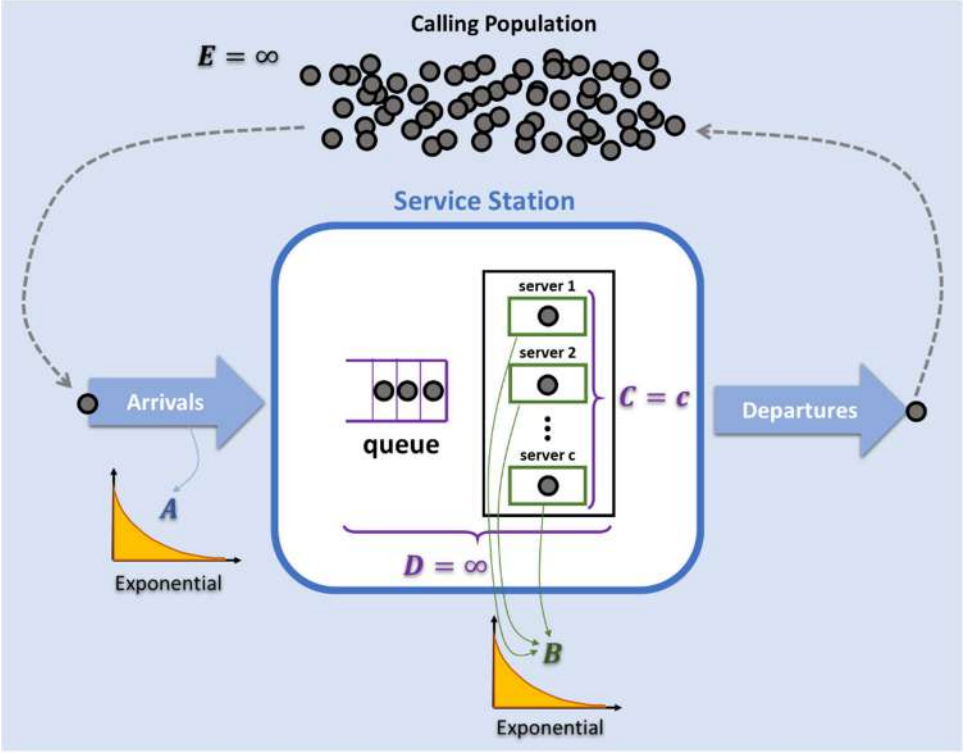


Figure 2-79: Illustration of M/M/c/∞/∞ (Multi-Server Queues)

Examples of an M/M/c/∞/∞ queueing system:

- A line of people at a call center who are waiting to speak to the next agent
- A line of customers who are waiting for an employee to take their lunch order
- A line of passengers at a hotel who are waiting for the next available receptionist

Formulas used for a generic M/M/c/∞/∞ queuing system:

$$r = \frac{\lambda}{\mu}$$

$$\rho = \frac{r}{c} = \frac{\lambda}{c\mu}$$

$$P_0 = \left[\frac{r^c}{c!\,(1-\rho)} + \sum_{n=0}^{c-1}\frac{r^n}{n!}\right]^{-1}$$

$$L_q = \left(\frac{r^c\rho}{c!\,(1-\rho)^2}\right)P_0$$

$$W_q = \frac{L_q}{\lambda} = \left(\frac{r^c}{c!\,(c\mu)(1-\rho)^2}\right)P_0$$

$$L = r + \left(\frac{r^c \rho}{c!(1-\rho)^2}\right) P_0 \equiv r + L_q \qquad W = \frac{1}{\mu} + \left(\frac{r^c}{c!\,(c\mu)(1-\rho)^2}\right) P_0 \equiv \frac{1}{\mu} + W_q$$

P_0 denotes the probability there are 0 entities in the system within a fraction of time during a long-run of the simulation.

We can easily modify the M/M/1/∞/∞ model to an M/M/c/∞/∞ by increasing the capacity of the resource pool from 1 to *c*.

M/M/c/∞/∞ queue example (mathematical solution)

Assumptions made for this example:

- $\lambda = \ the\ arrival\ rate = 50\ \left(\frac{entity}{second}\right)$
- $\mu = service\ rate\ of\ an\ individual\ server = 20\ \left(\frac{entity}{second}\right)$
- $c = \ number\ of\ parallel\ servers\ = 3\ \leftarrow (M/M/\boldsymbol{c})$
- $Queueing\ discipline\ is\ FIFO\ (First\ in\ First\ Out)$

Calculations of exact values:

$\lambda = 50$

$\mu = 20$

$c = 3$

$r = \frac{50}{20} = 2.5$

$\rho = \frac{2.5}{3} = 0.8333$

$$P_0 = \left[\frac{2.5^3}{3!\,(1-0.8333)} + \left(\frac{2.5^0}{0!}\right) + \left(\frac{2.5^1}{1!}\right) + \left(\frac{2.5^2}{2!}\right)\right]^{-1} = 0.0449$$

$$L_q = \left(\frac{2.5^3(0.8333)}{3!\,(1-0.8333)^2}\right)(0.0449) = \ 3.5112$$

$$L = 2.5 + \left(\frac{2.5^3(0.8333)}{3!\,(1-0.8333)^2}\right)(0.0449) = 6.0112$$

$$W = \frac{1}{20} + \left(\frac{2.5^3}{3!\,60(1-0.8333)^2}\right)(0.0449) = 0.1202$$

$$W_q = \left(\frac{2.5^3}{3!\,60(1-0.8333)^2}\right)(0.0449) = 0.0702$$

M/M/c/∞/∞ queue example (simulation model)

Prerequisites:

Model Building Blocks (Level 1): **Source**, **Service**, **Sink**, **TimeMeasureStart**, **TimeMeasureEnd**, and **ResourcePool** blocks.

Math: Random variable, Random variate, [Poisson process], Exponential Distribution

How to build the simulation model, using the same assumptions as in the mathematical solution:

1. Open the M/M/1/∞/∞ model in the previous example.

   Before we modify the model, you may want to keep your original version intact. To do this, point to the **File** menu and then click **Save As** to save a copy with a different name.

2. Click **source** and change the **arrival rate** to 50 per second. If you are using interarrival time, then the distribution should be `exponential(50, 0)` seconds.

3. Click **resourcePool** and change its capacity to 3.

4. Run the **Simulation** experiment and wait until the 10000 second simulation time is finished (you can check the model time in the **Developer** panel). If you use the PLE version, the model will stop after 50000 entities arrive, but your results should be comparable.

5. Click **statistics_Queue**, **statistics_System**, **tmE_Q**, **tmE**, and **resourcePool** to open their inspection windows. Figure 2-80 shows us one iteration.

Figure 2-80: Simulation output for an M/M/c/∞/∞ system, λ=50, μ=20, c=3

Again, remember that the direct estimates from the simulation are rough estimates from one realization of the system. We can improve them with methodological corrections and systematic output analysis.

Table 2-11 shows a comparison of exact and simulated values:

Table 2-11: Comparison of exact and simulated values (M/M/c/∞/∞ example)

Description	From simulation	From calculations
Long-run average number of entities in system	$\hat{L} = 5.962$	$L = 6.0112$
Long-run average number of entities in queue	$\hat{L}_q = 3.46$	$L_q = 3.5112$
Long-run average time spent in system per entity	$\hat{W} = 0.119$	$W = 0.1202$
Long-run average time spent in queue per entity	$\hat{W}_q = 0.069$	$W_q = 0.0702$
Utilization of resource pool (average of 3 servers)	$\hat{\rho} = 0.834$	$\rho = 0.83333$

M/M/c/k/∞ (queues with truncation)

Characteristics of the M/M/c/k/∞ queueing system (Figure 2-81):

- Interarrival time = M (exponential)
- Service time = M (exponential)
- Number of parallel servers = c
- System capacity = k
- Calling population = ∞

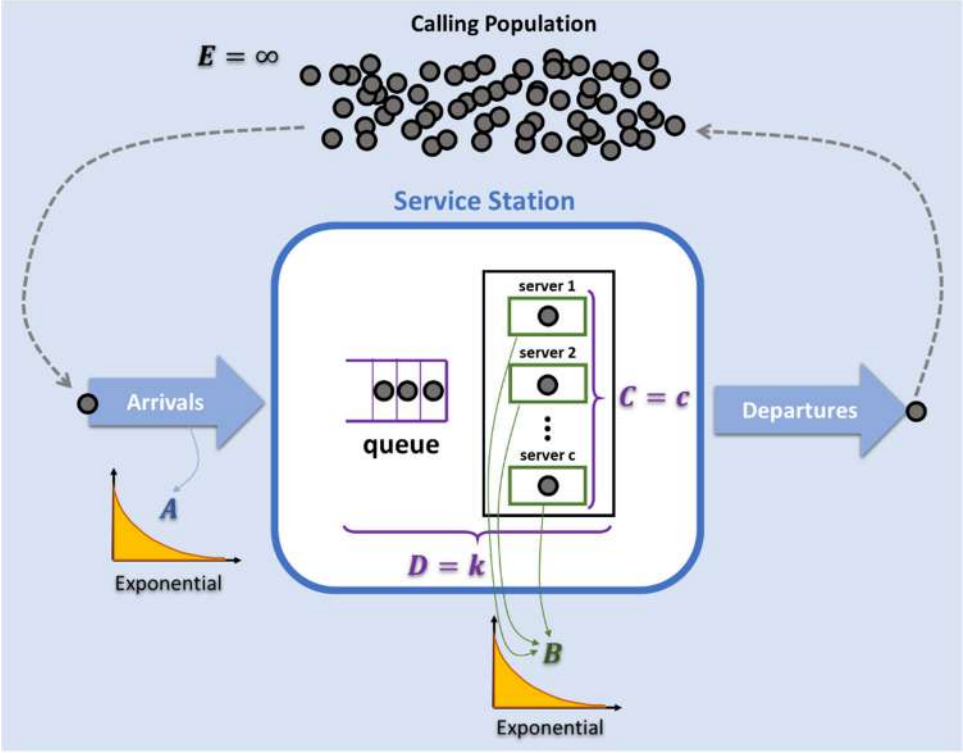


Figure 2-81: Illustration of M/M/c/k/∞ (Queues with Truncation)

Examples of an M/M/c/k/∞ queueing system:

- A line of customers who want to order fast food and are waiting for the next available cashier. If the line is at or past a certain length, they will go to the restaurant next door.
- A queue of customers who are waiting to speak with the next available support representative. If the queue has more than 100 people, the customer will encounter a busy signal and won't be placed on hold.

Formulas used for an M/M/c/k/∞ queuing system:

$$r = \frac{\lambda}{\mu} \qquad \rho = \frac{r}{c} \equiv \frac{\lambda}{c\mu}$$

$$P_0 = \begin{cases} \left[\frac{r^c}{c!}\left(\frac{1-\rho^{k-c+1}}{1-\rho}\right) + \sum_{n=0}^{c-1}\frac{r^n}{n!}\right]^{-1} & (\rho \neq 1), \\ \left[\frac{r^c}{c!}(k-c+1) + \sum_{n=0}^{c-1}\frac{r^n}{n!}\right]^{-1} & (\rho = 1). \end{cases} \qquad P_n = \begin{cases} \frac{\lambda^n}{n!\,\mu^n}P_0 & (0 \leq n < c), \\ \frac{\lambda^n}{c^{n-c}c!\,\mu^n}P_0 & (c \leq n \leq k). \end{cases}$$

$$L_q = \frac{P_0 r^c \rho}{c!\,(1-\rho)^2}[1-\rho^{k-c+1}-(1-\rho)(k-c+1)\rho^{k-c}] \qquad W_q = W - \frac{1}{\mu} \equiv \frac{L_q}{\lambda_{eff}} \equiv \frac{L_q}{\lambda(1-P_k)}$$

$$L = L_q + r(1-P_k) \equiv L_q + \frac{\lambda_{eff}}{\mu} \equiv L_q + \frac{\lambda(1-P_k)}{\mu} \qquad W = \frac{L}{\lambda_{eff}} \equiv \frac{L}{\lambda(1-P_k)}$$

$$\lambda_{eff} = \lambda(1-P_k) \qquad \rho_{eff} = \frac{\lambda_{eff}}{c\mu}$$

P_n denote the probability (long-run fraction of time) there are n entities in the system.
λ_{eff} denotes the effective arrival rate based on the entities that entered the system, the effective arrival rate might be different from λ due to the truncation effect

M/M/c/k/∞ queue example (mathematical solution)

Assumptions for this example:

- $\lambda = \ the\ arrival\ rate = 55\ \left(\frac{entity}{second}\right)$
- $\mu = service\ rate\ of\ an\ individual\ server = 20\ \left(\frac{entity}{second}\right)$
- $c = \ number\ of\ parallel\ servers\ = 3\ \leftarrow\ (M/M/\boldsymbol{c}/k/\infty)$
- $k = \ system\ capacity\ = 9\ \leftarrow\ (M/M/c/\boldsymbol{k}/\infty)$
- $Queueing\ discipline\ is\ FIFO\ (First\ in\ First\ Out)$

Calculations of exact values:

$$r = \frac{55}{20} = 2.75$$

$$\rho = \frac{2.75}{3} = 0.9167$$

$$(\rho \neq 1) \Rightarrow P_0 = \left[\frac{2.75^3}{3!}\left(\frac{1-0.9167^{9-3+1}}{1-0.9167}\right) + \left(\frac{2.5^0}{0!}\right) + \left(\frac{2.5^1}{1!}\right) + \left(\frac{2.5^2}{2!}\right)\right]^{-1} = 0.0377$$

$$P_{n=9} = P_k\ \&\ (3 \leq 9 \leq 9) \Rightarrow P_9 = \frac{(55)^9}{(3)^6(6)(20)^9}(0.0377) = 0.0776$$

$$L_q = \frac{(0.0377)(2.5)^3(0.9167)}{(6)(1-0.9167)^2}[1-(0.9167)^{9-3+1}-(1-0.9167)(9-3+1)(0.9167^{9-3})]$$
$$= 1.9000$$

$$L = 1.9 + 2.75(1-0.0776) = 4.4366$$

$$\lambda_{eff} = (55)(1-0.0776) = 50.7326$$

$$W = \frac{4.4366}{50.7326} = 0.0875$$

$$W_q = \frac{1.9000}{50.7326} = 0.0375$$

$$\rho_{eff} = \frac{50.7326}{(3)(20)} = 0.8455$$

M/M/c/k/∞ queue example (simulation model)

Prerequisites:

Model Building Blocks (Level 1): Source, Service, Sink, TimeMeasureStart, TimeMeasureEnd, SelectOutput, RestrictedAreaStart, RestrictedAreaEnd, and ResourcePool blocks.

Math: Random variable, Random variate, [Poisson process], Exponential Distribution

With the same assumption as the mathematical solution, the following steps show how to build the simulation model:

We'll now modify the base M/M/1/∞/∞ model to an M/M/c/k/∞ by increasing the resource pool's capacity from 1 to *c* and putting a cap on the entities that can be in the system at an time.

1. Open the M/M/1/∞/∞ model built in the previous example.

   Before we modify the model, you may want to keep your original version intact. To do this, point to the **File** menu and then click **Save As** to save a copy with a different name.

2. Click the **source** and change the arrival rate to 55 per second. If you are using interarrival time, the distribution should be `exponential(55, 0)` seconds.

3. Click **resourcePool** and change its capacity to 3.

4. Add a **RestrictedAreaStart** from PML and put it after the source, rename it to **raS**, and set **Capacity (max allowed)** to 9. The value 9 corresponds to the "k", the truncation threshold for the maximum number of entities in the system.

5. Add a **RestrictedAreaEnd** before the sink and rename it to **raE** (Figure 2-82). To couple **raS** to **raE**, you need to select **raS** from the drop-down menu in front of the **RestrictedAreaStart** object field. Alternatively, you can choose **raS** from the graphical editor by clicking on the graphical element chooser to the right of drop-down menu.

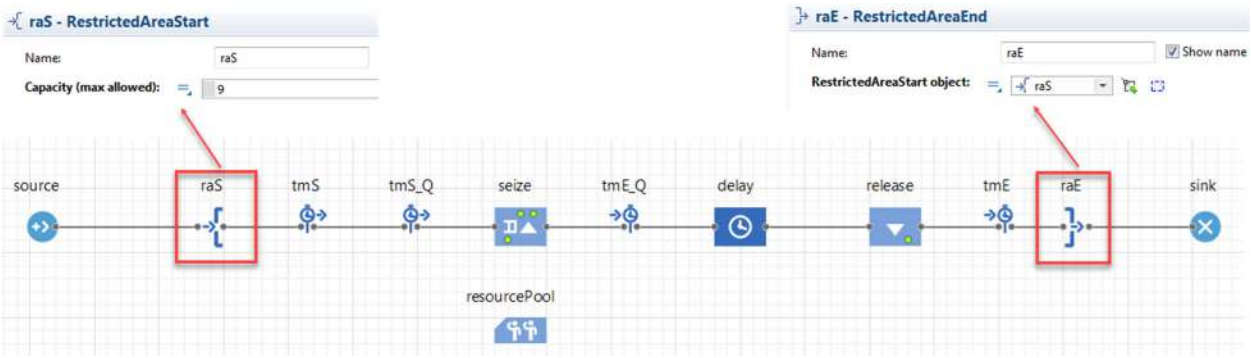


Figure 2-82: Adding RestrictedAreaStart and RestrictedAreaEnd blocks

6. Add a **SelectOutput** block (you can keep the automatically generated name) from the PML and place it after the source.
7. Add a **Sink** (you can also keep the automatically generated name) and connect it to the **outF** port of **selectOutput** (Figure 2-83).

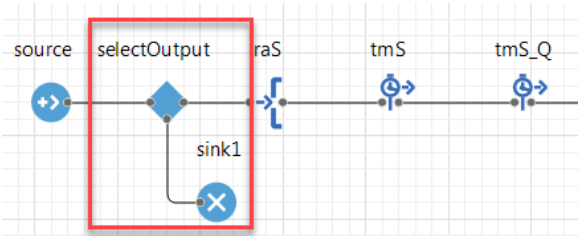


Figure 2-83: Adding a SelectOutput and an extra sink to the out false (outF) branch

8. Click **selectOutput**, select the **If condition is true** choice, then add to its condition field: `!raS.isBlocked()` (Figure 2-84).

   `isBlocked()` is a method belonging to the **RestrictedAreaStart** block that returns true if the max number of entities are inside the system (between the restricted area start and end). The exclamation mark (!) in Java (and some other programming languages) is the "logical not operator" and inverts the Boolean (true/false) value. This condition will return True if the restricted area is *not* blocked (and thus is not full). If restricted area *is* blocked, the entities will flow to the *outF* block (the bottom port of the **SelectOutput** block).

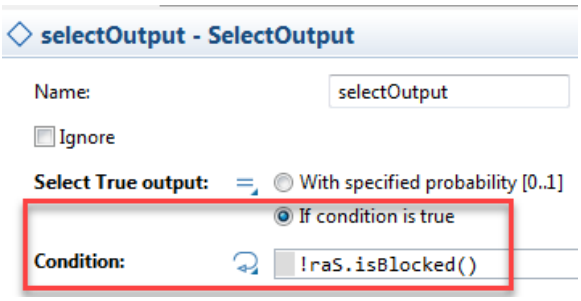


Figure 2-84: Setting condition for redirecting entities if the truncation threshold has been reached

9. We also must move the code that updates the number of entities in the system from the source to the select output block. This is because only the entities that exit via *outT* (true port of **selectOutput**) are the ones that enter the system, while the rest that exits via *outF* never enter the system. To perform this change (Figure 2-85):

    a. Click **source** and cut the code snippet in the **On exit** field.

    b. Click **selectOutput** and paste the code **On exit (true)** field.

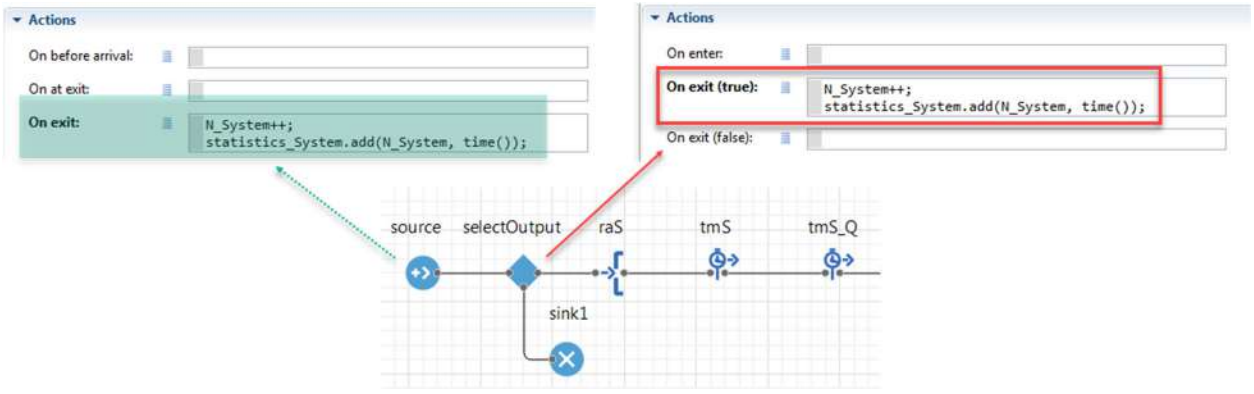


Figure 2-85: Moving the code that updates the number of entities in the system from source to selectOutput

10. Run the **Simulation** experiment and wait until the 10000 second simulation time is finished (you can check the model time in the **Developer** panel). If you use the PLE version, the model will stop after 50000 entities arrive, but your results should be comparable.

11. Click **statistics_Queue**, **statistics_System**, **tmE_Q**, **tmE**, and **resourcePool** to open their inspection windows. We can see one iteration's results in Figure 2-86.

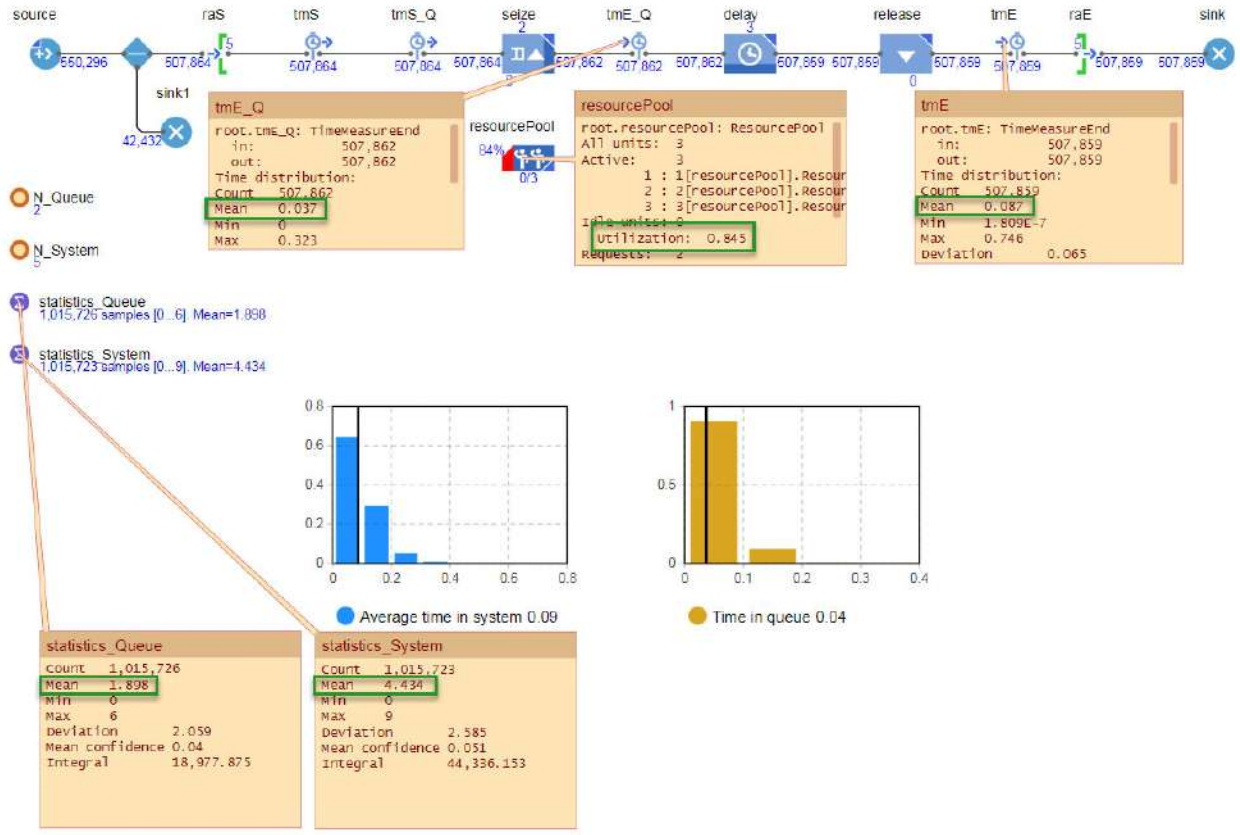


Figure 2-86: Simulation output for a M/M/c/k/∞ system, λ=55, μ=20, c=3, k=9

Table 2-12 shows a comparison of exact and simulated values:

Table 2-12: Comparison of exact and simulated values (M/M/c/k/∞ example)

Description	From simulation	From calculations
Long-run average number of entities in system	$\hat{L} = 4.434$	$L = 4.4366$
Long-run average number of entities in queue	$\hat{L}_q = 1.898$	$L_q = 1.9000$
Long-run average time spent in system per entity	$\hat{W} = 0.087$	$W = 0.0875$
Long-run average time spent in queue per entity	$\hat{W}_q = 0.037$	$W_q = 0.0375$
Utilization of resource pool (average of 3 servers)	$\hat{\rho}_{eff} = 0.845$	$\rho_{eff} = 0.8455$

M/M/c/c/∞ (Erlang's loss formula)

Characteristics of the M/M/c/c/∞ queueing system (Figure 2-87):

- Interarrival time = M (exponential)
- Service time = M (exponential)
- Number of parallel servers = c
- System capacity = c
- Calling population = ∞

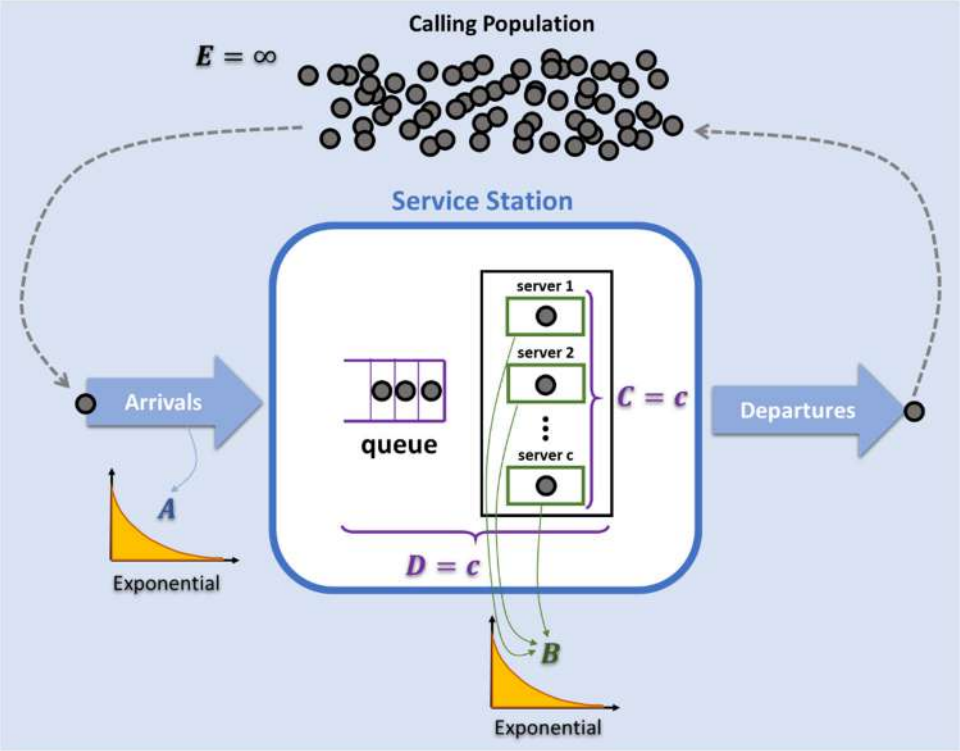


Figure 2-87: Illustration of M/M/c/c/∞ queueing system

Examples of flow systems we can model as M/M/c/c/∞ queueing system:

- A call center in which incoming calls follow a Poisson stream, where calls that find all lines busy will be turned away - This is the original problem that motivated Erlang to develop the mathematical solution for these types of systems.
- A car wash with five power wash machines can work on 5 cars at the same time. However, the car wash does not have room for cars to wait for a washing spot.
- A parking garage with room for up to 120 cars has an entrance gate that prevents cars from entering unless there's at least one available spot.

Formulas for a generic M/M/c/c/∞ queuing system:

$$r = \frac{\lambda}{\mu} \qquad \rho = \frac{r}{c}$$

(Iterative version) (Recursive version)

$$P_c \equiv B(c,r) = \frac{\frac{r^c}{c!}}{\sum_{i=0}^{c} \frac{r^i}{i!}} \qquad P_c \equiv B(c,r) = \begin{cases} 1, & c = 0 \\ \frac{rB(c-1,r)}{c + rB(c-1,r)}, & c > 0 \end{cases}$$

$$L_q = 0\ (by\ definition) \qquad W_q = 0\ (by\ definition)$$

$$L = \frac{\lambda_{eff}}{\mu} \equiv \frac{\lambda(1-P_c)}{\mu} \qquad W = \frac{L}{\lambda_{eff}} \equiv \frac{L}{\lambda(1-P_c)}$$

$$\lambda_{eff} = \lambda(1-P_c) \equiv \lambda\big(1 - B(c,r)\big) \qquad \rho_{eff} = \frac{\lambda_{eff}}{c\mu} \equiv \frac{\lambda(1-P_c)}{c\mu}$$

$P_c\ or\ B(c,r)$ is called *Erlang's loss formula* or the *Erlang-B formula* and denotes the probability of a full system at any time while that system is in a steady state.

Both the iterative and recursive ways to solve P_c are shown above – both provide the same answer.

An interesting fact about the abovementioned formulas is they work well for the general service-time distribution case. This means we can apply them to any truncated system where the truncation cap and number of servers are equal, regardless of the distribution of the server's serving times (M/**G**/c/c/∞).

Another interesting point is the formulas of $P_c\ or\ B(c,r)$ are for systems with c system capacity (M/M/c/c/∞), but they also apply to performance metric calculation of systems with infinite queues (M/M/c/∞/∞). The formulas below show how you can use $B(c,r)$ to calculate $C(c,r)$ which is the probability of delay in a multi-server queue without truncation (M/M/c/∞/∞). In M/M/c/∞/∞ systems, we could use $C(c,r)$ as an alternative formula to calculate the L_q and imply the rest of the performance metrics.

Performance formulas for M/M/c/∞/∞ systems (without truncation):

$$C(c,r) \stackrel{\text{def}}{=} Probability\ of\ delay$$

$$C(c,r) = \frac{cB(c,r)}{c - r + rB(c,r)}$$

$$L_q = C(c,r)\frac{\rho}{1-\rho} = C(c,r)\frac{r}{c-r}$$

M/M/c/c/∞ queue example (mathematical solution)

Assumptions for this example:

- $\lambda = \ the\ arrival\ rate = 55\ \left(\frac{entity}{second}\right)$
- $\mu = service\ rate\ of\ an\ individual\ server = 20\ \left(\frac{entity}{second}\right)$
- $c = \ number\ of\ parallel\ servers\ = 3\ \leftarrow\ (M/M/\boldsymbol{c}/c/\infty)$
- $k = \ system\ capacity\ = 3\ \leftarrow\ (M/M/c/\boldsymbol{c}/\infty)$
- $Queueing\ discipline\ is\ FIFO\ (First\ in\ First\ Out)$

Calculations of exact values:

$r = \frac{55}{20} = 2.75$

$\rho = \frac{2.75}{3} = 0.9167$

$$Iterative: P_3 = \frac{\frac{2.75^3}{6}}{1 + 2.75 + \frac{2.75^2}{2} + \frac{2.75^3}{6}} = 0.3152$$

$$Recursive: \begin{cases} P_0 = B(0, 2.75) = 1 \\ P_1 = B(1, 2.75) = \dfrac{2.75 \times B(0, 2.75)}{\left(1 + 2.75 \times B(0, 2.75)\right)} = \dfrac{2.75 \times 1}{(1 + 2.75 \times 1)} = 0.7333 \\ P_2 = B(2, 2.75) = \dfrac{2.75 \times B(1, 2.75)}{\left(1 + 2.75 \times B(1, 2.75)\right)} = \dfrac{2.75 \times 0.7333}{(2 + 2.75 \times 0.7333)} = 0.5021 \\ P_3 = B(3, 2.75) = \dfrac{2.75 \times B(2, 2.75)}{\left(1 + 2.75 \times B(2, 2.75)\right)} = \dfrac{2.75 \times 0.5021}{(3 + 2.75 \times 0.5021)} = 0.3152 \end{cases}$$

$B(3,2.75) = \ P_3 = 0.3152\ (alternative\ formula\ for\ P_3)$

$L_q = 0\ (By\ design)$

$W_q = 0\ (By\ design)$

$\lambda_{eff} = \lambda\big(1 - B(c, r)\big) = 55(1 - 0.3152) = 37.6652$

$L = \frac{37.6652}{20} = 1.8833$

$W = 0.0500$

$\rho_{eff} = \frac{0.0500}{(3)(20)} = 0.6278$

M/M/c/c/∞ queue example (simulation model)

Prerequisites*:*

Model Building Blocks (Level 1)*:* **Source**, **Service**, **Sink**, **TimeMeasureStart**, **TimeMeasureEnd**, **SelectOutput**, **RestrictedAreaStart**, **RestrictedAreaEnd**, and **ResourcePool** blocks.

Math*:* Random variable, random variate, [Poisson process], Exponential Distribution

We'll modify the previous generic M/M/c/k/∞ model to an M/M/c/c/∞ by setting the maximum number of entities that can be in the system at any time to equal the number of servers (resource pool's capacity):

1. Open the M/M/c/k/∞ model you built in the previous example.

   Before we modify the model, you may want to keep your original version intact. To do this, point to the **File** menu and then click **Save As** to save a copy with a different name.

2. Click **raS** and set the value for **Capacity (max allowed)** to 3. This ensures the truncation threshold for the maximum number of entities in system and number of servers will be equal.

3. Run the **Simulation** experiment and wait until the 10000 second simulation time is finished (you can check the model time in the **Developer** panel). If you use the PLE version, the model will stop after 50000 entities arrive, but your results should be comparable.

4. Click **statistics_Queue**, **statistics_System**, **tmE_Q**, and **tmE** to open their inspection windows (Figure 2-88).

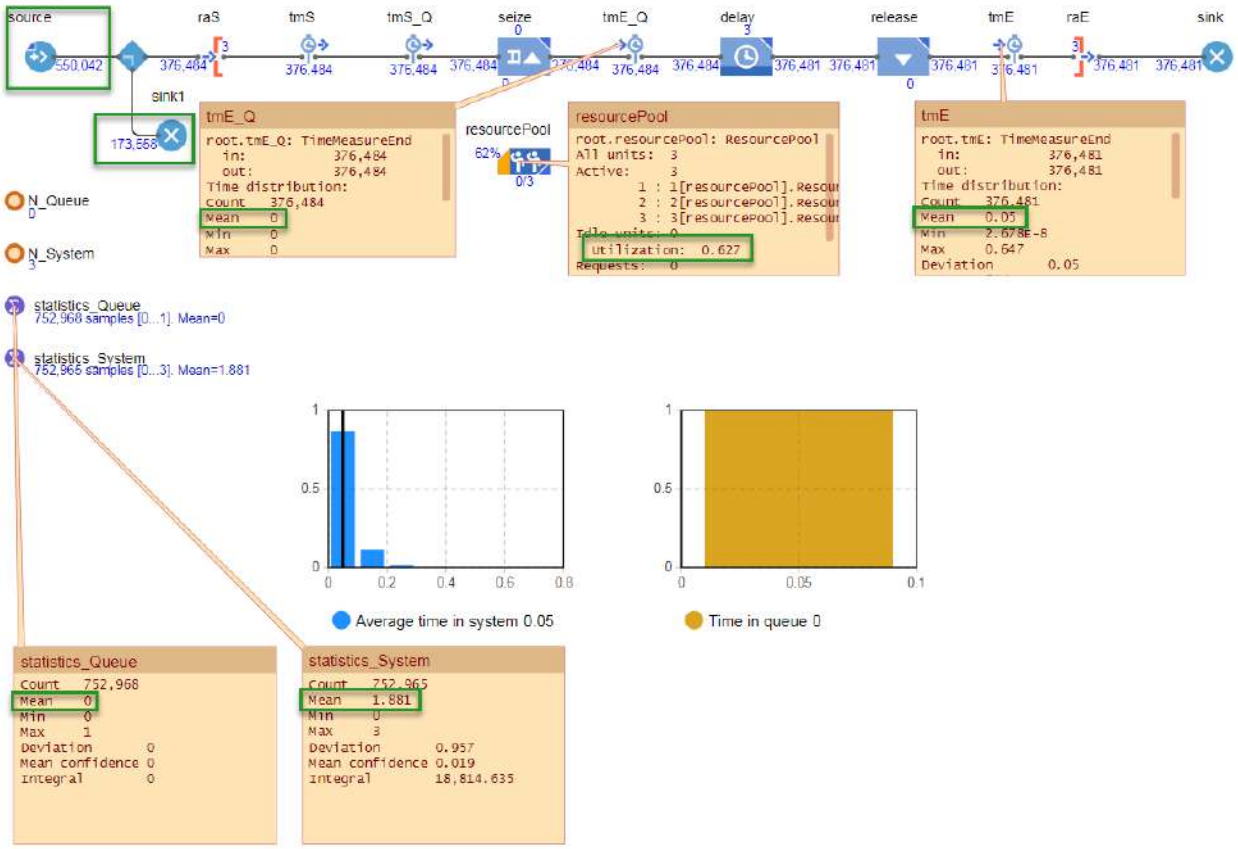


Figure 2-88: Simulation output for a M/M/c/c/∞ system, λ=55, μ=20, c=3

Table 2-13 shows a comparison of exact and simulated values:

Table 2-13: Comparison of exact and simulated values (M/M/c/c/∞ example)

Description	From simulation	From calculations
Long-run average number of entities in system	$\hat{L} = 1.881$	$L = 1.8833$
Long-run average number of entities in queue	$\hat{L}_q = 0$	$L_q = 0$
Long-run average time spent in system per entity	$\widehat{W} = 0.05$	$W = 0.05$
Long-run average time spent in queue per entity	$\widehat{W}_q = 0$	$W_q = 0$
Utilization of resource pool (average of 3 servers)	$\hat{\rho}_{eff} = 0.627$	$\rho_{eff} = 0.6278$
Fraction that could not enter the system	$\hat{B} = 0.3155$	$B = 0.3152$

In an M/M/c/∞/∞ system (without truncation) we can use the $C(c,r)$ formula to calculate the probability an entity must wait in a queue (in contrast to immediately being served). We can also get a direct estimate of $B(c,r)$ from a simulation to then calculate $C(c,r)$.

All you need to do is to first add truncation to your M/M/c/∞/∞ system and change it to M/M/c/**c**/∞. Then you would divide the number of entities that left the system immediately (through sink1) by the total number of entities that tried to enter the system to estimate $\hat{B}(c,r)$. You could then calculate $\hat{C}(c,r)$ from its formula.

For example, if we had a M/M/c/∞/∞ system with $\lambda = 55$, $\mu = 20$, $c = 3$, then the probability of entities being delayed before entering the servers is:

- $\hat{B}(3,2.75) = 0.3155$
- $\hat{C}(c,r) = \frac{c\hat{B}(c,r)}{c-r+r\hat{B}(c,r)} = \frac{3\times0.3155}{3-2.75+2.75\times0.3155} = 0.8469$

The following are the calculation of real values of $C(c,r)$ estimated using the simulation output above:

$$C(3,2.75) = \frac{3 \times 0.3152}{3 - 2.75 + 2.75 \times 0.3152} = 0.8467$$

We can also use the $C(c,r)$ to calculate the L_Q for an M/M/c/∞/∞ system ($\lambda = 55$ and $\mu = 20$, $c = 3$):

$$L_q = C(c,r)\frac{\rho}{1-\rho} = 0.8467 \times \frac{0.9167}{1-0.9167} = 9.3136$$

M/M/∞/∞/∞ (queues with unlimited service)

Characteristics of the M/M/∞/∞/∞ queueing system (Figure 2-89):

- Interarrival time = exponential
- Service time = exponential
- Number of parallel servers = ∞ (no resource constraint)
- System capacity = ∞
- Calling population = ∞

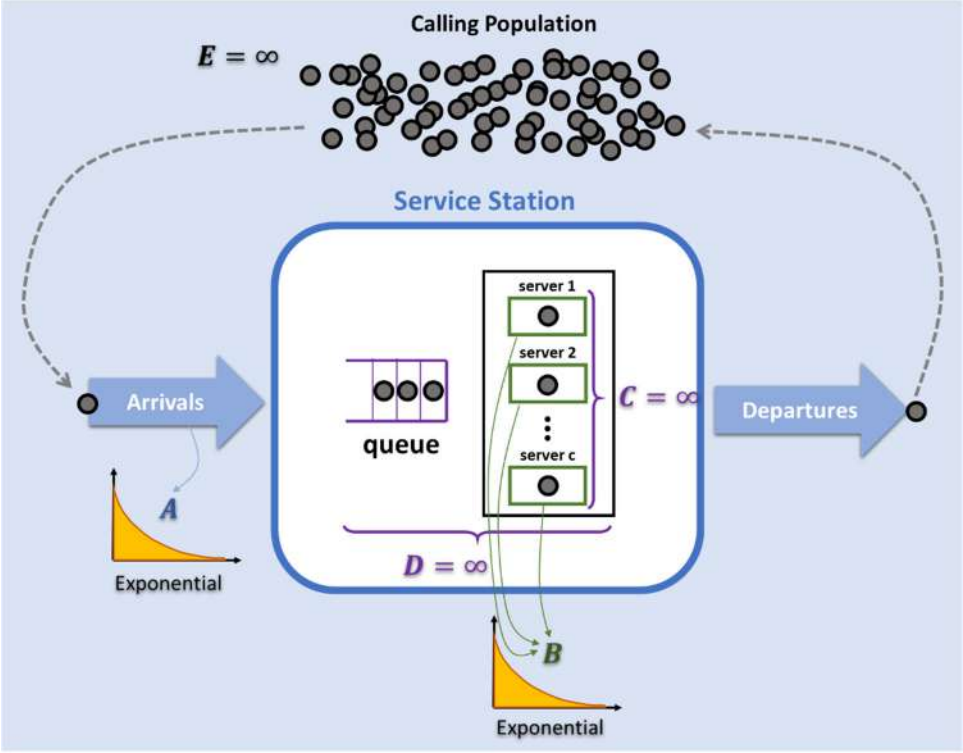


Figure 2-89: Illustration of M/M/∞/∞/∞ (Queues with Unlimited Service)

Examples of flow systems we can model as M/M/∞/∞/∞ queueing system:

- Customers at a self-service buffet can serve themselves and leave. There's no restriction on the number of customers that can eat at the same time: the buffet can serve more than 500 people and the historical maximum concurrent customers was 250.
- An over-designed retail parking lot is never filled. A car that enters will immediately find an empty spot.

Formulas for a generic M/M/∞/∞/∞ queuing system:

$r = \frac{\lambda}{\mu}$	$\rho = 0\ (no\ resource)$
$L_q = 0$	$W_q = 0$
$L = r = \frac{\lambda}{\mu}$	$W = \frac{1}{\mu}$

M/M/∞/∞/∞ queue example (mathematical solution)

Assumptions for this example:

- $\lambda =\ the\ arrival\ rate = 50\ \left(\frac{entity}{second}\right)$
- $\mu = service\ rate\ of\ an\ individual\ server = 20\ \left(\frac{entity}{second}\right)$
- $c =\ number\ of\ parallel\ servers\ =\ \infty\ \ (or\ no\ resource\ constraint)$
- $Queueing\ discipline\ is\ FIFO\ (First\ in\ First\ Out)$

Calculations of exact values:

$$r = \frac{\lambda}{\mu} = \frac{15}{20} = 0.75$$

$$L = 0.75$$

$$W = \frac{1}{20} = 0.05$$

$$L_q = W_q = 0$$

M/M/∞/∞/∞ queue example (simulation model)

Prerequisites:

Model Building Blocks (Level 1): **Source**, **Service**, **Sink**, **TimeMeasureStart**, **TimeMeasureEnd**, and **ResourcePool** blocks.

Math: Random variable, Random variate, [Poisson process], Exponential Distribution

We can modify the original M/M/1/∞/∞ model to an M/M/∞/∞/∞ by removing the resource constraint or increasing the resource pool's capacity to a level where it will not be a constraint:

1. Open the original M/M/1/∞/∞ model you built in the previous example.

   Before we modify the model, you may want to keep your original version intact. To do this, point to the **File** menu and then click **Save As** to save a copy with a different name.

2. Click the **seize** block, change its seize policy to **(alternative) resource sets** and remove any resource pool from the set (Figure 2-90).

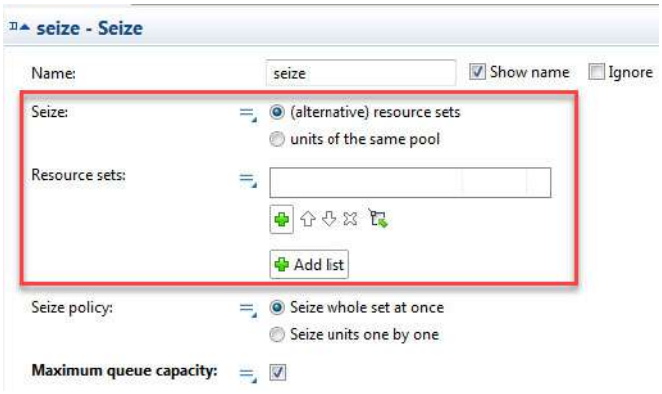


Figure 2-90: assigning an empty set as the resource (no resource constraint)

We haven't discussed the **(alternative) resource sets** choice for the **Seize** option in a **Seize** block. To explain it briefly, an empty resource gives the **Seize** flexibility to work without resources - in other words, with infinite resource. Alternatively, we could have increased the resource pool's capacity to a number large enough to guarantee no resource limitation.

3. Run the **Simulation** experiment and wait until the 10000 second simulation time is finished (you can check the model time in the **Developer** panel). If you use the PLE version, the model will stop after 50000 entities arrive, but your results should be comparable.

4. Click **statistics_Queue**, **statistics_System**, **tmE_Q**, and **tmE** to open their inspection windows (Figure 2-91).

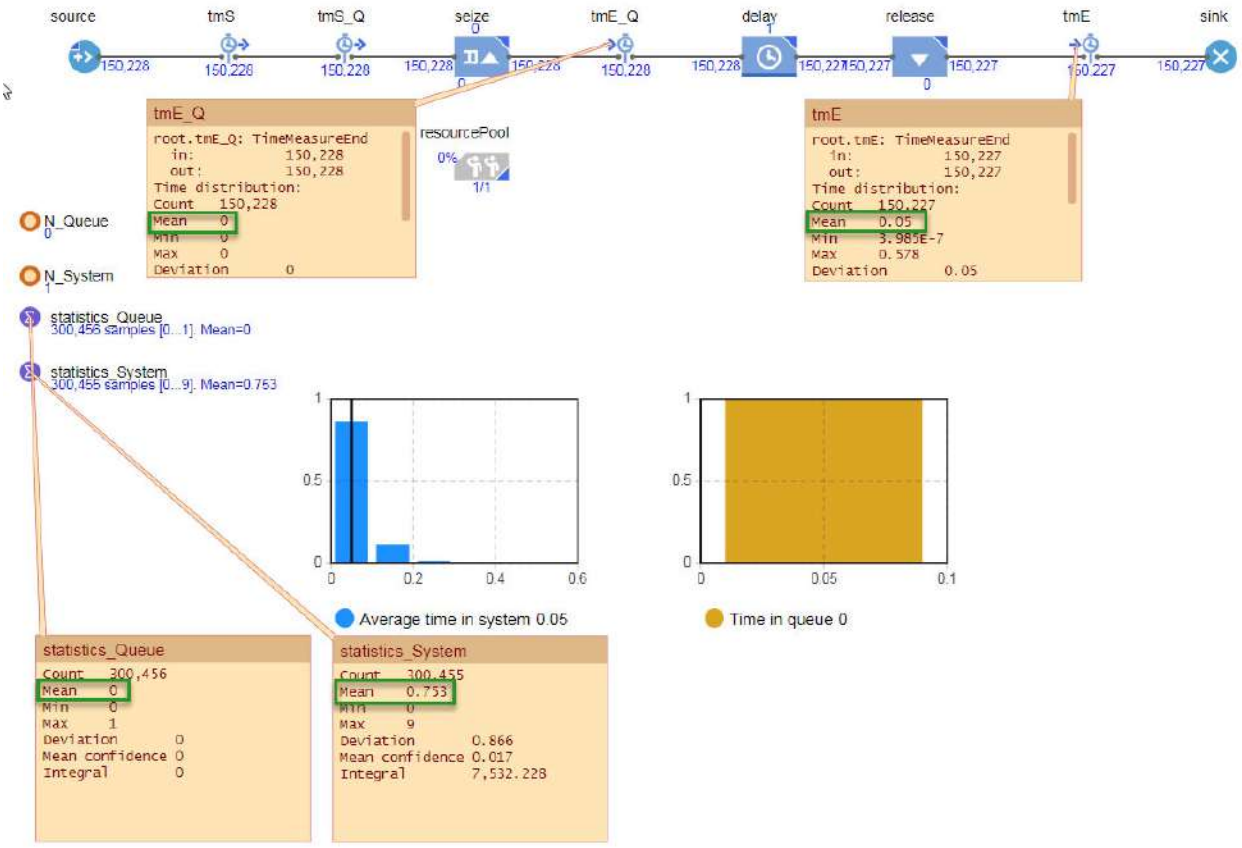


Figure 2-91: Simulation output for an M/M/∞/∞/∞ queuing system, λ=50, μ=20

Comparison of exact and simulated values are shown in Table 2-14:

Table 2-14: Comparison of exact and simulated values (M/M/∞/∞/∞ example)

Description	From simulation	From calculations
Long-run average number of entities in system	$\hat{L} = 0.753$	$L = 0.75$
Long-run average number of entities in queue	$\hat{L}_q = 0$	$L_q = 0$
Long-run average time spent in system per entity	$\widehat{W} = 0.05$	$W = 0.05$
Long-run average time spent in queue per entity	$\widehat{W}_q = 0$	$W_q = 0$

Again, the direct estimates of the simulation outputs are reasonably like the closed-form solutions of the performance metrics.

M/M/c/∞/d (finite-source queues)

Characteristics of the M/M/c/∞/d queueing system (Figure 2-92):

- Interarrival time = M (exponential)
- Service time = M (exponential)
- Number of parallel servers = c
- System capacity = ∞
- Calling population = d (finite)

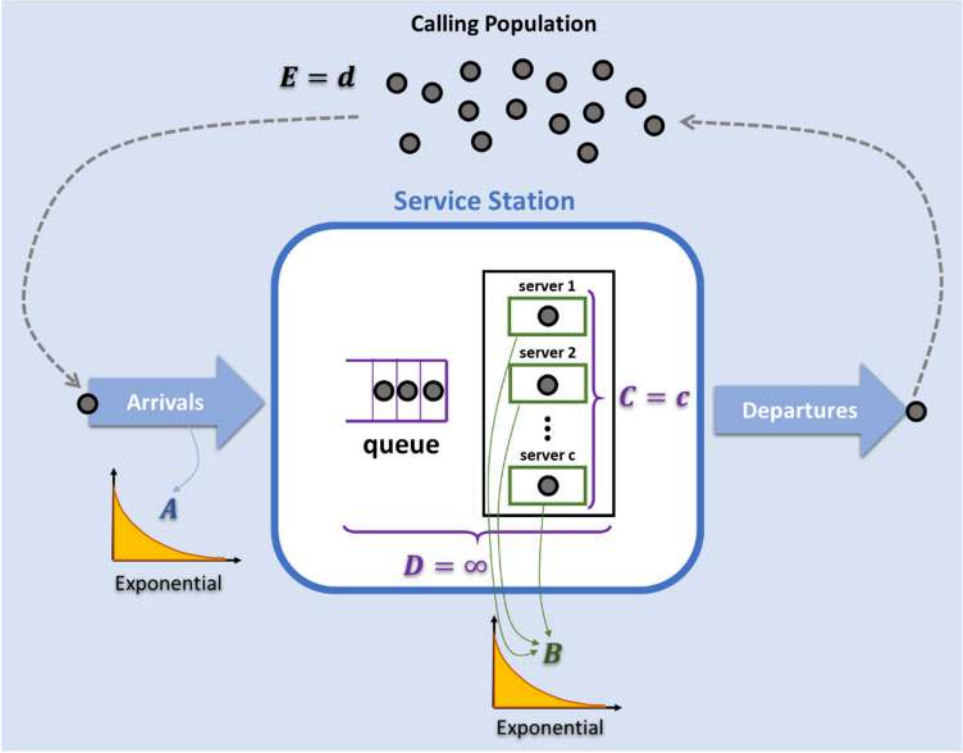


Figure 2-92: Illustration of M/M/c/∞/d (Finite-Source Queues)

Examples of flow systems that can be modeled as M/M/c/∞/d queueing system:

- An earthwork company has 10 backhoes that break down randomly, but with a known interarrival time. The calling population consists of the 10 backhoes and arrival to the system is associated with the breakdown. The resource units (servers) are the mechanics that fix the equipment. The number of servers tells us about the company's ability to repair several broken backhoes at one time. The calling population is limited in this scenario; this means we can't have more than 10 entities (backhoes) in the system at one time.
- A golf club has a limited number of members. While our analysis takes place, no members join or leave the club. Since the flow system is closed, only members can play in the golf course. After a member leaves the club, some time will pass (interarrival time) before they visit again.

Formulas used for a generic M/M/c/∞/d queuing system:

$$P_n = a_n P_0$$

$$a_n = \begin{cases} \binom{d}{n} r^n & (1 \le n < c), \\ \binom{d}{n} \frac{n!}{c^{n-c} c!} r^n & (c \le n \le d). \end{cases}$$

$$\binom{d}{n} = \frac{d!}{(d-n)!\, n!}$$

$$P_0 = \frac{1}{1 + \sum_{n=1}^{d} a_n} \equiv \frac{1}{1 + a_1 + a_2 + a_3 + \cdots + a_d}$$

$$r = \frac{\lambda}{\mu}$$

$$L = P_0 \sum_{n=1}^{d} n a_n$$

$$L_q = L - r(d - L)$$

$$W = \frac{L}{\lambda(d - L)}$$

$$W_q = \frac{L_q}{\lambda(d - L)}$$

$$\lambda_{eff} = \lambda(d - L)$$

$$\rho_{eff} = \frac{\lambda_{eff}}{c\mu} \equiv \frac{\lambda(d - L)}{c\mu}$$

M/M/c/∞/d queue example (mathematical solution)

Assumptions for this example:

- $\lambda = \ the\ arrival\ rate = \frac{1}{30}\left(\frac{entity}{second}\right)$
- $\mu = service\ rate\ of\ an\ individual\ server = \frac{1}{3}\left(\frac{entity}{second}\right)$
- $c = \ number\ of\ parallel\ servers \ = 2 \leftarrow (M/M/\boldsymbol{c}/\infty/d)$
- $d \ = Size\ of\ calling\ population \ = 5 \leftarrow (M/M/c/\infty/\boldsymbol{d})$
- $Queueing\ discipline\ is\ FIFO\ (First\ in\ First\ Out)$

Calculations of exact values:

$\lambda = \frac{1}{30}$

$\mu = \frac{1}{3}$

$c = 2$

$d = 5$

$r = \frac{1}{10}$

$a_1 = \binom{5}{1}\left(\frac{1}{10}\right)^1 = \frac{5}{10} = \frac{1}{2}$

$$a_2 = \binom{5}{2}\left(\frac{2!}{2^0 \times 2!}\right)\left(\frac{1}{10}\right)^2 = \left(\frac{5!}{2!\,3!}\right)(1)\left(\frac{1}{100}\right) = \left(\frac{5 \times 4}{2}\right)(1)\left(\frac{1}{100}\right) = \frac{1}{10}$$

$$a_3 = \binom{5}{3}\left(\frac{3!}{2^1 \times 2!}\right)\left(\frac{1}{10}\right)^3 = \left(\frac{5!}{3!\,2!}\right)\left(\frac{3}{2}\right)\left(\frac{1}{1000}\right) = \left(\frac{5 \times 4}{2}\right)\left(\frac{3}{2}\right)\left(\frac{1}{100}\right) = \frac{15}{1000} = \frac{3}{200}$$

$$a_4 = \binom{5}{4}\left(\frac{4!}{2^2 \times 2!}\right)\left(\frac{1}{10}\right)^4 = \left(\frac{5!}{4!\,1!}\right)\left(\frac{4 \times 3}{4}\right)\left(\frac{1}{10{,}000}\right) = \left(\frac{5}{1}\right)(3)\left(\frac{1}{10{,}000}\right) = \frac{15}{10{,}000} = \frac{3}{2{,}000}$$

$$a_5 = \binom{5}{5}\left(\frac{5!}{2^3 \times 2!}\right)\left(\frac{1}{10}\right)^5 = (1)\left(\frac{5 \times 4 \times 3}{8}\right)\left(\frac{1}{100{,}000}\right) = (1)\left(\frac{15}{2}\right)\left(\frac{1}{10{,}000}\right) = \frac{15}{200{,}000} = \frac{3}{40{,}000}$$

$$P_0 = \left(1 + \frac{1}{2} + \frac{1}{10} + \frac{3}{200} + \frac{3}{2000} + \frac{3}{40{,}000}\right)^{-1} = \frac{40{,}000}{64{,}663} = 0.61859$$

$$L = \frac{40{,}000}{64{,}663} \times \left(1 \times \frac{1}{2} + 2 \times \frac{1}{10} + 3 \times \frac{3}{200} + 4 \times \frac{3}{2000} + 5 \times \frac{3}{40000}\right) = \frac{30{,}055}{64{,}663} = 0.46479$$

$$L_q = 0.4648 - \frac{1}{10} \times (5 - 0.4648) = 0.01127$$

$$W = \frac{0.4648}{0.1 \times (5 - 0.4648)} = 3.07458$$

$$W_q = \frac{0.01127}{0.1 \times (5 - 0.4648)} = 0.07458$$

$$\lambda_{eff} = \frac{1}{30} \times (5 - 0.465) = 0.15117$$

$$\rho_{eff} = \frac{0.15117}{2 \times \frac{1}{3}} = 0.22676$$

M/M/c/∞/d queue example (simulation model)

Prerequisites:

Model Building Blocks (Level 1): Source, Service, Sink, TimeMeasureStart, TimeMeasureEnd, RestrictedAreaStart, RestrictedAreaEnd, and ResourcePool blocks.

Model Building Blocks (Level 2): Source block.

Math: Random variable, Random variate, [Poisson process], Exponential Distribution

We'll modify the base M/M/1/∞/∞ model to a M/M/c/∞/d by increasing the resource pool's capacity, limiting the number of arrivals to *d*, and adding a loop that continually brings the entities back to the system.

1. Open the M/M/1/∞/∞ model we built in the previous example.

   Before we modify the model, you may want to keep your original version intact. To do this, save the file, point to the **File** menu and then click **Save As** to save a copy with a different name.

2. Click the **source** and change the arrival rate to $\frac{1}{30}$ per second. If you are using interarrival time, the distribution should be `exponential(1.0/30, 0)` seconds (Figure 2-93). We also need to check the **Limited number of arrivals** checkbox and set **Maximum number of arrivals** to 5. With these settings, only 5 entities will arrive (with exponential interarrivals times) and continue to circulate in the system.

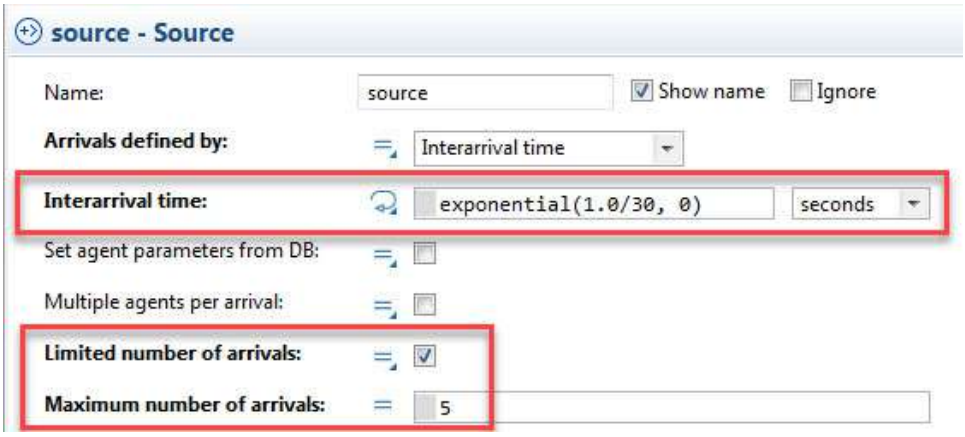


Figure 2-93: Limiting the number of arrivals to 5

3. Click **resourcePool** and change its **capacity** to 2.

4. We need to move the code that updates the number of entities in the system from the source to the **tmS** block. Click **source**, cut the code snippet in the **On exit** field, click **tmS** and paste the code in the **On enter** field (Figure 2-94).

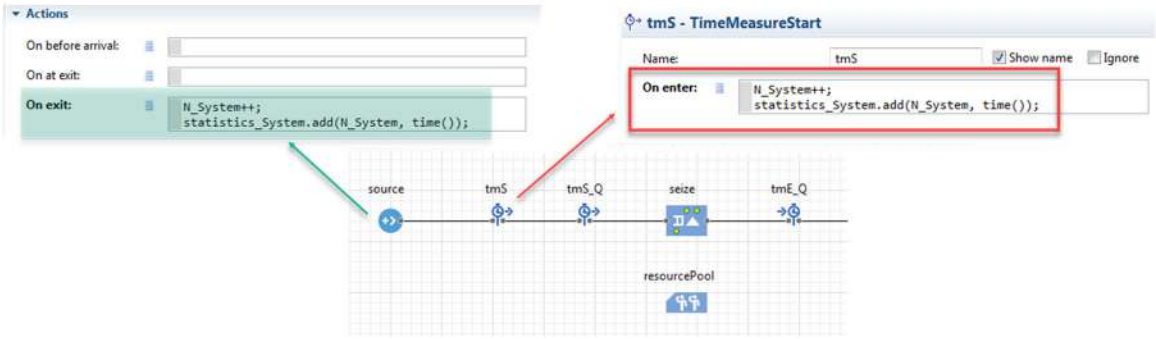


Figure 2-94: Moving the Number in system updating code from source to tmS

5. We also need to move the code that updates the number of entities in the system from **sink** to **tmE**. Click **sink**, cut the code snippet in the **On enter** field, click **tmE** and paste the code in the **On enter** field (Figure 2-95).

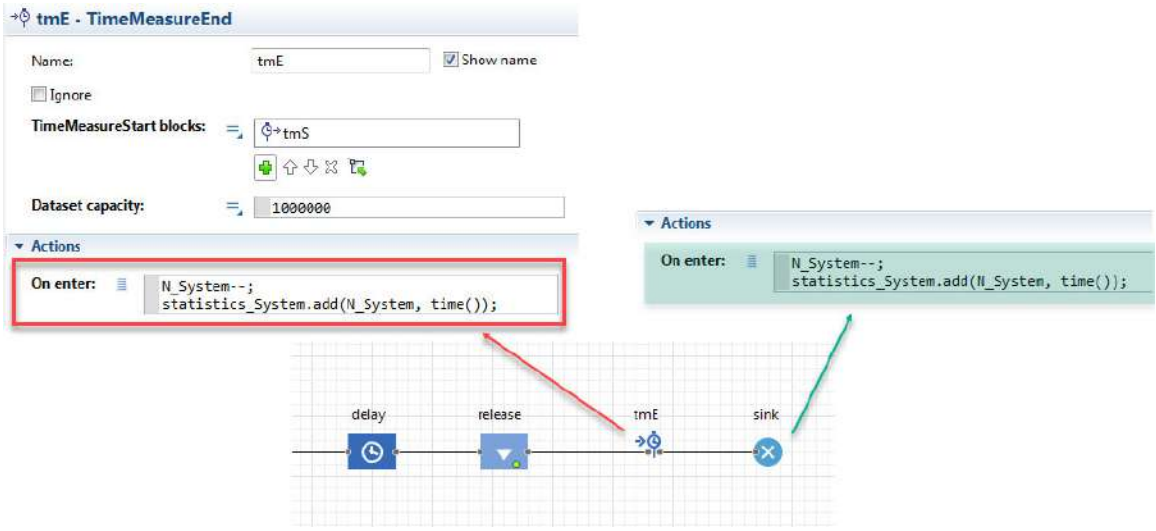


Figure 2-95: Moving the Number in system updating code from sink to tmE

6. Delete the **sink** from the end of the flowchart.

7. Add a **Delay** from PML and put on top of the current flowchart, rename it to **re_enter**, set the delay time to `exponential(1.0/30)` seconds – the same as the source's interarrival time – and check the **Maximum capacity** box (Figure 2-96).

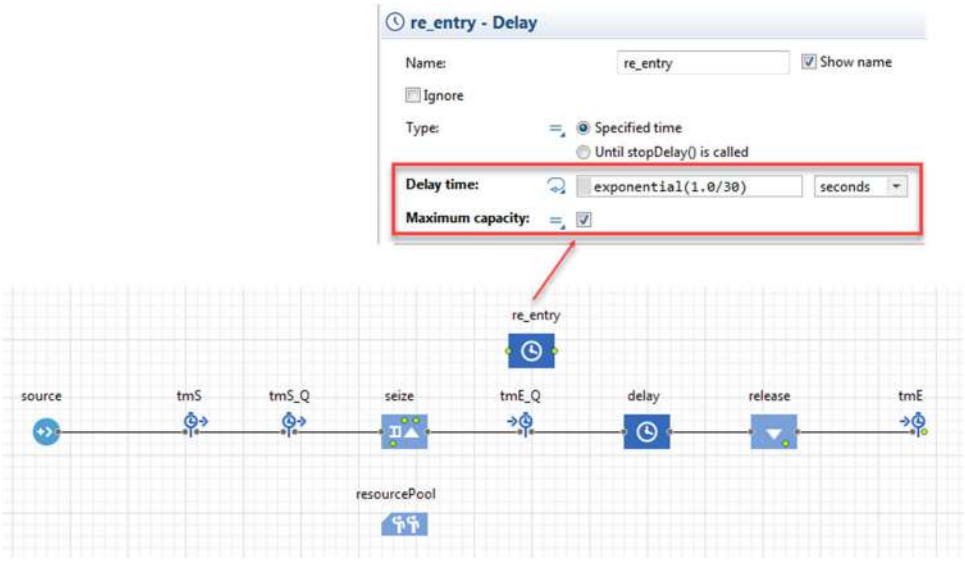


Figure 2-96: add the delay and rename it to re_entry

8. Connect the *out* port of **tmE** to *in* port of **re_entry** (Figure 2-97).

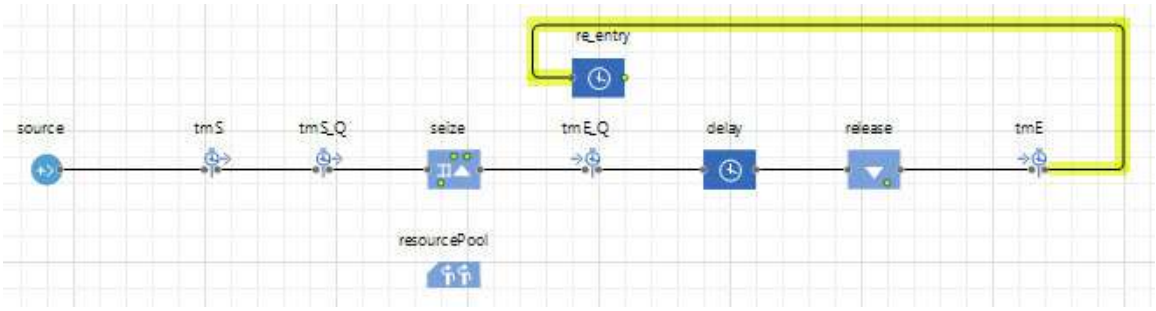


Figure 2-97: connecting ports of tmE and re_entry

9. Connect the *out* port of **re_entry** to the *in* port of **tmS** (Figure 2-98).

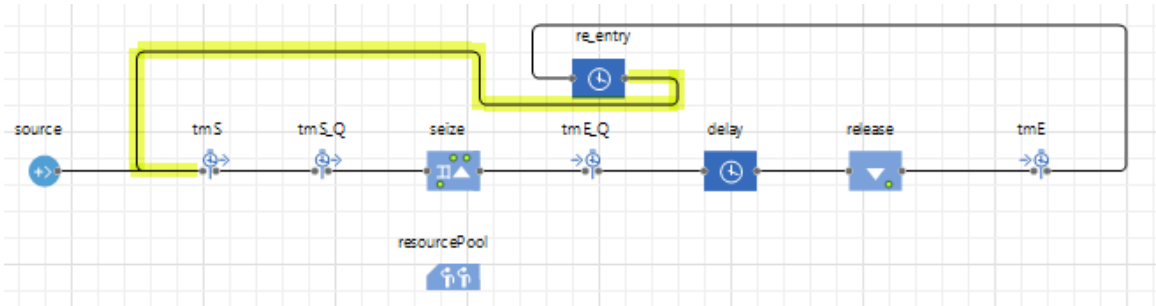


Figure 2-98: connecting ports of re_entry and tmS

The **re_enter** delay keeps the entities out of the circulation for some time (equivalent to the exponentially distributed interarrival time). When an entity leaves the circulation, it uses the "*out*" port of **tmE**. After the interarrival time passes (the time inside **re_entry**), the entity uses the "in" port of **tmS** to reenter the circulation. This explains why we moved the code that updates the number of entities in the system to "*in*" port of **tmS** and "out" port of **tmE**.

10. In the **Projects** window, click the **Simulation** experiment and then change the **Stop** time to 100000 (one hundred-thousand) to make it 10 times longer.
11. Run the **Simulation** experiment and wait until the 100000 second simulation time is finished. You can use the **Developer** panel to check the model time.
12. Click **statistics_Queue**, **statistics_System**, **tmE_Q**, and **tmE** to open their inspection windows (Figure 2-99).

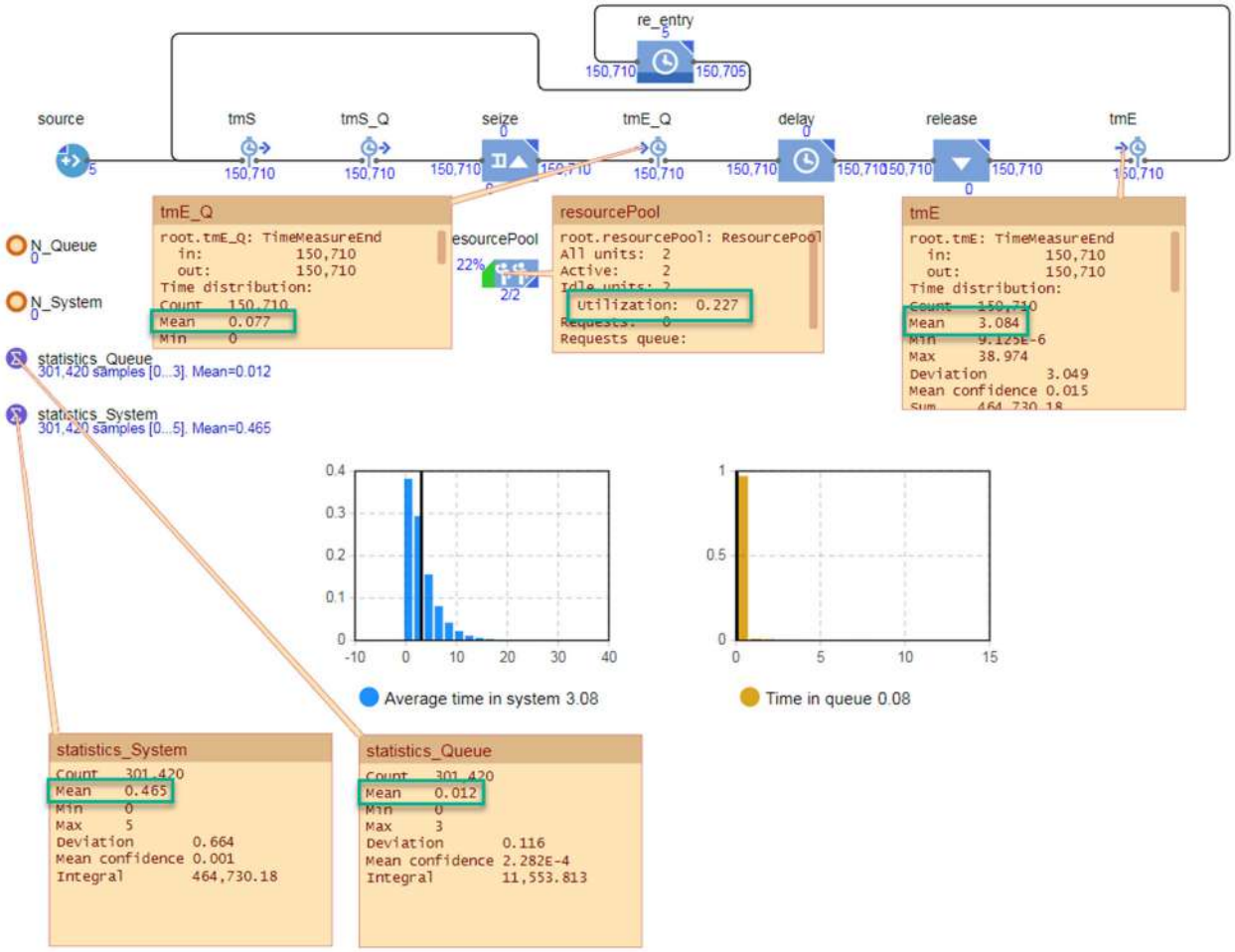


Figure 2-99: Simulation output for a M/M/c/∞/d system, λ=1/30, μ=1/3, c=2, d=5

Comparison of exact and simulated values are shown in Table 2-15:

Table 2-15: Comparison of exact and simulated values (M/M/c/∞/d example)

Description	From simulation	From calculations
Long-run average number of entities in system	$\hat{L} = 0.465$	$L = 0.4648$
Long-run average number of entities in queue	$\hat{L}_q = 0.012$	$L_q = 0.0113$
Long-run average time spent in system per entity	$\widehat{W} = 3.084$	$W = 3.0746$
Long-run average time spent in queue per entity	$\widehat{W}_q = 0.077$	$W_q = 0.0746$
Utilization of resource pool (average of 2 servers)	$\hat{\rho}_{eff} = 0.227$	$\rho_{eff} = 0.2268$

G/G/c/∞/∞ (general input, general service, multi-server queues)

There are many other closed-form solutions for more general queueing systems which don't use exponential distributions for their interarrival and service times. In this section, we jump into a general (non-exponential) queuing systems.

Characteristics of the G/G/c/∞/∞ queueing system (Figure 2-100):

- Interarrival time = arbitrary with known mean and variance
- Service time = arbitrary with known mean and variance
- Number of parallel servers = c
- System capacity = ∞
- Calling population = ∞

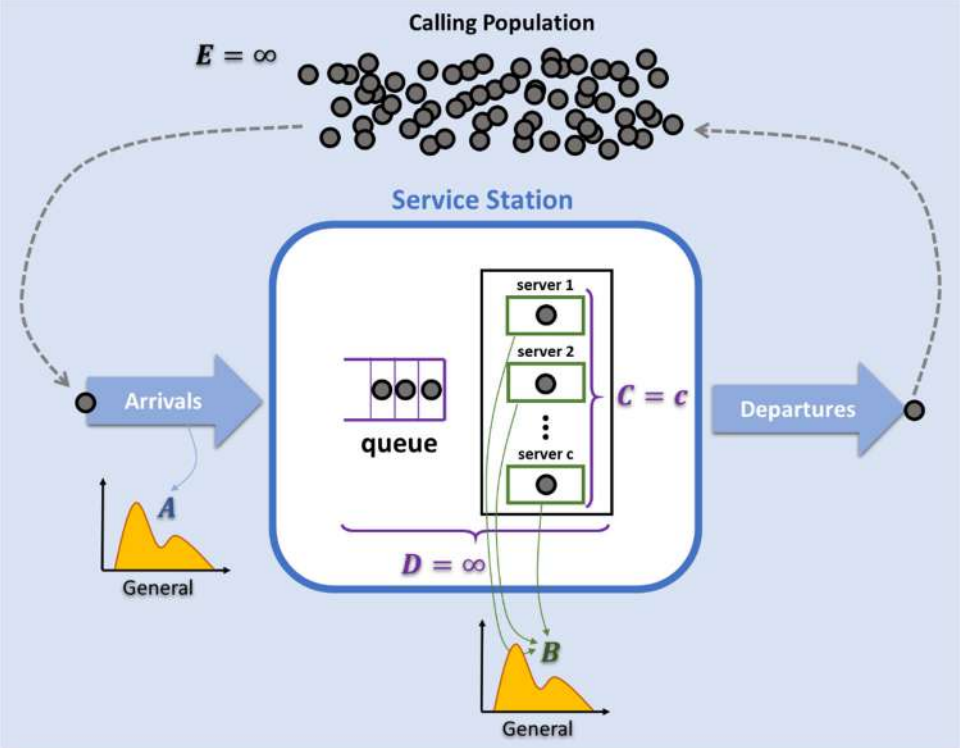


Figure 2-100: Illustration of G/G/c/∞/∞ (General Input, General Service, Multi-Server Queues)

Instead of trying to find an exact closed form solution, we'll discuss a close approximation formula. Both Kingman and Sakasegawa developed widely-used formulas for W_q (long-run average waiting time in the queue).

Kingman's equation/VUT equation for a G/G/1/∞/∞ system is:

$$W_q = \left(\frac{CV_A^2 + CV_S^2}{2}\right)\left(\frac{\rho}{1-\rho}\right)E[S]$$

For G/G/c/∞/∞ systems, we'll focus on the more general form of the Kingman's formula proposed by Sakasegawa (1977):

$$W_q = \left(\frac{CV_A^2 + CV_S^2}{2}\right)\left(\frac{\rho^{\sqrt{2(c+1)}-1}}{c(1-\rho)}\right)E[S]$$

- $V \rightarrow variability$: $\left(\frac{CV_A^2+CV_S^2}{2}\right)$
- $U \rightarrow utilization$: $\left(\frac{\rho^{\sqrt{2(c+1)}-1}}{c(1-\rho)}\right)$
- $T \rightarrow time\ scale\ or\ average\ service\ time$: $E[S]$

$A = random\ interarrival\ time$
$S = random\ service\ time$
$\lambda = \frac{1}{E[S]}$
$\mu = \frac{1}{E[S]}$

CV_A: $coefficient\ of\ variation\ of\ random\ variable\ A$
$CV_A^2 = \frac{Var[A]}{E^2[A]}$
CV_S: $coefficient\ of\ variation\ of\ random\ variable\ S$
$CV_S^2 = \frac{Var[S]}{E^2[S]}$

$r = \frac{E[S]}{E[A]}$

$\rho = \frac{r}{c} = \frac{\lambda}{c\mu} = \frac{E[S]}{cE[A]}$

A few notes to make:

- This formula is an approximation, and its accuracy can vary significantly by the queue's inputs and structure. Whitt (1993) provides a full discussion of different scenarios.
- This formula's use of *U* is related to the resource utilization but it isn't the actual utilization. This means it isn't confined to being between zero and one, as in next section's example.

M/G/c/∞/∞ queue example (mathematical approximation)

Prerequisites*:*

PML Blocks: **Source**, **Service**, **Sink**, **TimeMeasureStart**, **TimeMeasureEnd**, **RestrictedAreaStart**, **RestrictedAreaEnd**, and **ResourcePool** blocks in the **MODEL BUILDING BLOCKS – LEVEL 1**.

Math: Random variable, Random variate, [Poisson process], Exponential Distribution

The formula above applies to any queueing system with General distributions for their interarrival and service times. For simplicity, we use the following formula to approximate performance metrics of a M/G/c/∞/∞ system:

- $\lambda = \ the \ \ arrival \ rate = 50 \ \left(\frac{entity}{second}\right)$
  - Interarrival time is exponential
- $\mu = the \ service \ rate \ of \ an \ individual \ server = 20 \ \left(\frac{entity}{second}\right)$,
  - Normal distribution $(a = 0.04, b = 0.06)$
- $c = \ number \ of \ parallel \ servers \ = \ 3 \ \leftarrow \ (M/G/\boldsymbol{c}/k/\infty)$
- $Queueing \ discipline \ is \ FIFO \ (First \ in \ First \ Out)$

Calculation of the long-run average time spent in queue per entity by the VUT approximation formula:

$A \ = \ exponential \ (50, 0) \rightarrow \ \lambda = 50$

$S = \ uniform \ (0.04{,}0.06) \rightarrow a = 0.04, b = 0.06$

$$E[A] = \frac{1}{\lambda} = \frac{1}{50} = 0.02$$

$$Var[A] = \frac{1}{\lambda^2} = \frac{1}{50^2} = 0.0004$$

$$E[S] = \frac{a+b}{2} = \frac{0.04+0.06}{2} = 0.05$$

$$Var[S] = \frac{(b-a)^2}{12} = \frac{(0.06-0.04)^2}{12} = 0.000033333$$

$$\rho = \frac{E[S]}{cE[A]} = \frac{0.05}{0.02 \times 3} = 0.833333333$$

$$CV_A^2 = \frac{Var[A]}{E^2[A]} = \frac{0.0004}{0.02^2} = 1$$

$$CV_S^2 = \frac{Var[S]}{E^2[S]} = \frac{0.000033333}{0.05^2} = 0.013333333$$

$$V = \left(\frac{CV_A^2 + CV_S^2}{2}\right) = \frac{1 + 0.013333333}{2} = 0.506666667$$

$$U = \left(\frac{\rho^{\sqrt{2(c+1)}-1}}{c(1-\rho)}\right) = \frac{(0.833333333\ )^{\sqrt{2\times(3+1)}-1}}{3 \times (1-0.833333333\ )} = 1.43302199950$$

$T = E[S] = 0.05$

$W_q = VUT = 0.125 \times 1.433021999 \times 0.05 = \ 0.0363$

M/G/c/∞/∞ queue example (simulation model)

Prerequisites*:*

Model Building Blocks (Level 1): **Source**, **Service**, **Sink**, **TimeMeasureStart**, **TimeMeasureEnd**, and **ResourcePool.**

Math: Random variable, Random variate, [Poisson process], Exponential Distribution

We can easily modify the M/M/c/∞/∞ model to an M/G/c/∞/∞ by assigning the desired distribution to the delay time. If we assigned any distribution other than the exponential to the interarrival times, the queuing system would have changed to a G/G/c/∞/∞. To keep the manual calculation simple, we decided to only change the delay time.

To build the model:

1. Open the M/M/c/∞/∞ model you built.

   If you want to keep the original file intact, point to the **File** menu, click **Save as…** and save the model with a different name.

2. Click **delay** and change the delay time to `uniform(0.04,0.06)`.

   This distribution was selected in a way that it has the same expected value as the previous example (that is, `exponential(20, 0)` also had an expected value of 0.02).

3. Run the **Simulation** experiment and wait until the 10000 second simulation time is complete (you can check the model time in the **Developer** panel). If you use the PLE version, the model will stop after 50000 entities arrive, but your results should be comparable.

4. Click **statistics_Queue**, **statistics_System**, **tmE_Q**, **tmE**, and **resourcePool** to open their inspection windows. Figure 2-101 shows the results of an iteration.

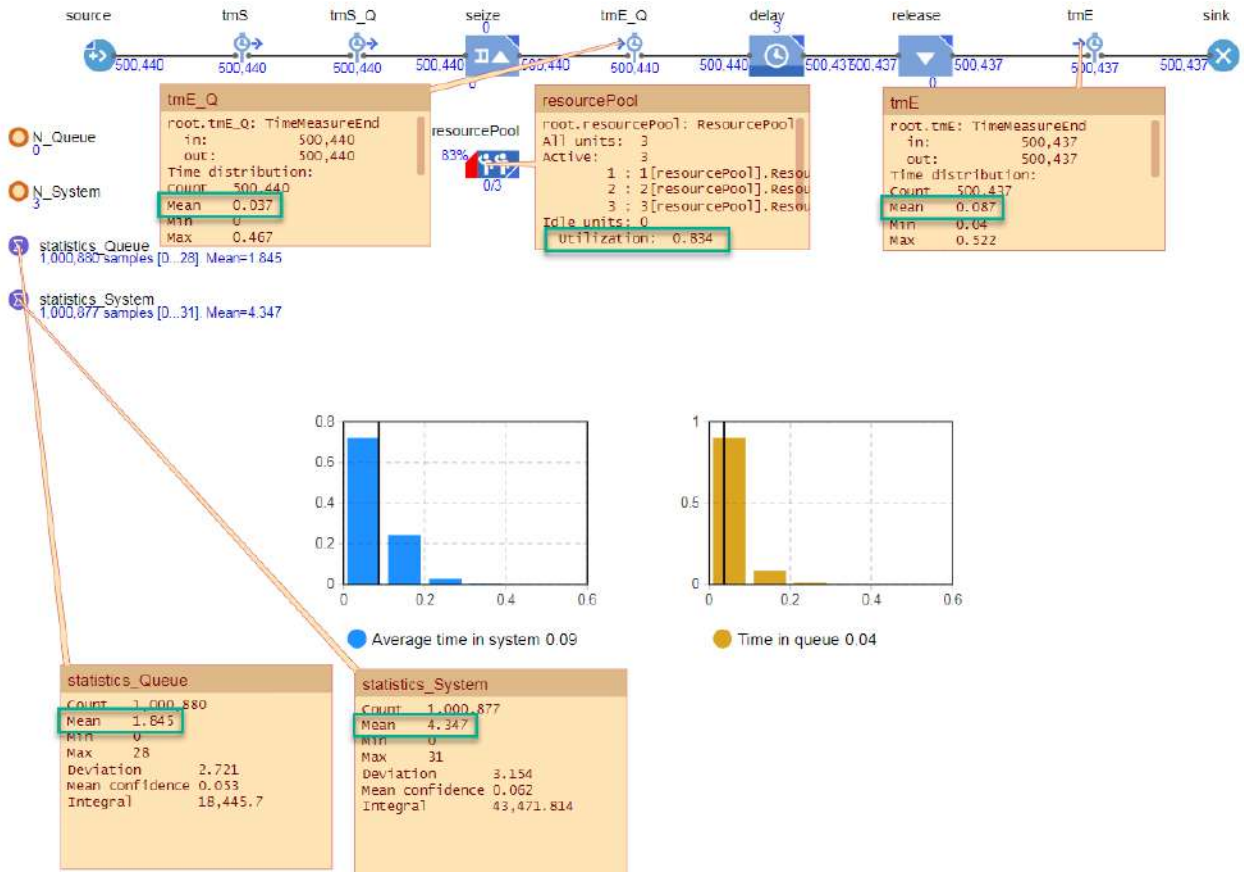


Figure 2-101: Simulation output for a M/G/c/∞/∞ system, λ=55, μ=20 (uniform), c=3

Comparison of exact and simulated values:

Description	From simulation	Approximation
Long-run average number of entities in system	$\hat{L} = 4.347$	N/A
Long-run average number of entities in queue	$\hat{L}_q = 1.845$	N/A
Long-run average time spent in system per entity	$\widehat{W} = 0.087$	N/A
Long-run average time spent in queue per entity	$\widehat{W}_q = 0.037$	$\widetilde{W}_q = 0.0363$
Utilization of resource pool (average of 3 servers)	$\hat{\rho}_{eff} = 0.834$	N/A

As you can see, the VUT formula reasonably approximated the W_q (simulation estimate: 0.037; mathematical output: 0.03630). However, its accuracy isn't consistent across different scenarios. We almost reached the limit of closed form solutions for queueing systems. In the next chapter, we'll move beyond the limitations of queueing networks and explore the unlimited world of flow systems and their process-centric models.

Chapter 3 : State-of-the-Practice Process Centric Models

Process centric modeling in action

What we covered in the previous chapter was processes simple enough to have analytical solutions. Those models and their analytical solutions were indispensable in the absence of simulation models. But as you've seen, they're limited and inflexible.

Technically, you don't need to know anything about queueing systems to learn process-centric models. In fact, most people who learned simulation started by learning about processes! However, you've learned a great deal by exploring the queueing systems, as they're the methodological basis of process-centric simulation models. Besides their educational value, you can use the previous chapter's closed form solutions to establish baseline approximation (albeit at a high abstraction level) of more complex simulation outputs.

In Chapter 2, we saw the simulation approximations of several queueing systems side by side with their analytical solutions. You should keep in mind when we're using simulation models, our approach in estimating their outputs differs from analytical solutions. Rather than build a probability model of the process, we build a virtual imitation of the process and let it run (move forward in time). The simulation approach is flexible and wide-ranging; we can model just about any system we could imagine as a flow system with a process-centric simulation model. Using such a powerful and comprehensive methodology isn't a small task, but the alternatives are tedious probability models. Hopefully by now you're convinced simulation models are much simpler than any other alternative.

In the previous chapter, we focused on foundational blocks such as **Queue**, **Delay**, **Seize**, **Release** and **ResourcePool** to build the simple archetypes. Now, we're ready to tap into the power of process-centric models. Learning about these other blocks will allow us do (almost) anything we want!

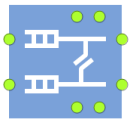
We'll start by briefly reviewing Process Modelling Library (PML) blocks. At first glance, you might think they can't possibly meet the many needs that come with simulating real-life processes. But like Lego blocks, we can use a small number of generic blocks to build amazingly complex things.

Two appendices, **Model Building Blocks 1 & 2**, describe their PML blocks in detail. We don't need to learn everything about each block in this section, it is just an introduction to their main use. For now, use Table 3-1 to familiarize yourself with them.

Table 3-1: Process Modeling Library (PML) building blocks

Block name	Process Modeling Library Icon	Description
Source		Generates the entities and adds them to them to the process
Sink		Disposes of the entities leaving the model
Queue		Stores entities that cannot move forward in the process immediately
Delay		Delays the entities from moving forward in the process
ResourcePool		Defines and stores the resource unit(s) of a certain type
Seize		Assigns some resource units to the entities passing through; contains an embedded queue
Release		Releases one or more resource units seized by the entity that's passing through it
Service		The block is a combination of three inner blocks: **Seize**, **Delay**, and **Release** that simplifies modelling common processes
SelectOutput		Separates the entities into two groups either probabilistically or based on a condition
SelectOutput5		Separates the incoming entities into five groups either probabilistically or based on some conditions
MoveTo		Moves entities from their current location to a set destination

Schedule		Returns numerical or boolean values based on current time or date of the model; it defines values that change during the simulation.
Hold		Blocks the flow of entities based on logical rules independent of the resource or the system's capacity constraints
Split		Creates one or more new entities while preserving the original
Combine		Waits for two entities, then outputs either one of the incoming entities or a new entity of a specified type
ResourceAttach		Attaches a resource unit to the entity that seized it, so they can move together
ResourceDetach		Detaches a resource unit that was attached to an entity, so they can move separately
Batch		Accumulates entities, then outputs them contained within a new agent
Unbatch		Extracts all entities contained in the incoming entity and outputs them
Pickup		Adds entities from a linked Queue block to the internal container of the entity that's passing through it (e.g., a taxi that picks up passengers)
Dropoff		Removes the contained entities from the entity passing through it; the entity serving as the container should have picked up one or more entities by a Pickup block
ResourceTaskStart		Defines the start of a preparation flowchart branch, which models a resource units' tasks they must perform before they can start helping the entity.
ResourceTaskEnd		Defines the end of the wrap-up flowchart branch, which models the resource units' wrap-up tasks they must perform before they can serve another entity.

Match		Synchronizes two streams of entities by matching pairs according to a given criteria; contains embedded queues for each stream.
Assembler		Assembles entities (from one to five sources) into a single entity based on a predefined bill of material
Exit		Takes away entities from a flowchart and lets the user to manually specify what to do with them (e.g., entering a separate flowchart)
Enter		Takes in already existing entities (e.g., entities that exited from another flowchart by an Exit block) and insert them into a point of a flowchart
TimeMeasureStart		Couples with a **TimeMeasureEnd** object to measure the time the entities spend between them
TimeMeasureEnd		Couples with a **TimeMeasureStart** object to measure the time the entities spend between them
RestrictedAreaStart		Couples with a **RestrictedAreaEnd** object to limit the number of entities that could be in the subsection of the flowchart that's enclosed between them
RestrictedAreaEnd		Couples with a **RestrictedAreaStart** object to limit the number of entities that could be in the subsection of the flowchart that's enclosed between them
PMLSettings		An optional block that defines some global settings related to all Process Modeling Library blocks present on the same graphical diagram where you placed **PMLSettings**

As mentioned, this table introduces you to some of the available PML blocks. Our two appendices, **MODEL BUILDING BLOCKS 1 & 2**, describe their properties in detail. When you come across an example model, use the Prerequisite box to determine which appendix has necessary information about the blocks that example uses.

Top-level agent, model initialization, and extension points (callbacks)

No matter which modeling paradigm you choose (Discrete Event, Agent Based, or System Dynamics), AnyLogic treats all model environments as agents. The models we built in the previous chapter all had one model building environment – the default "Main".

We'll review the specifics and ramifications of Agent Based modeling in our next chapter. For now, you need a basic understanding. Let's start with an incomplete definition that still can help us understand the meaning of **Top-level agent**:

> *Agent Based modeling compartmentalizes a system that has several influential actors into encapsulated components. Instead of dealing with all the components in one flat system in which all innerworkings of all components are exposed in one giant environment, each component [that usually] has some level of autonomy will be separated and modeled as a self-containing agent that interact with other components (agents). No matter how many agents are in a model, a top-level agent is necessary to encompass everything else inside of it.*

Consider a dedicated slaughter house that sends a carcass to a second facility for processing. At its end, the packing process sends the packaged meats to a distribution warehouse that sorts them and ships the orders to retailers. In a flat modeling approach, we model these three processes in one environment with all their inner workings exposed (Figure 3-1). The model is one large process, with sections conceptualized as sub-processes.

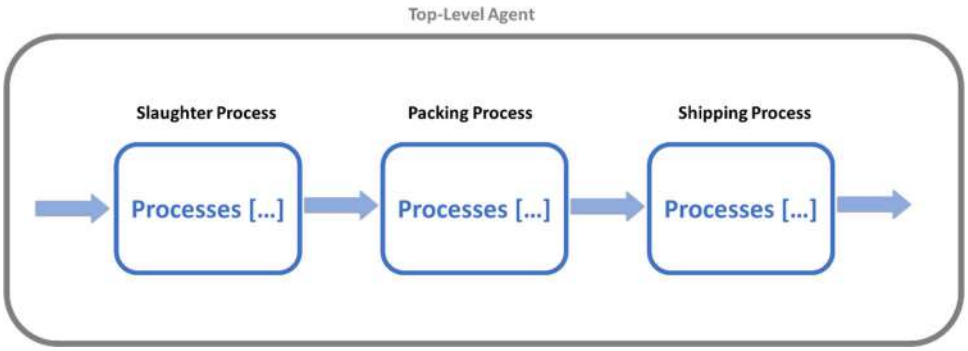


Figure 3-1: All processes in one containing environment

In an agent-based design of the same exact process, we reimagine the flow in a more modular and compartmentalized fashion (Figure 3-2). In this design, we model each facility as an agent that encapsulates only the process that occur in that facility. As shown, these facilities interact much like they do in real life (e.g., output of slaughter house will be the meat packing facility's process input).

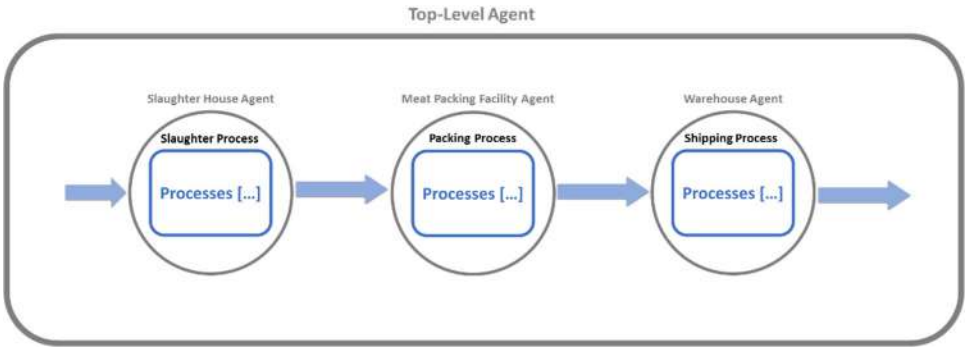


Figure 3-2: Processes are encapsulated in agents (sub containing environments)

While the benefits of agent-based modeling may not be apparent with this example, there's more than simply encapsulating a process. Figure 3-3 is a better illustration of the Agent Based design, since it shows that we can't view each agent's inner processes from the outside.

Figure 3-3: The inner workings of agents are encapsulated and their interactions with other agents are through specific interfaces

A modularized model treats each component as if it was a separate model connected to other components through specific interfaces. For example, the meat packing agent can accept some input (carcasses) from the slaughter house agent, perform its self-contained process (packaging), and then send its output (packaged cuts) as the input to the warehouse agent.

There are other important benefits, but this bird's eye view of Agent Based modeling allows us to clarify a point we made earlier: *"AnyLogic treats all model environments, including the default (top-level), as agents"*. For cases in which we don't use any agent-based design, we put all our processes inside one environment. However, without us noticing, these processes are in a default environment AnyLogic treats as an agent. This environment is, by default, called Main and is the agent that encompasses everything in your model. When you run your simulation experiment, it uses the selection set in the top-level agent field as the universe your model runs in (Figure 3-4). We'll show in a later example that you can change this to another agent in your model.

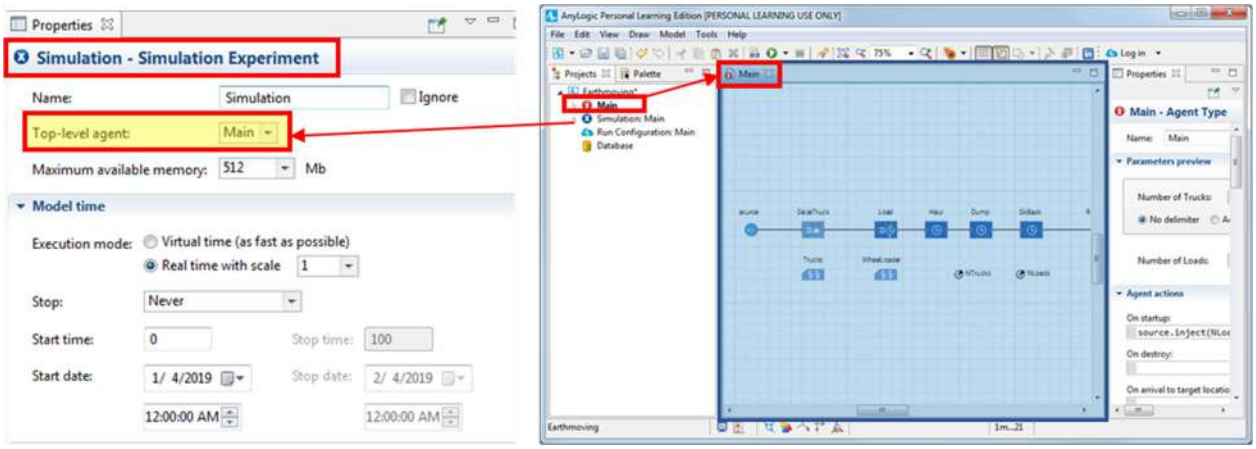


Figure 3-4: The agent "Main" as the top-level agent, which the Simulation experiment uses for the model environment

We won't use agent-based modeling design in our examples in this chapter. In the next chapter, we'll learn how to use agent-based modeling to associate internal attributes and behavior to individual entities and resource units. We'll also go over how to build modular, scalable process-centric models with the help of agent-based design. For now, it's enough to know your process-centric models are by default inside a one giant agent (the top-level agent).

Model startup field

You'll often want to initialize your model with certain values or start it in a certain state. However, the starting of an AnyLogic model has multiple stages. The order events occur is important when you want to initialize the model in a specific manner. In these cases, you'll need to understand the differences between each step of the initialization process to determine the right step for your initialization task. The following is an overview of a model's initialization order.

1. The top-level agent (Main, by default) and everything else inside it will be built and set up. This is analogous to the moments that led up to the Big Bang. Unless you build everything by code and you decide to not use the graphical user interface (possible, though impractical), AnyLogic will automatically perform all these tasks for you. After everything is set up, AnyLogic executes a function called `create()` that marks the end of the setup procedure.

2. Immediately after AnyLogic executes the `create()` function, your model's universe comes into existence. It's at this point you can access the model's elements. *The create function doesn't do anything new, it just marks the end of the setup procedure for an agent.* In more complex, modular agent-based models where other environments exist inside the top-level agent, local "Big Bangs" also will happen for those environments. Like the top-level agent, those local creations occur after the call of `create()` function for each agent; however, creation of the top-level agent happens first.

You should be aware that contrary to our big bang analogy, during the setup step and even after the creation of the model, *time* still doesn't exist. In other words, since nothing has started to change in our universe, there's no concept of time (real or model) yet. Time won't begin to exist until the next described step starts.

3. Upon creation of the model (top-level agent and everything inside it), the objects inside the model environment start their initialization activities (that is, scheduling initial events). If the top-level agent has

some inner agents, the model initialization must occur in these objects first. In other words, the function `start()` should be called for every single agent that exists in the model before the top-level agent starts its initialization activities.

The starting order differs from the creation order. As we mentioned in step 2, our model creates the top-level agent and then creates the agents it encompasses. In contrast, "start" happens in inner agents first, then their encompassing environment (e.g., top-level agent).

Modelers often want to add custom initialization tasks before the model starts to run, but those tasks often related to some of the model's components. This means the model should execute them after the model creation and all the default start tasks, but before the model starts to run. That's why every agent (including the top-level agent) has a code extension **On startup** field. The model executes the code you add to this field during the final stage of the start process.

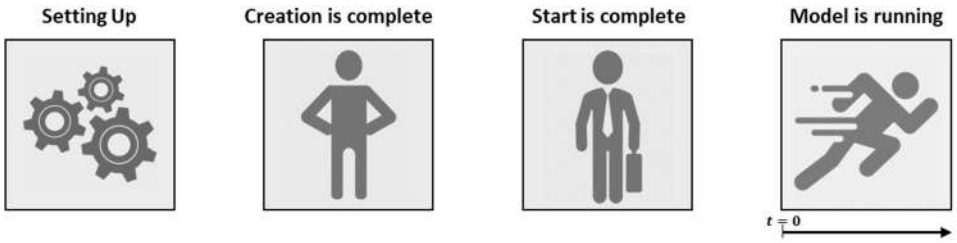


Figure 3-5: Conceptual illustration of setting up, creating, starting, and running a model

Figure 3-5 shows a conceptual representation of the three steps of model initialization, plus the stage where it starts running. All components of the model are constructed and then a function `create()` is called from the top-level agent, down to all inner agents inside it. After the completion, all components of the models are ready and available - but they have not started. For the components to start finalizing their initialization, a `start()` function is called from the inner objects and then up to the top-level agent. After AnyLogic calls the start function for all model components, the initialization is complete, and the model is ready to run.

You won't need to know much about model initialization. In most of your models, all these steps take a fraction of a second without you even noticing. However, the most common scenario of model initialization is when the modeler wants to execute some extra tasks after the model creation and at final stage of the model start (right before the model time begins, as discussed in the start step). As shown in Figure 3-6, each agent has an **On Startup** field where you can add custom code. For each agent, this code gets executed when the agent is created, and all its start activities are completed.

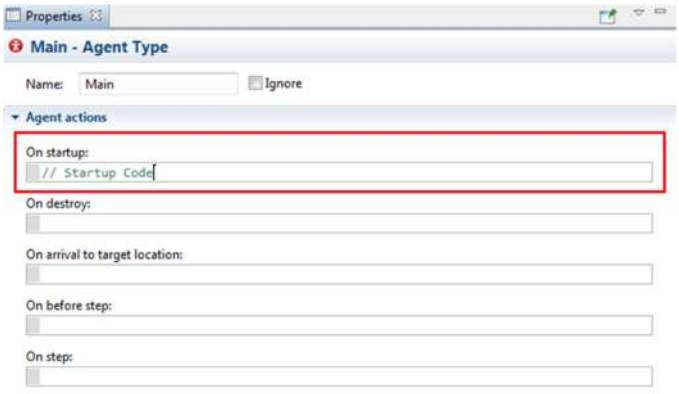


Figure 3-6: "On startup" field of the Main agent (top-level agent)

> *The code in the top-level agent's "On startup" field (Main by default) gets executed after everything inside the model has been constructed, created, and started but just before the model starts to run. Therefore, all components of the model are available and accessible in this field. The top-level agent's "On startup" field is a special place in which you have access to all components of the model and let you add your final initialization tasks, just before the model starts to run.*

In the next section, we'll look at other extension points we can use to execute custom tasks at specific moments other than initialization.

Extension points (callbacks)

In the previous section, we reviewed the **On startup** field of top-level agent (**Main**) – this field is what's known as extension points or callbacks. Similarly, all blocks in the Process Modeling Library (PML) have extension points, which are places where you can use code to insert actions or expressions. We use extension points to execute custom tasks at specific moments other than initialization.

An important part of writing code in extension points is understanding how code gets executed in a regular computer program. Computers execute codes line by line; think of the code as a list of tasks written in which the computer reads these tasks one by one, executing them in order. It is possible to break from the default line-by-line order of execution with control statements (e.g., if a variable isn't greater than 10, skip five lines). Another way is by function calls - such as in Figure 3-7 where the `factorial` function is called on line 6; the computer then skips to the line where that function starts (line 9), runs the code until its completion, then returns to its original location where it replaces the function call with the returned value.

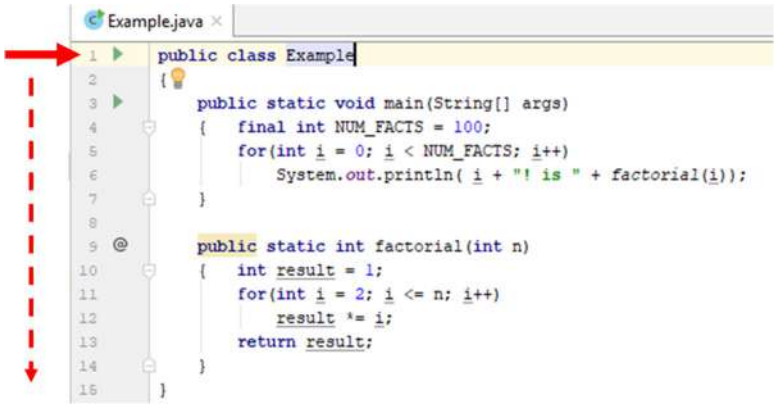


Figure 3-7: Example of a line by line execution of code in Java

The time it takes to execute the code depends on the hardware specification of the machine that runs it and the machine's state at the runtime. Most computer programs don't associate (virtual or real) time with the code execution, the computer runs the code and returns the results as fast as it can.

Like a computer program, a simulation model is made up of lines of code that run in a specified order. However, the code in a simulation model and a regular computer program are fundamentally different:

> *In a simulation model, when you tell the computer to do a task by code, you also must tell it WHEN to execute the code. In other words, a simulation model can execute your code exactly like a regular program, but it also needs to know when you want to execute it. This is because a simulation program maps the tasks to a virtual clock (model time) and executes them accordingly. You may also map that model time to the real time (with a scale) to be able watch the execution of tasks in a virtual imitation.*

We'll see how you can execute your code at any point in time (model time) with the help of the **Event** and **Dynamic Event** objects. These objects let us assign an *explicit* time of execution to your code. But our focus here is a common occurrence in process centric models, where the desired moment for some code to execute is tied to another event in the process. For example, you may want to print the time when an entity uses the **Sink** block to leave the model. The problem is we don't know when each entity will leave the flowchart. It happens as part of the simulation execution and due to the model randomness, entities leave the model at different times in each run.

Extension points/callbacks are placeholders tied with blocks and let you *implicitly* associate the execution time to your code. These callback fields are available in the **Actions** section of the Properties of each block (where applicable).

Figure 3-8 shows the callbacks for the Queue and Seize blocks. The **Queue** block has extension points at **On enter**, **On at exit**, **On exit**, and **On remove** moments, which are relative to the queue. Depending on each block's functionality, it makes sense to have extensions points specific to that block and therefore, extension fields might differ for each block. The **Seize** block shares some fields with the Queue block but also has fields for **On seize unit** and **On prepare unit**, while also lacking the **On at exit** field.

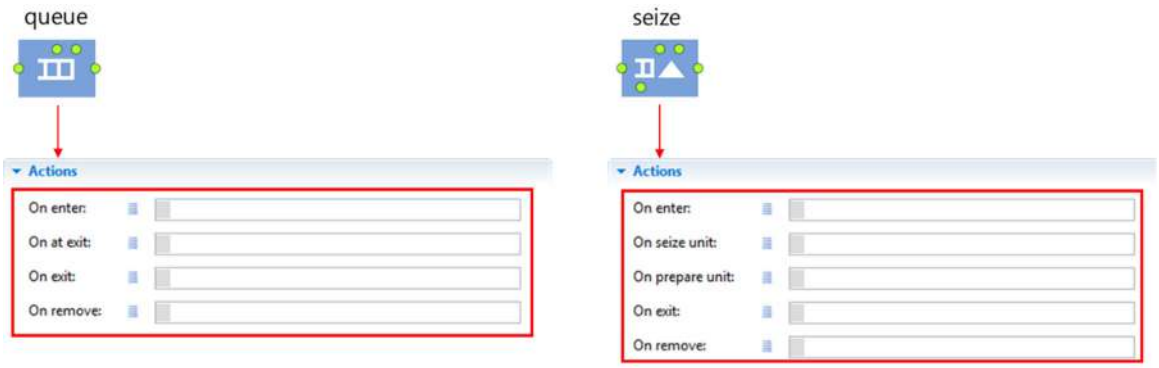


Figure 3-8: Actions section of Queue block

As we mentioned, any code in a simulation model needs to know the moment it should get executed. You can consider the callbacks as fields that implicitly associate execution time to your code. This assignment is *implicit* because the associated time depends on the flow of each run but nevertheless a time is assigned to the code.

Table 3-2 gives you an overview of the extension points for some of commonly used PML blocks. At this point you don't need to know all the callback fields. Just familiarize yourself with the diverse types of available callbacks and you'll learn in later examples how to use them to add customized tasks in your process-centric models.

Table 3-2: An overview of extension points (callbacks) for a selected number of PML blocks

	Sink	Source	Delay	Queue	SelectOutput	MoveTo	Seize	Release	Service
On before arrival	No	Yes	No	No	No	No	No	No	No
On enter	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
On release	No	No	No	No	No	No	No	Yes	No
On at exit	No	Yes	Yes	Yes	No	No	No	No	Yes
On exit	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
On 'Wrap-up' terminated	No	No	No	No	No	No	No	Yes	No
On exit (true)	No	No	No	No	Yes	No	No	No	No
On exit (false)	No	No	No	No	Yes	No	No	No	No
On exit (preempted)	No	No	No	Yes	No	No	Yes	No	Yes
On exit (timeout)	No	No	No	Yes	No	No	Yes	No	Yes
On seize unit	No	No	No	No	No	No	Yes	No	Yes
On enter delay	No	No	No	No	No	No	No	No	Yes
On prepare unit	No	No	No	No	No	No	Yes	No	No
On task suspended	No	No	No	No	No	No	Yes	No	Yes
On task resumed	No	No	No	No	No	No	Yes	No	Yes
On task terminated	No	No	No	No	No	No	Yes	No	Yes
On discard	No	Yes	No	No	No	No	No	No	No
On remove	No	No	Yes	Yes	No	Yes	Yes	No	Yes

The following provides a brief explanation of when the code gets executed for each callback shown in Table 3-2:

On before arrival [Source] - Executed before the entity is generated

On enter [Sink, Delay, Queue, SelectOutput, MoveTo, Seize, Release, Service] - Executed when the entity enters the block

On release [Release] - Executed when the resource unit has been released

On at exit [Source, Delay, Queue, Service] - Executed when the entity is ready to exit the block. To better understand the difference between this field and the next one (**On exit**), refer to the Push-Pull Protocol section.

On exit [Source, Delay, Queue, SelectOutput, MoveTo, Seize, Release, Service] - Executed when the entity exits the block.

On 'Wrap-up' terminated [Release] – Executed when the resource unit that is in a wrap-up branch or when they're in the middle of wrap up task and the task is terminated (preempted)

On exit (true) [SelectOutput] - Executed when the entity exits the block by the upper "*outT*" (condition is true) port

On exit (false) [SelectOutput] - Executed when the entity exits the block by the lower "*outF*" (condition is false) port

On exit (preempted) [Queue, Seize, Service] - Executed when the entity exits by the "*outPreempted*" port because the queue (or embedded queue) was full. This field is only visible if the "Enable preemption" option is enabled. To read more about this option refer to MODEL BUILDING BLOCKS - LEVEL 2 APPENDIX.

On exit (timeout) [Queue, Seize, Service] - Executed when the agent exits by the "*outTimeout*" port because of waiting in the queue for more than a set threshold. This field is only visible if the "*Enable exit on timeout*" option is enabled. To read more about this option refer to MODEL BUILDING BLOCKS - LEVEL 2 APPENDIX.

On seize unit [Seize, Service] - Executed when a resource unit is seized

On enter delay [Service] - Executed when the entity begins the internal delay

On prepare unit [Seize, Service] - Executed when a resource unit is prepared

On task suspended [Seize, Service] – Executed if the block allows preemption and another block with a higher priority grabs the entity's resource, suspending its current task. This field is only visible if the task preemption policy is set to **Wait for original resource** or **Seize any resource**. To read more about this option refer to TECHNIQUE 6: USING TASK PREEMPTION.

On task resumed [Seize, Service] - Executed when an entity that had their task preempted by a higher priority block is able to grab a resource unit and resume the its suspended task. This option is only visible if task preemption policy is set to **Wait for original resource** or **Seize any resource**. To read more about this option refer to TECHNIQUE 6: USING TASK PREEMPTION.

On task terminated [Seize, Service] – Executed if the block allows preemption and another block with a higher priority grabs the entity's resource. This field is only visible if the task preemption policy is set to **Terminate serving**. To read more about this option refer to TECHNIQUE 6: USING TASK PREEMPTION.

On discard [Source] - Executed when the entity can't exit the block because of the downstream blockage. This option is only visible if the **Forced pushing** checkbox is unchecked and agents that can't exit are set to be destroyed. To read more about this option refer to MODEL BUILDING BLOCKS - LEVEL 2 APPENDIX.

On remove [Delay, Queue, MoveTo, Seize, Service] - Executed when the entity is intentionally removed from this block by calling the `remove()` function of the agent (entity).

Many of these fields are related to concepts and functionalities we haven't covered. Table 3-2 and the descriptions above introduce you to the possibilities we'll cover in greater detail later.

Pull vs Push protocol

As we've discussed, process-centric models are modeling "flow systems" where things move in metaphorical flow channels. Two fundamental mechanisms can trigger the movement of entities (the thing that flows) in the flow system: push protocol and pull protocol. A flow system that follows the push protocol is a *push system* and one that follows the pull protocol is a *pull system*. More formally we can define these two system types:

> *A push system schedules the release of work based on demand, while a pull system authorizes the release of work based on the system statues (Hopp and Spearman, 2000).*

This definition works well for a manufacturing context, but we could make it better fit process-centric models:

> *In a push system, an entity leaves (released) from a flowchart block when its task at that block is complete. In a pull system, the release of a ready-entity from a flowchart block depends on the demand of downstream block(s).*

In other words: for push systems, the entity attempts to leave the block as soon as it's done, regardless of the status of downstream (succeeding) blocks. This means each block works autonomously - it doesn't need to communicate with downstream blocks before the entities depart (Figure 3-9).

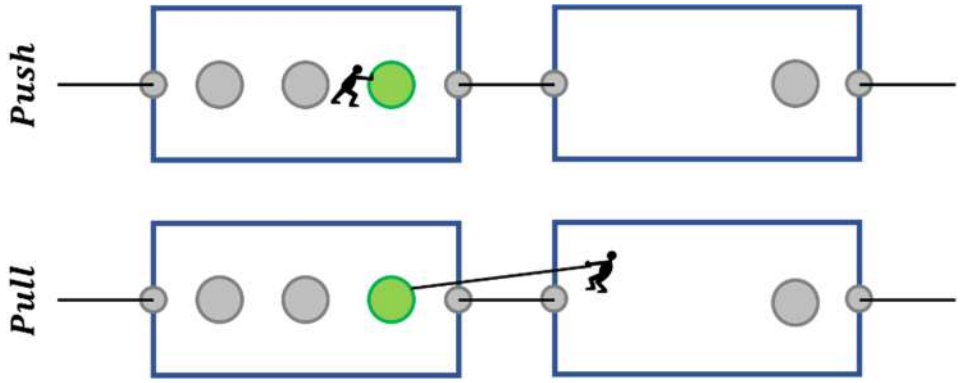


Figure 3-9: Push vs Pull protocol (the green entity is done and is ready to move forward)

In contrast, for pull systems, each block checks the ability of the next block to take in a new entity, which can only move forward if the next block is ready to take it (Figure 3-9).

In general, a system that follows the pull protocol is closer to the natural flow of entities in real systems. That's why AnyLogic's default is to implement the pull protocol for process-centric models for everything but the **Source** block; however, you can change these properties.

Implementation of Push and Pull protocols in AnyLogic

Warning! The exact implementation of the pull and push protocol in AnyLogic might be different from the following explanations. Our objective here is to develop a high-level intuition about the inner workings of the protocols in context of blocks and not reviewing the actual algorithms.

When there's no bottleneck in a flow system's entities pass smoothly, push and pull protocol behave the same. Let's use an example that has two **Delay** blocks: both blocks have a capacity of three and the left block has three entities in it (1, 2, 3). Each circle represents an entity, with its number inside it. The timer above each entity shows the elapsed delay time.

Let's assume all entities spend an equal time in the delay. This means the delay will send the entities out in a FIFO manner. The subscript in the times (t_1 and t_2) tell us model time has passed from t_1 to t_2.

Figure 3-10 shows the push system version of this example. Entity 1 is ready to depart at t_2 and the left block pushes it out (without looking into the right block). Since the right block isn't full, the entity is successfully pushed out in zero model time.

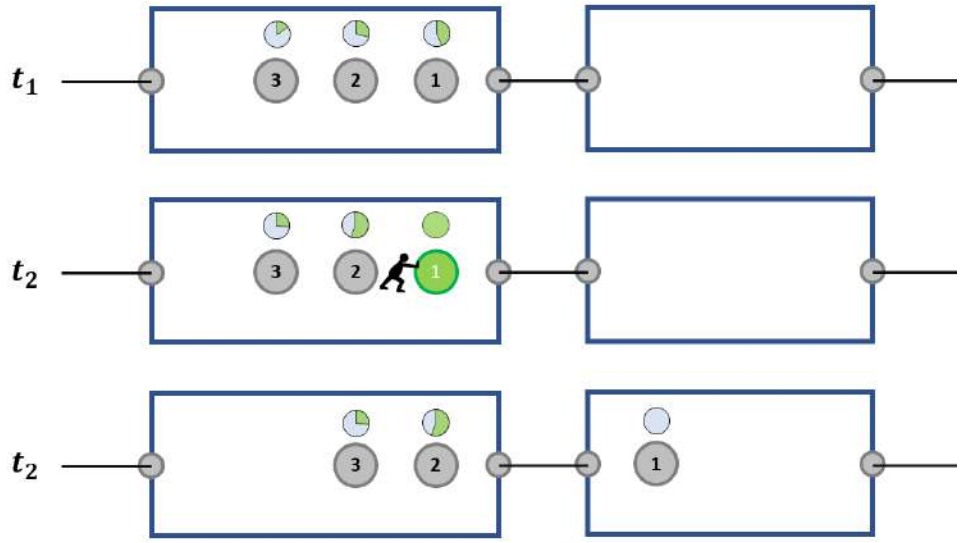


Figure 3-10: Push systems without blockage in downstream

This is also the case for the pull system, shown in Figure 3-11. When the entity is ready to depart, it is assigned to the downstream block (the block to the right). If that block has unfilled capacity, it pulls the ready entity (entity 1) from the left block. If that block doesn't have unfilled capacity, the entity must wait till the right block is ready to pull it in (that is, the downstream block has at least one available capacity).

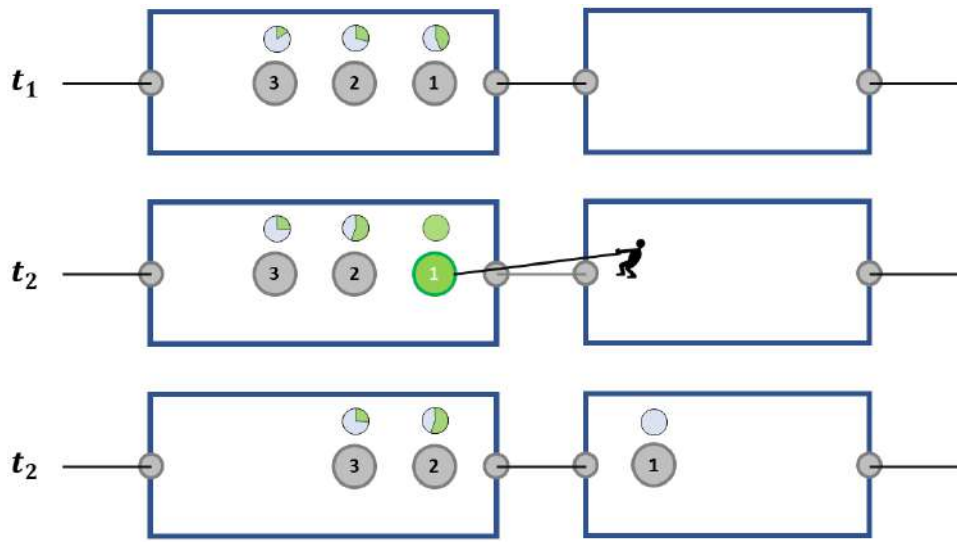


Figure 3-11: Pull systems without blockage in downstream

Since the downstream block (the right block) is empty and open, there's no difference between the left block pushing an entity or the right block pulling it. You'll start to see the effect of pull protocol in a system where there are blockages and bottlenecks. In a general setting with blockages, the two protocols are implemented in the following ways:

Push Protocol: Each block pushes a done entity to the next block without communicating to it or knowing about its availability. Figure 3-12 shows what happens in a scenario where an entity wants to move between two consecutive blocks.

Since the right block is full at time t_1, its "*in*" port is blocked (shown by an orange outline), preventing entities from arriving until at least one entity leaves. At time t_1, the left block isn't blocked because none of the entities inside of it are ready to depart. Then at time t_2, the delay of entity 1 in the left block is finished. This means the left block will try to push it out – even though the right block is still full and its "*in*" port is still blocked.

In this case, you'll see a runtime error. The error message will be something like "*Exception during discrete event execution, Model logic error*". A red outline shows the port the unsuccessful exit attempts initiated. AnyLogic doesn't allow implicit (hidden) buffering when entities can't enter the receiving block. This means, in push systems you'll always be notified about an overflow situation and will be able to locate the bottleneck.

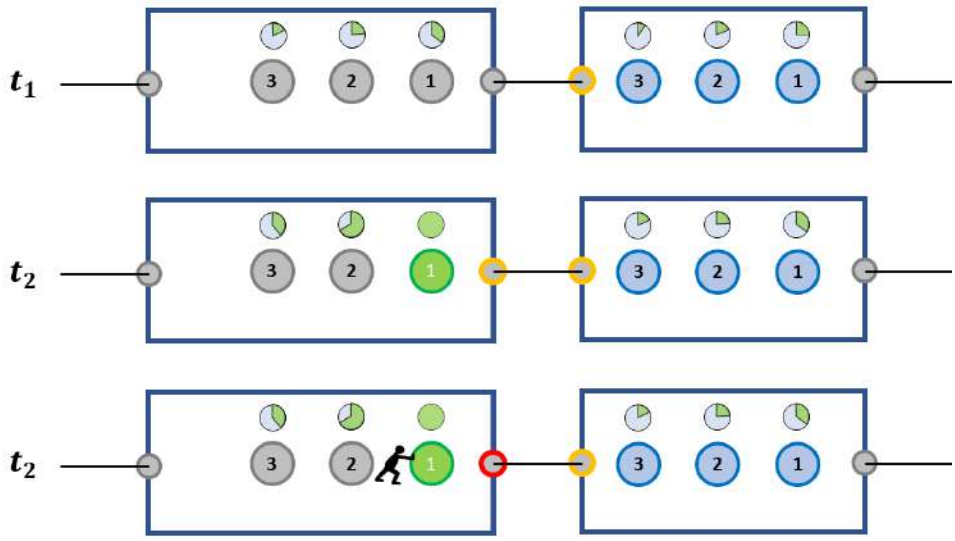


Figure 3-12: Push system with blockage in downstream

Pull protocol: In the "*pull*" method of ejecting entities, the sending block notifies the receiving block an entity is ready to exit. If the receiving block can't accept the entity (that is, the block is full and its "in" port is blocked), the entity that is ready to exit must wait. When the receiving block has availability, it opens its "*in*" port and notifies the upstream block to pass the entity. Said another way, the downstream block pulls the ready entity when it has availability.

Figure 3-13 shows this procedure in more detail. At time t_1 the right block is full and its "*in*" port is blocked. This causes AnyLogic to add an orange outline around the right block's "*in*" port. At time t_2, the delay for entity 1 is over and it's ready to leave. However, the right block is still blocked. As a result, the left block's "*out*" port gets an orange outline. This shows an entity is ready to depart but is blocked by its successive block.

> *In general, when the receiving block's "in" port is shown by an orange outline color, it is full and it is blocking entities from entering it. However, this doesn't mean there's an entity in the preceding block that's trying to enter. The sending block's "out" port turns orange only if it holds an entity that is ready to depart but is being blocked by the receiving block.*

Continuing with the example shown in Figure 3-13, at t_3 the entity 1 from the right block leaves and opens space for an entity. At the same time, the right block opens its blocked "*in*" port and signals its availability to the left block. This results in entity 1 from the left block to enter the right block. Exactly after pulling the entity, the right block closes its "*in*" port since it is full with three entities again.

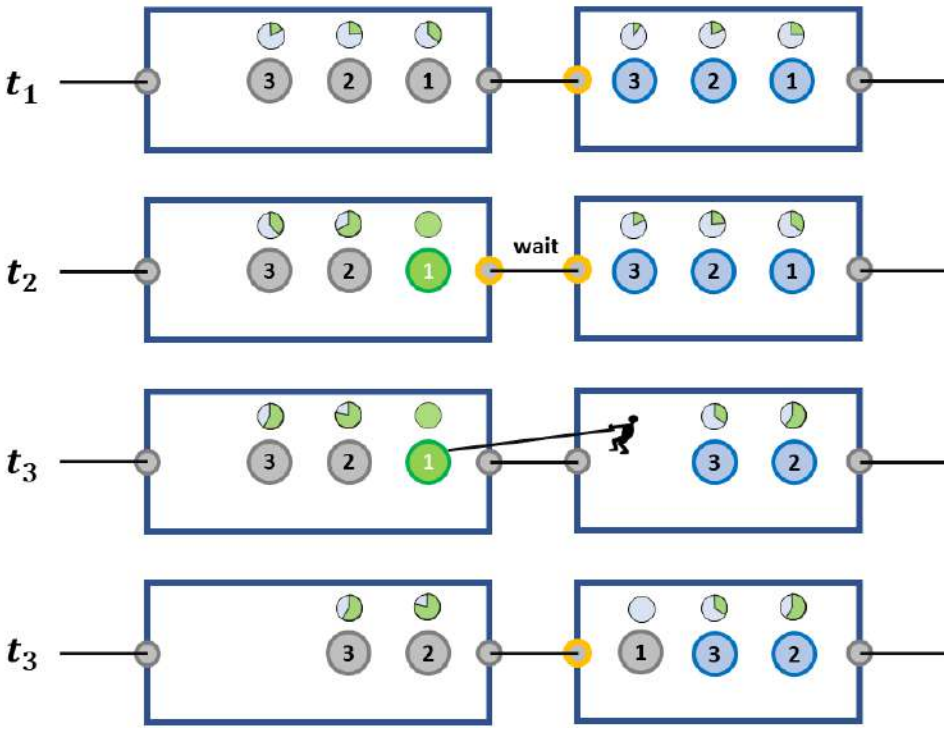


Figure 3-13: Pull system with blockage in downstream

Backward ripple effect of Pull protocol (right to left influence)

Even without you noticing, AnyLogic uses the pull protocol (and the implications that come with it) by default. However, you should understand what and why the pull protocol causes the model to exhibit specific behaviors. In flow systems, you'll need a clear understanding of why bottlenecks and accumulation occur to effectively verify your model.

The flow of entities in a flowchart is from left to right. That's why we intuitively follow the process in this direction. However, when you're implementing the pull protocol, the downstream blocks could have a major effect on the upstream blocks. While the flow is always from left to right, the chain of influence could work in the opposite direction!

In the example in Figure 3-14, the robot (resource unit) fails. This results first in the **Checking Service** block can't accept anything. Afterward, the entities pile up in the **Packing Delay** and once that's full, in the **UnprocessedOrders Queue**. A cursory left to right follow up of the entities may not show the accumulation of entities in the **UnprocessedOrders Queue** is because of a block two step down the road.

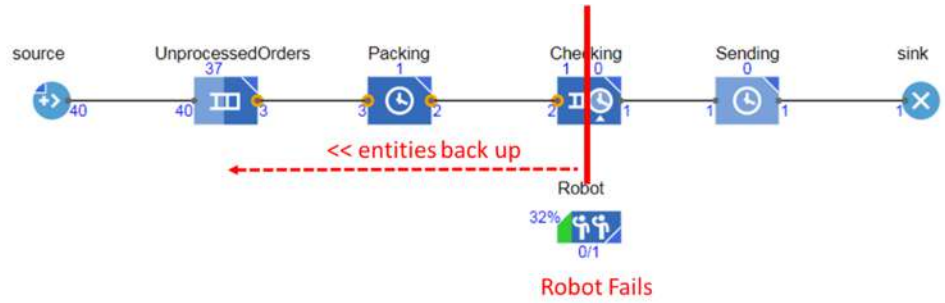


Figure 3-14: The resource unavailability forces the entities to back up in the preceding blocks

To better explain the implications of the pull protocol, we can review six scenarios as shown in Figure 3-16, which all entail a system whose flowchart consist of: **Source** -> **Queue** -> **Delay** -> **Delay** -> **Sink**. Furthermore, the **Source** block follows the push protocol with interarrival time of 1 second, the first and second **Delay** have a delay time of 5 and 10 seconds respectively. The screenshot is taken when the model is finished after 100 seconds.

- **Scenario 1**: All three blocks have maximum capacities and all entities that enter the process are served immediately. There are no blocked "*in*" ports and the only time the entities spend in the process is the result of a delay block.
- **Scenario 2**: The first delay (**delay_2A**) has a capacity of 1. When an entity enters this delay, nothing else can enter for 5 seconds. This blocks its "*in*" port and causes incoming entities to pile up in the preceding queue block.
- **Scenario 3**: The second delay (**delay_3B**) has a capacity of 1. Whenever an entity enters this delay, nothing else can enter for 10 seconds. This blocks its "*in*" port and causes incoming entities to pill up in its preceding block (the first delay). If you compare SC2 and SC3, you'll realize that the bottleneck has shifted from the first delay to the second.
- **Scenario 4**: The first and second delay have a capacity of 1. The blockages will change in this process at different times. At the moment we took this screenshot (Figure 3-15), an entity in the second delay was blocking the ready entity in the first delay. This (ready but waiting) entity prevents other entities from entering the first delay and causes them to pile up in the queue.

  This isn't a permanent state; during the simulation there are times the "*in*" port of second delay remains blocked while the first delay's "*out*" port isn't blocked (SC 4 in Figure 3-16). This is due to the fact there's a phase difference between the delay time of **delay_4A** and **delay_4B**.

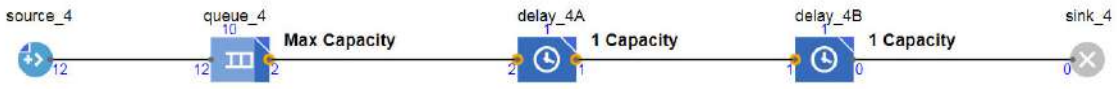


Figure 3-15: Scenario 4 part-way through the run, with two bottlenecks

Technically the overall accumulation of the entities at the queue that happens over the simulation run time is contributed by both the first and second delay (only at times when it blocks the first delay). In other words, the main bottleneck is dynamic and can change over time.

- **Scenario 5**: The queue has a capacity of 1 and the delays have max capacity. Since the delay after the queue has an infinite capacity, it can immediately take any incoming entity. In addition, since the second delay has infinite capacity, it isn't influencing the preceding blocks.
- **Scenario 6**: The queue and the second delay have a capacity of 1. The second delay's limitation blocks its "*in*" port whenever an entity is inside it. This blocks the first delay's "*out*" port and causes entities to pile up in the first delay. In other words, the fact the first delay has infinite capacity causes entities to pile up in there like the queue since the seceding block doesn't let them in. If the first delay had a limited capacity (such as in SC 4), entities would accumulate in a block upstream (the queue in this example).

Figure 3-16: Testing different scenarios in a pull system (except the Source which follows push)

Bottleneck detection in a system

To fully detect a bottleneck, it's useful to have a practical definition as it relates to process-centric models. The following definition of a "bottleneck block" is made for practical reasons so we can have a meaningful way to analyze our processes.

> *The momentary "bottleneck block" is the block that stops the flow of entities in one or more preceding blocks. This stoppage is apart from the natural delay in the flow from the behavior of blocks with built-in delays. It is important to realize in this definition, the bottleneck is identified at a snapshot of the process and not throughout its operation. In other words, the bottleneck might move from a block to another throughout the simulation, but at any moment we can identify the block that's the main source of blockage in the flow system. There are scenarios in which no*

unintended stoppage will be identified throughout the process execution. This means the time the entity spends in the process is only related to the specified delay times and therefore the dominant (average) bottleneck is the block that has the highest (expected) delay time.

Note this is an exploratory definition in the context of a simulation flowchart and may not be well-matched with definitions of bottleneck in other resources.

Having this definition is useful in figuring out the problematic components that if improved would have the most significant impact on the system's flow. Knowing this, we can review a process and visually identify the bottleneck block at any moment (specifically, in a paused model).

To standardize the bottleneck block detection, we first need to know if the system is following the pull or push protocol:

Push system: In this scenario, all the blocks that can follow the push protocol should be set to do so (explained further in the next section). If a bottleneck exists, running these scenarios will result in the model stopping with an error. If you review the output, you'll find the block that caused the error comes from an "*out*" port which is colored in red (**source_7** and **delay_8A** in the example models of Figure 3-17). The momentary bottleneck block is the one immediately after the block with the red "*out*" port (queue_7 and **delay_8B** in Figure 3-17).

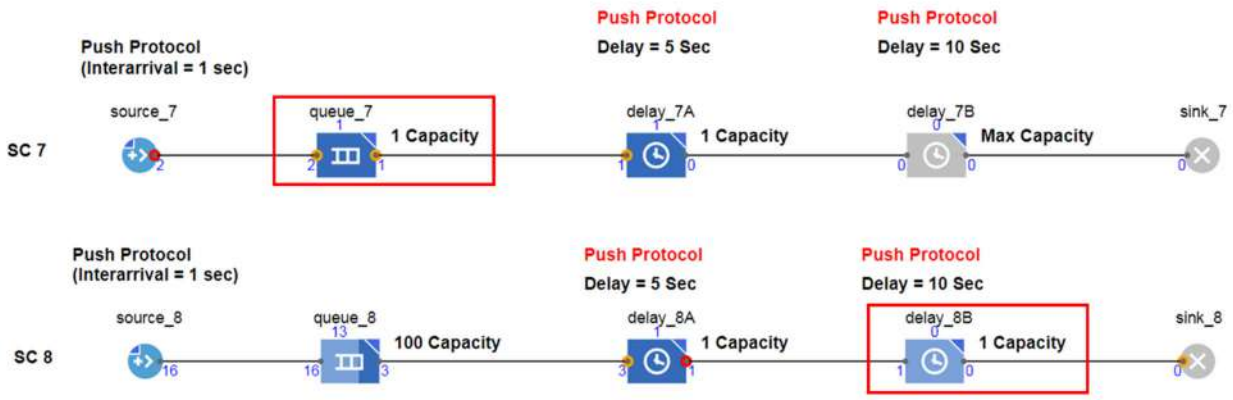


Figure 3-17: Detecting the bottleneck block in push systems

With a push system in these scenarios, we don't have to pause the model. When a block becomes a bottleneck, the model automatically stops and triggers the error. However, since the model won't continue, you only see the first block that becomes the bottleneck.

For further analysis, you can improve the bottleneck's performance (adding resources, reducing the delay time, increasing capacity) and run the model to find if the bottleneck moves to a different block. If the model ran without a problem, it means none of the entities spent any extra time in the process on top of what was necessary. As mentioned in our definition of a bottleneck block, in these scenarios the block with the highest expected delay time would be the average bottleneck (that is, over time, not just in snapshots).

Pull system (with the exception of Source and Enter blocks): Since AnyLogic's default behavior is to follow the push protocol for the **Source** and **Enter** blocks and pull for all the other PML blocks, you don't have to change any default settings to see the bottleneck occur in pull systems.

In a pull system, the bottleneck block could appear and move at different points in time. This means you must pause the model at a desired time to explore the bottleneck block at that moment.

In a paused model, if there's a pair of consecutive out and in-ports colored orange (indicating a ready entity at the "*out*" port which the blocked in-port is stopping), then the channel or link between these two ports is blocked. In Figure 3-18, AnyLogic displays these links in orange to highlight them (AnyLogic doesn't change the color of the links). If you identify multiple blocked links (for example, SC4 in Figure 3-18), you should focus on the one furthest to the right (for SC4, the line between **delay_4A** and **delay 4B**). After identifying the blocked link (or furthest to the right link in case there were more than one), the momentary bottleneck would be the block on the right (downstream) of the blocked link.

The red outlined blocks in Figure 3-18 (**delay_2A**, **delay_3B**, **delay4_B**, and **delay_6B**) are the bottleneck blocks. Notice that in SC4 there are two blocked links (**queue_4** to **delay_3A** and **delay_4A** to **delay_4B**). In this case, the link from **delay_4A** to **delay_4B** (far most to the right) is the main source of blockage and **delay_4B** is the bottleneck block.

Figure 3-18: Detecting the bottleneck block in pull systems (model status at time = 101)

As mentioned, bottleneck blocks are temporary and dynamically change. To identify blocks that slow the flow, you must examine the entire process throughout its full operation. Like push systems, if the system doesn't identify any blockages, the average bottleneck is the block with the highest expected delay time.

The above-mentioned bottleneck detection mechanism is an elementary and explorative way to identify bottlenecks. There are sophisticated methods for detecting momentary and average bottlenecks. Aside being simplistic and exploratory, it's important to note:

> *The above-mentioned "bottleneck block" detection is applicable to one uninterrupted flowchart. In most realistic scenarios, a complex flow system will be separated into multiple flowcharts working in a series, in parallel, or in a combination of both. Entities are moving between these*

flowcharts based on some rules that usually are more complicated than a single possible destination by a direct link. In these cases, the abovementioned bottleneck detection procedure may or may not be applicable to the entire flow system. However, it could be useful to evaluate each sub-flowchart in isolation.

The relationship of different PML blocks to push and pull protocol

Some transitional blocks in the PML do not support push or pull protocol. A few, such as **SelectOutput**, **Split**, **Hold** and **Release**, can't hold entities. Entities pass through them in zero model time and whether they do so is dependent on the blocks around them. When trying to determine the behavior of entities going through a passive block, predict their behavior as if the block wasn't there.

Other transitional blocks that can hold entities don't have any innerworkings to have push or pull protocol. For example, **Queue** can hold entities but not by its own mechanism; things accumulate in a queue because of successive blocks. In other words, an entity leaves the queue whenever the later block allows it in regardless of the pull or push method.

The pull protocol is used in blocks that can *actively store the entity*; this is used in case the receiving block isn't ready and needs time before it can accept the incoming entity. For the PML blocks that by default follow the pull protocol (the eight shown in Figure 3-19: **Delay**, **Pickup**, **Dropoff**, **Seize**, **Batch**, **Unbatch** and **Combine**), they also provide push protocol as an option. These blocks all have a checkbox in their **Properties** window's **Advanced** section that lets you select **forced pushing** as their behavior. Selecting this checkbox will enable the push protocol for that block.

Figure 3-19: PML blocks that by default are set to pull protocol but can also support push

The **Source** and **Enter** block are the only blocks that default to the push protocol. In the case of **Source**, you'll it is usually necessary to set a push protocol since any blocked arrival will disturb the set definition of arrivals. For example, if the arrival rate is set to one per second but the succeeding block doesn't let some of the entities depart, the arrival rate is no longer one per second. This means an error message will inform you the arrivals can't follow the specified pattern.

If you set **Source** to follow the pull protocol, you must set what it should do with excess arrivals. As shown in Figure 3-20, there are two options which state that agents which can't exit:

- **are destroyed**: the excess entities are deleted. This disturbs the total number of arrivals and the specified arrival rate won't be followed.

- **wait in this block**: the entities that can't exit wait in an internal hidden queue until the successive block allows them in. The total number of arrivals remains intact, but the arrival times don't follow the original setting.

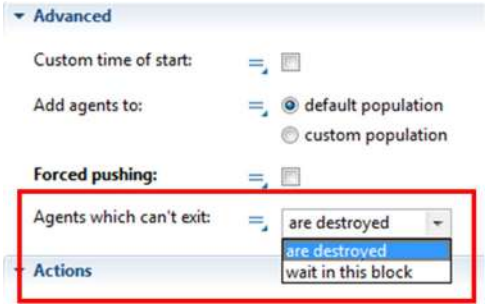


Figure 3-20: Setting for agents (entities) that can't exit the Source in the pull protocol

It's important to note these two options only apply to **Source**. If you set an **Enter** block to pull protocol (by clearing the **Forced pushing** checkbox), you can only set it to destroy the entities it can't take in.

Service is a specific case since under-the-hood; it's simply a combination of **Seize**, **Delay**, and **Release**. Both **Seize** and **Delay** have their own **forced pushing** setting; however, that's not visible from the **Service** block properties and they will follow the pull protocol. If it's important to enable the push option, you should use the three blocks explicitly.

Checking the status of in/out-ports [Optional]

The optionality of this section comes from the discussion of two functions designed for internal AnyLogic use, but it may be useful in certain cases. If at any time during the simulation, you want to check an in/out-port's status, you have the following options:

- **out-port of a predecessor block**: an *out* port is either idle or ready for an entity to exit from it. You can check the status of an *out* port by calling its `isReadyToExit()` method. This method returns true if there's an entity that's trying to depart, but blocked by the successive block ("*out*" port **Block_A** in bottom of Figure 3-21); otherwise, it returns false ("out" port of **Block_A** in top of Figure 3-21).
- **in-port of successor block**: an in-port is either idle or being blocked. You can check an "in" port's status by calling its `isCannotAccept()` method. This method returns true if the block is full and its in port is blocked ("in" port of **Block_B** in bottom of Figure 3-21). If not, it returns false ("in" port of **Block_B** in top of Figure 3-21).

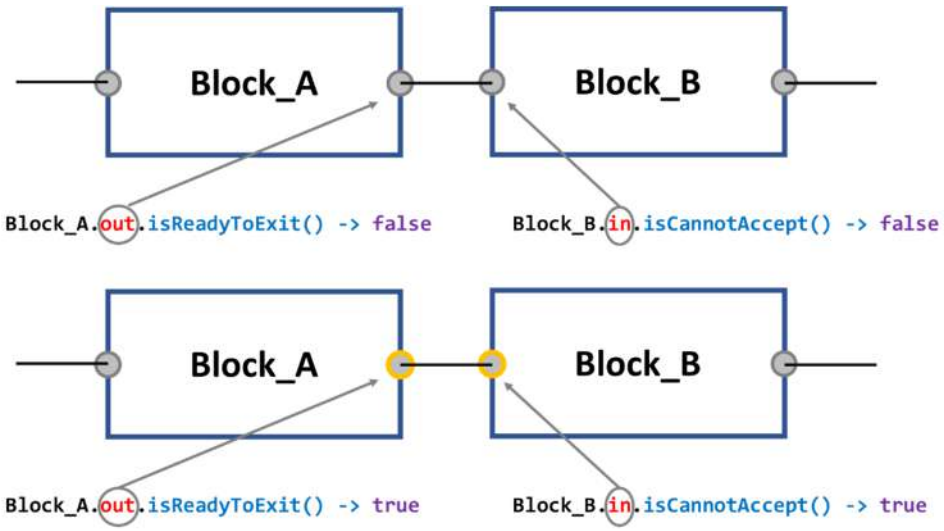


Figure 3-21: Checking the status of an "in" or "out" port

Be careful not to tie your model logic with the aforementioned methods. In many cases, the outputs of these functions change in zero model time. The order of events is important and sometimes hard to predict; you should avoid them unless you design your model to account for this. These methods are more reliable when the in/out-port blockages persist for an extended period.

Discussion on how SelectOutput and SelectOutput5 may result in counterintuitive behavior in combination with pull protocol [Optional]

In a pull system, downstream blocks can interfere with the flow of upstream blocks by blocking flow channels. We reviewed several examples of this in the **BACKWARD RIPPLE EFFECT OF PULL PROTOCOL** section. This is similar, but we focus on some counter-intuitive behavior from the **SelectOutput** and **SelectOutput5** blocks.

Let's compare two simple flowcharts where a **SelectOutput** block separates the flow into two streams. In the example (shown in Figure 3-22), we make the following assumptions:

- Model unit time is in seconds
- In the **On Startup** field of **Main** we inject 10000 entities to each of the flowcharts:
    - `source_1.inject(10000);`
    - `source_2.inject(10000);`
- Model stop time is set to **never**; instead of setting a pre-defined stop time, the model will stop when either `queue1.size()` or `queue2.size()` reaches zero, whichever comes first. This is the model with the code that's added to the **On enter** of **sink_1** and **sink_2** (Figure 3-32).

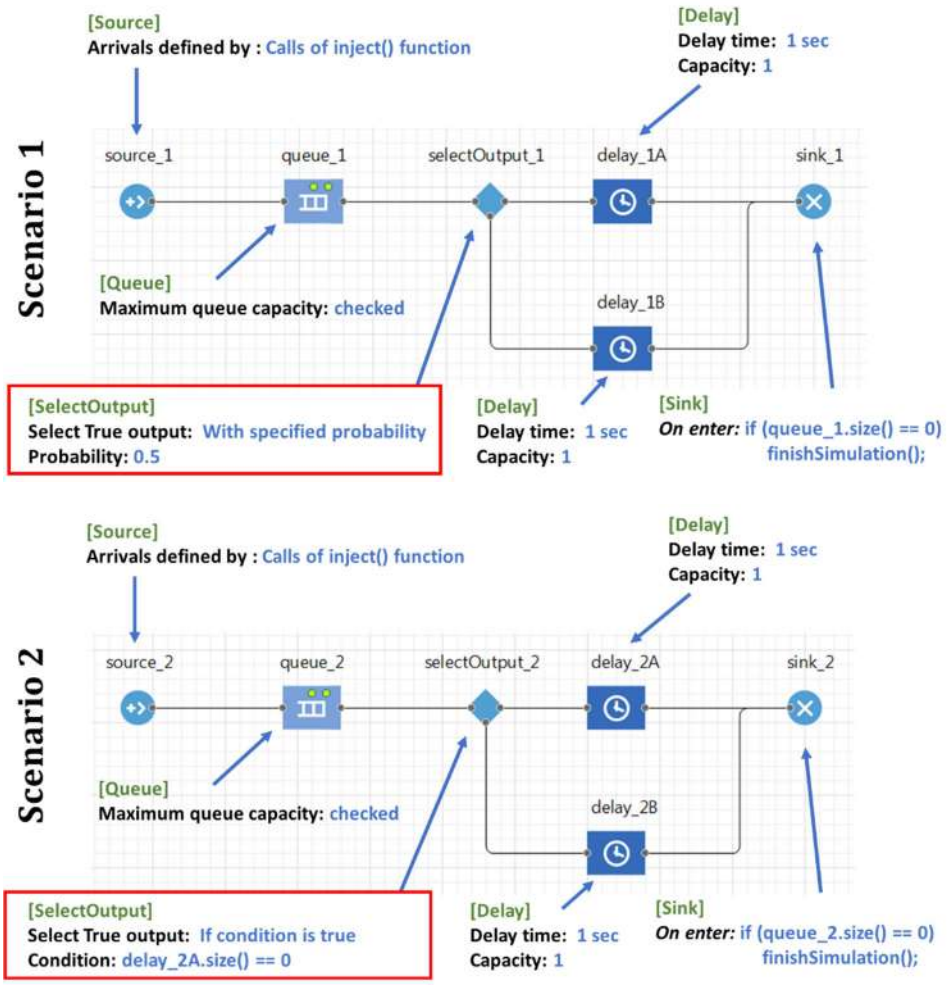


Figure 3-22: Scenario 1 & 2, the only difference is in the SelectOutput block's "Select True output" setting

This model has two nearly-identical processes (Scenario 1 & 2) with the difference in how the **SelectOutput** block passes the entity out the true port. Each scenario starts with 10000 entities in their queue and the two delays work to process them. Think of it as a competition between the two scenarios with two players on each team; the model which flushes all 10000 entities from its queue wins.

The following two rules are set for selection of true port in each scenario:

- In Scenario 1, **SelectOutput_1** uses a probability to select the true port. Behind the scenes, AnyLogic checks the condition based on the result of a `randomTrue(p)` function. This function randomly picks a number from a uniform distribution between 0 to 1 and if the number is less than p, returns true. This means the probability of returning true is p and the probability of false is 1 - p. Since the probability is set to 0.5, AnyLogic executes `randomTrue(0.5)`, returning true 50% of the time and false 50% of the time.
- In Scenario 2, **SelectOutput_2** selects the true port with a condition based on the size of (number of entities inside) the **delay_2A** block. If that block is empty, `delay_2A.size()` would return 0. Thus, the condition would be true (`0 == 0`) and the entity would exit from the true port, being sent to **delay_1A**. Had there been anything inside **delay_2A**, the entity would be sent to **delay_2B**.

In other words: if nothing is inside **delay_2A**, the entity gets sent to it, otherwise it gets sent to **delay_2B**.

If you run the model (Figure 3-23), you'll see Scenario 2's process flushed out all the entities faster than Scenario 1. At the time Scenario 2 processed all 10,000 entities, there were still 2,496 entities left in Scenario 1's queue (**queue_1**). In both scenarios, the number of entities in each branch is roughly equal (Scenario 1 is probabilistic and is offset by a reasonable amount). In both scenarios, the entity can't move forward if the downstream blocks are full.

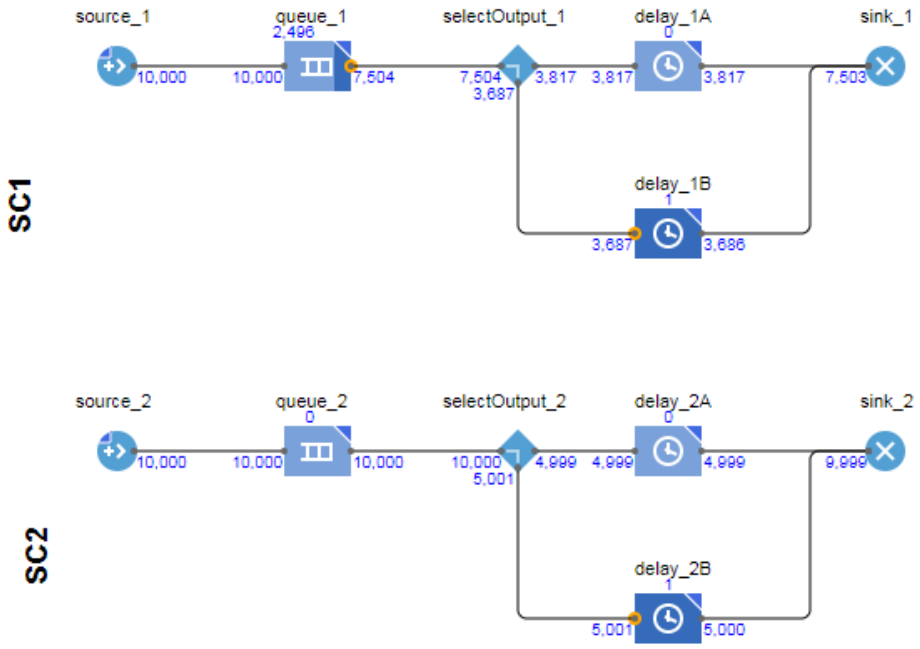


Figure 3-23: Model output, in scenario 2 entities get processes faster

Although these scenarios may look similar on the surface, the pull protocol makes their throughput significantly different. To understand the reason for this discrepancy, let's examine the underlying mechanisms. We explain Scenario 2 first since it is easier to understand.

Scenario 2: At the beginning of the model, when the first entity reaches the head of the queue and is ready to depart (time = 0), AnyLogic checks the condition (`delay_2A.size() == 0`). If the condition is true, it assigns the entity to the true-port, otherwise the false-port. Remember this is just the assignment; it doesn't mean the entity can leave the port. If both downstream processes are blocked, the entity must wait in the queue. Said another way, this setup results in the downstream blocks (**delay_2A** and **delay_2B**) to pull the ready entity if they're ready (with **delay_2A** having the precedence).

Scenario 1: This scenario's underlying mechanism is like Scenario 2, but the probabilistic condition results in a behavior you should be aware of. Whenever an entity is ready to depart, AnyLogic checks the condition (`randomTrue(0.5)`). If the condition is true, the entity is assigned to the true-port, otherwise it's assigned to the false-port. This condition is random and has nothing to do with the availability of the downstream blocks. Once the port is assigned, the entity will wait until the block following the selected port is open. This means there's a chance we'll assign the entity to a blocked port, possibly several times

in a row, while the other delay remains empty. If this entity waits for the blocked successive block to be available, it also prevents other entities from leaving the queue. These added wait times explain why Scenario 1's throughput is less than Scenario 2.

In Scenario 2 we assigned the entities based on the availability of the downstream blocks. If there's immediate availability, the entity can leave the queue. In Scenario 1, we assigned the "out" port randomly, so we might have assigned a busy downstream block to the entity. Assigning a blocked port (port of a downstream block) forces the entity to wait until the assigned blocks are available.

When you use the **SelectOutput** and its big brother **SelectOutput5**, keep the following rule of thumb in mind:

> *If SelectOutput or SelectOutput5 uses probabilities to assign the "out" port and your process is following a pull protocol (AnyLogic's default), consider the SelectOutput or SelectOutput5 as an added constraint on the process' throughput. In other words, you're forcing each port's output to follow a proportion and for AnyLogic to enforce the set proportions it puts some constrains on the entities flowing through SelectOutput or SelectOutput5. This added constraint, which makes sure your proportions are implemented, results in less entities to pass compared to a scenario where any available downstream block can pull an entity. Either use condition(s) for selecting of the "out" ports or know using probabilities is a logical constraint that effects the flow of entities.*

As mentioned above, **SelectOutput5** follows a similar logic as **SelectOutput** and the abovementioned discussion is applicate to **SelectOutput5** too. For a more comprehensive understanding of **SelectOutput5** inner working, please refer to **MODEL BUILDING BLOCKS – LEVEL 1 APPENDIX**.

Example Model 2: Earthmoving operation

Prerequisites:

Model Building Blocks (Level 1): **Source**, **Queue**, **Delay**, **Sink**, **Seize**, **Release**, **Service**, and **ResourcePool**.

Math: Random variable, Random variate, [Poisson process]

Learning Objectives:

1. Identification of entities and resources
2. Set the model stop time based on a condition(s)
3. Define multiple Simulation experiment for one model

Problem statement

Our objective is to model an earth-moving operation that uses a single wheel loader and five trucks with 15 m^3 capacity to move 990 m^3 of soil (Figure 3-24). The following activities make up the operation:

1. A single wheel loader (**Load**) at the loading site fills the trucks on a first-come first-served basis.
2. Full trucks haul the soil to the dumping location (**Haul**).
3. It takes time to fully empty the truck (**Dump**). Several trucks can dump their load at the same time), which can take place immediately after hauling.
4. Empty trucks return to the wheel loader for another load (**GoBack**).
5. The operation repeats the previous four steps until the trucks move all the soil. Specifically, we can calculate how many cycles must occur: $\frac{990}{15} = 66$.

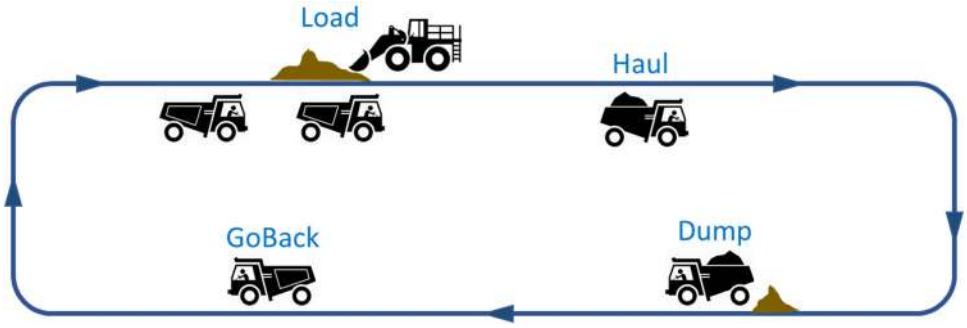
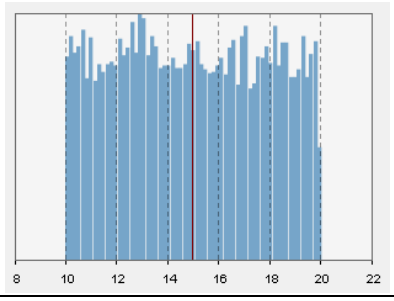
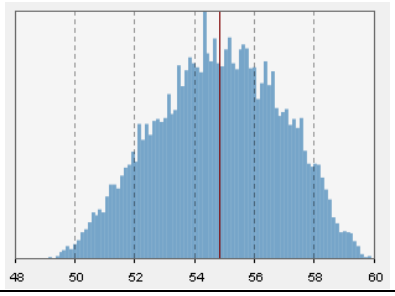
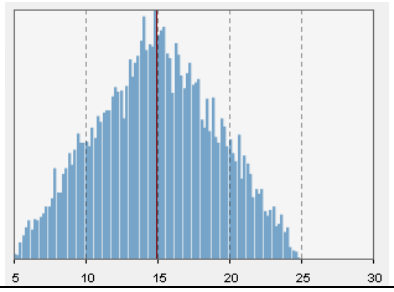
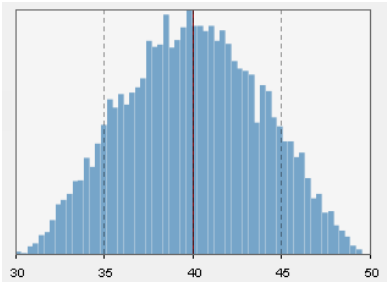


Figure 3-24: Cycle of earthmoving operation

Each activity in the cycle (Load, Haul, Dump, and GoBack) takes time, which we define by random variables with probability distributions. Based on the historical data, Table 3-3 shows appropriate distributions for these activities.

Table 3-3: Sampling distributions used for Load, Haul, Dump, and GoBack activities

Load	**Haul**
Uniform with min of 10 and max of 20 minutes: `uniform(10, 20)` 	PERT with min of 40, max of 60 and mode of 55 minutes: `pert(40, 60, 55)` 
Dump	**GoBack**
Triangular with min of 5, max of 25 and mode of 15 minutes: `triangular(5, 25, 15)` 	PERT with min of 30, max of 50 and mode of 40 minutes: `pert(30, 50, 40)` 

The shapes in Table 3-3 are sample histograms generated from many random draws based on the provided settings. They give you a general idea of the distribution and are not an exact match of what the outcome will be. This means it's best to view them for their overall shape.

Examining the model design

The first things we must identify in a process centric model are entities and resources – and in most, the choice of entities and resources is clear. However, we selected a model that offers options. The two main alternatives we can use:

- We make the entities moving in the system be the trucks and the wheel loader as the resource which helps them through the process. In this case, we have five entities (trucks) that continuously cycle through the process and stop after they move the stockpile. This design is a bit different from the more common flow systems in which entities pass through the process only once. This design is like the finite-source queue example discussed in Chapter 2.
- The entities are the soil loads and the resources are the trucks and the wheel loader, which help the loads to move. In this case, we know from the previous calculation we'll have 66 entities (soil loads) that only pass through the process one time.

The instructions below show how to build design 1 and then how to build design 2.

Design 1: Trucks are entities, wheel loader is the resource

1. Create a new model, setting the time units to minutes.
2. Build the flowchart shown in Figure 3-25 by dragging and dropping the needed blocks from Process Modeling Library (PML) palette.

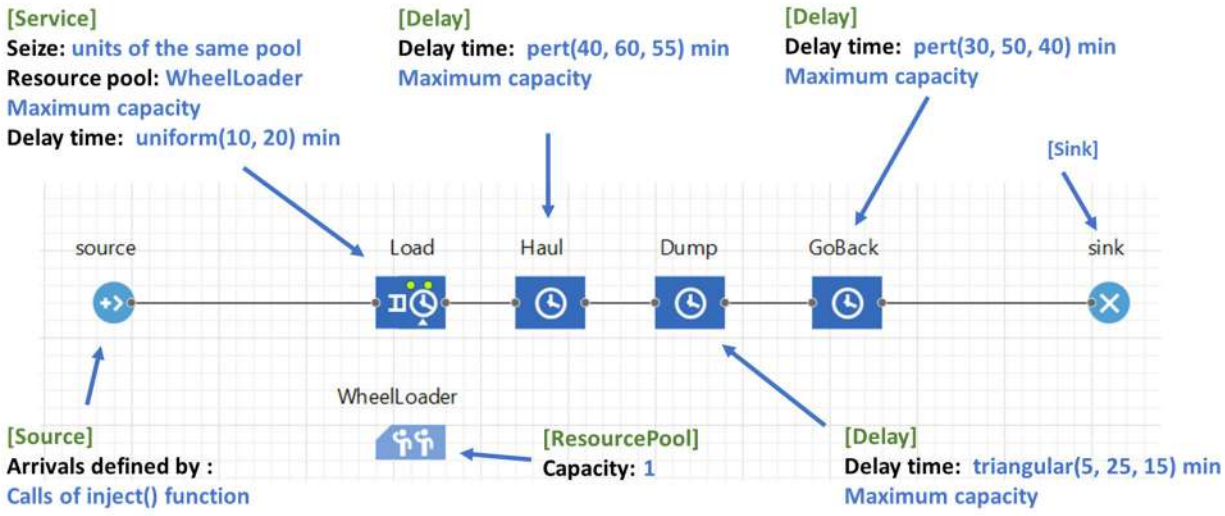


Figure 3-25: Flowchart of earthmoving operation (Design 1)

In the flowchart you made:

- **source**: arrivals are defined manually. At the start of simulation (initialization), we must have five trucks ready to start at the **Load** block (inside its embedded queue). This means we can't use an arrival rate or interarrival time. We must manually inject five trucks into the process when the model starts.
- **Load**: A **Service** block that seizes the wheel loader during the loading step. We set the capacity of the embedded queue inside this seize block to maximum to remove that constraint. This allows us to have an unlimited number of trucks waiting in front of the wheel loader if necessary.
- **Haul**, **Dump**, and **GoBack** are **Delay** blocks with maximum capacity (no constraint).

If you run the simulation experiment, the model will progress in time. However, the flowchart won't have any entities passing through it. If we want to match the model's description and ensure the operation starts with five trucks, we must inject five trucks at the start of the model.

3. Open the properties of **Main** (either click an empty area in the graphical editor or click **Main** under the **Projects** view) and, as shown in Figure 3-26, type the following in the field of **On startup**: `source.inject(5);`

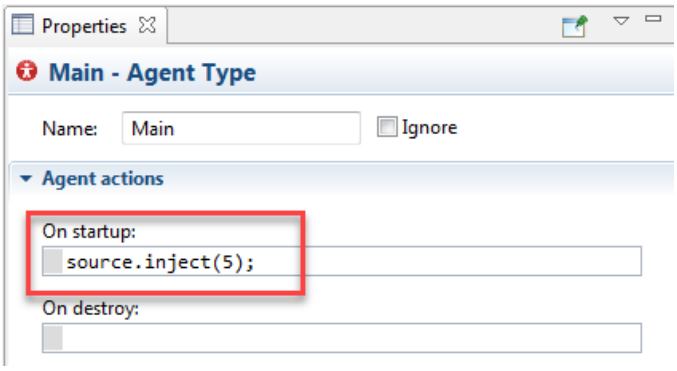


Figure 3-26: injecting five entities into the process on startup of the model

4. Run the model, monitor and manually pause it when all five trucks leave the flowchart (Figure 3-27). Since the model isn't finished, if you don't pause, it will continue to move forward. This will reduce and utilization of the wheel loader, as it will be idle (since there are no more trucks to load). However, we want the trucks to go back and complete 66 cycles; currently each truck hauls one load and leaves.

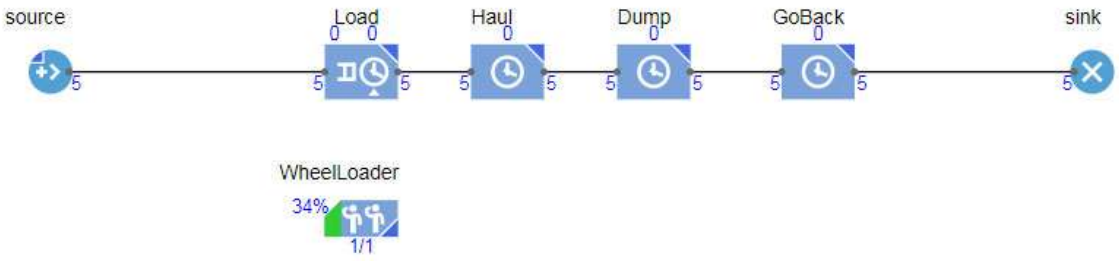


Figure 3-27: Result of simulation run (your result might be different based on the time you've pause the model)

5. Like the finite-source queues (M/M/c/∞/d) shown in the queueing model section, this model has a finite number of entities (trucks) that circulate until they dump 66 cycles. This means we must add a **SelectOutput** to decide to let the trucks leave or to loop them back to the start.

   To track of number of finished cycles, we'll add an integer variable as a counter and increment it each time a truck goes to get another load (enters the **Load** block). Each time a truck finishes the **GoBack** task, it checks the current number of complete cycles. If there are less than 66 complete cycles, the truck will return to the **Load** task. If there are 66 complete cycles, the truck will leave the model: there's nothing left to move.

   In addition, we'll add some code to stop the model after all five trucks have left. To implement this logic, you must modify the model as shown in Figure 3-28 below.

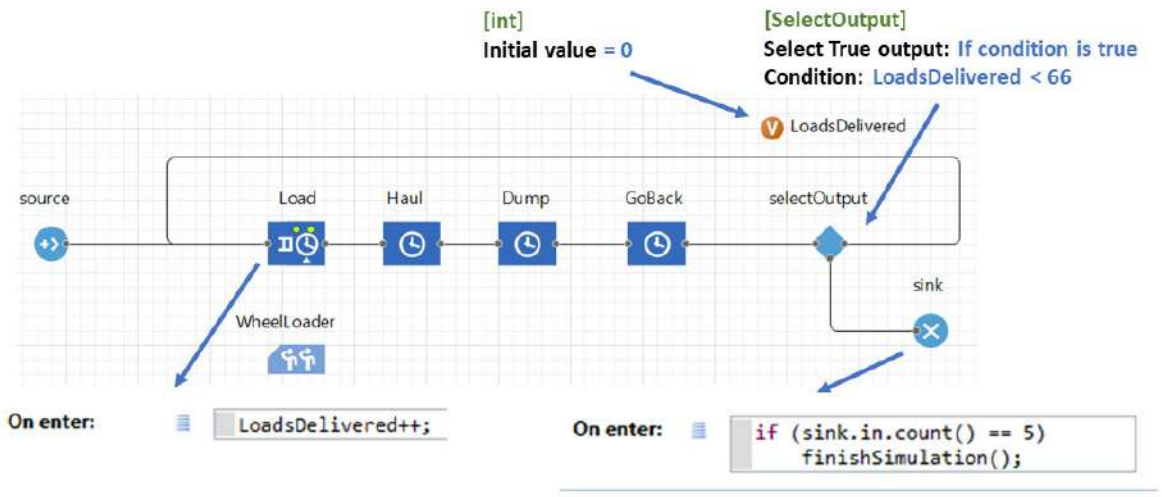


Figure 3-28: Completed flowchart of earthmoving operation

Each time an entity enters the sink block, the code at its **On enter** field checks the number of entities that have entered the sink so far. Like any other port, the in-port of the **Sink** block has a built-in `count()` function (Figure 3-29).

As an aside: the **Sink** block and the **Source** block have an identical `count()` function. This means we could have used the `count()` method directly for the **Sink** block and our condition would have been `sink.count() == 5`. To make it clear that any port has this function (and since it only requires us to type three extra characters), we'll use the `count()` function on the port. This condition compares the number of entities that entered the sink (left the process) to the number five. If this condition is true (in other words, if all five trucks left the operation), it ends the simulation.

Figure 3-29: "in" port of the Sink block

In most terminating simulation models, we prespecify the end time explicitly. However, we did not change the default setting for the simulation experiment, which has its stop time set to **never**. In this specific example, we do not know the stop time of the simulation since a condition to trigger the end. That's why we used the `finishSimulation()` function to achieve a similar outcome without having to set an explicit stop time. This function doesn't destroy the model and it allows us to examine and analyze the results.

6. Run the model and observe the outputs (Figure 3-30).

   As shown, the utilization of the wheel loader is around 56% and the entire operation (delivering 66 soil loads) took 1,766 minutes. To see the time the model finished (end of the operation), click the **Toggle Developer** panel button in the model window's lower right corner.

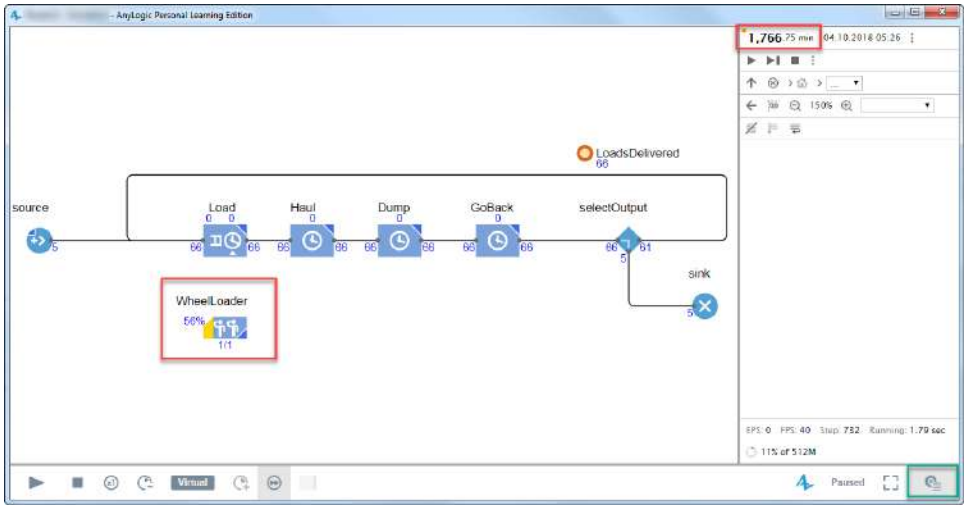


Figure 3-30: The output of Design 1 at the end simulation (66 loads have been delivered)

This is just one realization of a random process. This means you should expect different outputs if you run the model again. However, AnyLogic's Simulation experiment by default sets the random number generator to a fixed seed. This setting results in receiving the same exact result each time you run the model. If you want to observe the random nature of the operation, you must click the **Simulation: Main** experiment under the project's view and under the **Randomness** section of its properties, select the option for **random seed**.

Design 2: soil loads are entities, wheel loader and five trucks are resources

In our previous design, our trucks were the entities. The calling population was finite and entities cycled through the operation until it was done. In our second design, the entities will be the 66 soil loads. Each entity (soil load) will pass through the flowchart one time and leave at the end.

To implement the second design:

1. Create a new model and set the time units to minutes. As an alternative, you can modify the previous model. To keep the original model intact open the **File** menu and select **Save as…**, choosing a new name appropriate to this second design.
2. Build the flowchart shown in Figure 3-31 by dragging and dropping the needed blocks from Process Modeling Library (PML) palette:

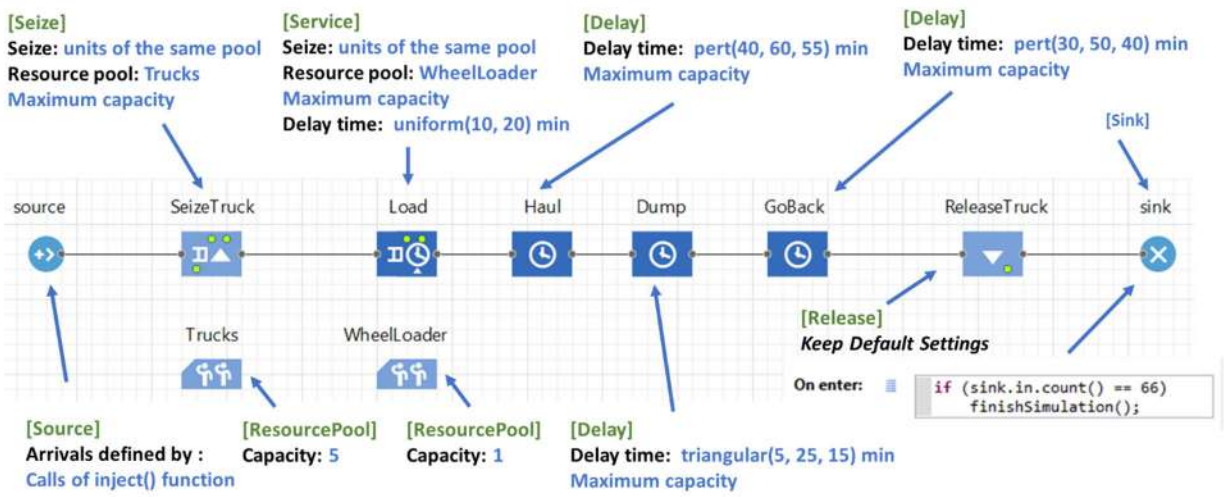


Figure 3-31: Flowchart of earthmoving operation (Design 2)

3. Like the first design, we must manually inject the entities ("loads" in this case) into the system. Open the properties of **Main** and in the **On startup** field, type: `source.inject(66);`
4. Run and observe the outputs.

As shown in Figure 3-32, the utilization of the wheel loader and the trucks is 58% and 97%, respectively. The operation took 1,739 minutes to deliver 66 soil loads. Again, this result is just one realization. If you set a random seed for this experiment will allow you to observe unique results out of each run.

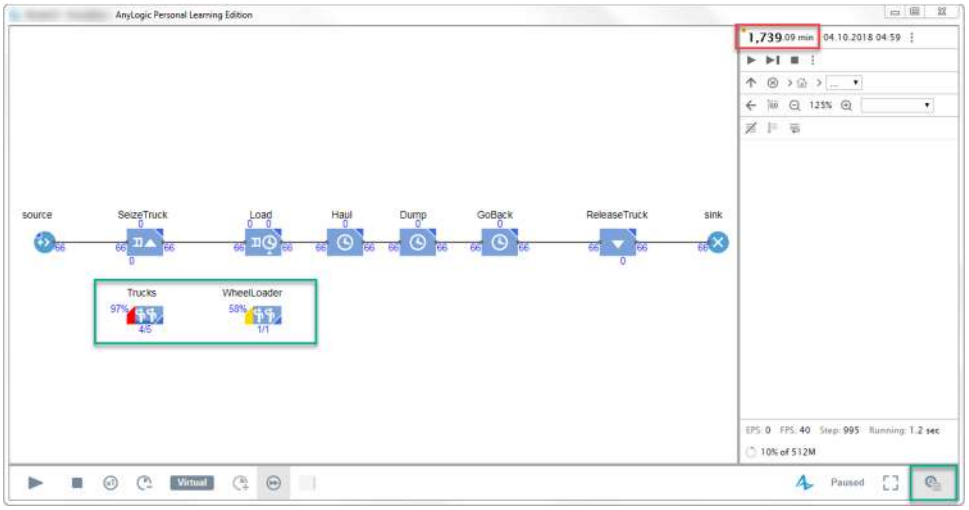


Figure 3-32: Outputs of Design 2, at the end simulation

The utilization of trucks will be *almost* 100% because the model's design ensures they remain busy until they deliver all the loads. The 3% idle time (100% of time minus the 97% utilization) reflects the increasing number of idle trucks as the trucks deliver the last five loads. After a truck delivers the 62nd load, the four trucks that follow it complete the remaining four loads. This means the first truck is idle, as shown in Figure 3-33 below. This continues until the last truck leaves and the simulation finishes.

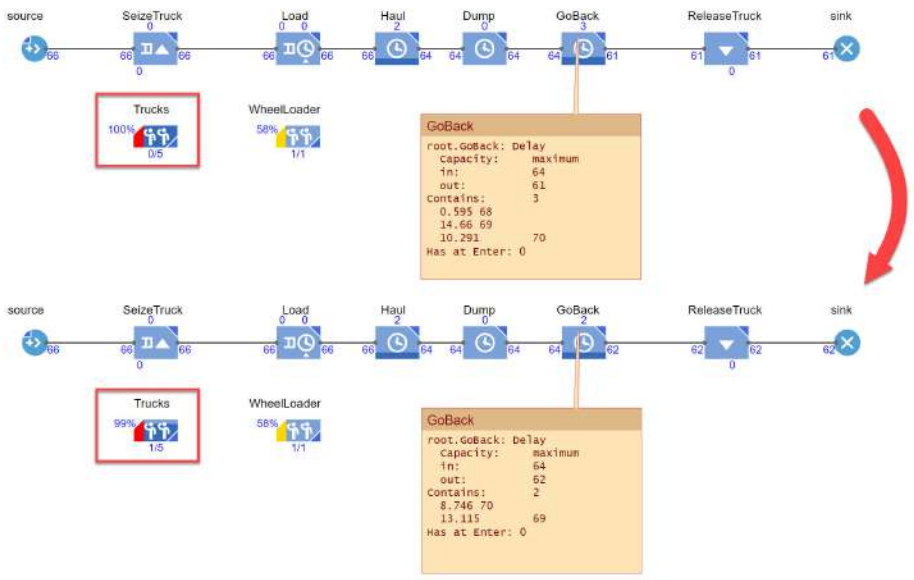


Figure 3-33: Transition from 100% utilization after the 62nd load is delivered

Now that we've built two versions of this model, it's a good time to think about what would happen if we changed the number of loads, trucks or wheel loaders. We can change these quantities and rerun the model. But each time you change the inputs, you'll lose the previous scenario's settings.

Parametrizing the model: parametrizing the number of soil loads, wheel loaders and trucks

Before we run the simulation with different input values and save our setup as a simulation experiment, we must parametrize the quantities we want to change and then set up a simulation experiment for each scenario. We can do this for both designs, but for brevity, the instructions below only show the modification of design 2.

1. Use the Agent palette to add two **Parameters** (Figure 3-34):
    - The first should be an integer (int) named **NTrucks**, with a default value of 5 and the label "Number of trucks".
    - The second should be an integer (int) named **NLoads** with a default value of 66 and the label "Number of loads".
2. In the **Trucks** resource pool, replace the hard-coded value of 5 with **NTrucks**.

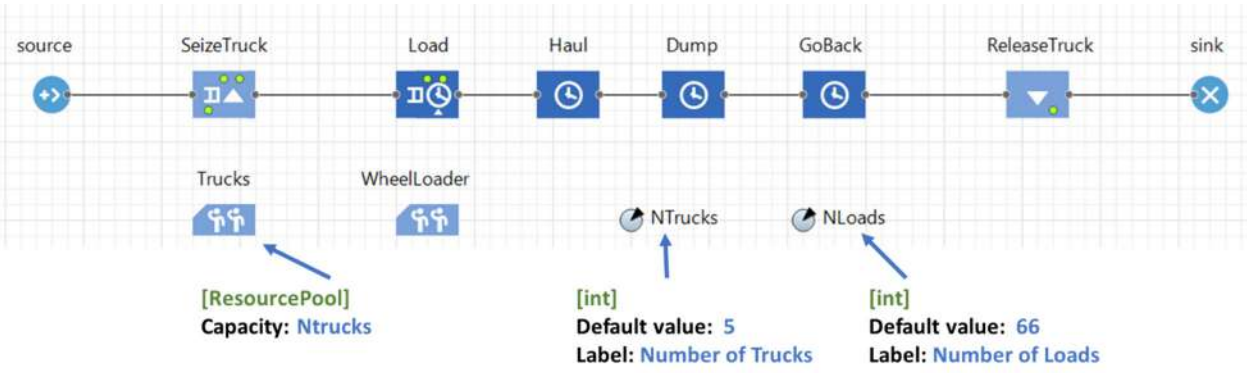


Figure 3-34: Parameterizing the capacity of Trucks resource pool

3. Click any empty space in the graphical editor of **Main** to see its **Properties** (alternatively, you can click the **Main** file from the Projects panel). Under the **Agent actions** area, in the **On startup** box, change the code to `source.inject(NLoads);` (Figure 3-35)

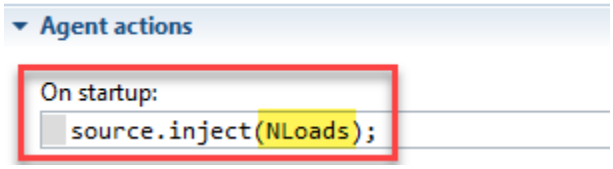


Figure 3-35: Parametrizing the number of loads injected into the flowchart at the initialization

If you're modifying Design 1: Note the parameterization setting in step 3 only works for Design 2. In design 1, `source.inject(5)` won't change, but you must delete and replace the **LoadsDelivered** by a parameter of type integer and initial value of 66. You also must attribute the simulation stop condition in the **sink** block with the **LoadsDelivered** parameter.

Now that you've parametrized the number of loads and the number of trucks, run the model. You'll notice the results are the same - the simulation uses the parameters' default values.

4. In the Projects view, click the **Simulation: Main** experiment and use the **Name** box to change its name to "BaselineScenario" (Figure 3-36).

   You can see the **Parameters** section shows the parameters we added. AnyLogic shows the parameters' labels instead of their name in experiment's **Properties** view. This is where using the labels defined in step 2 allows more readable identifiers for our parameters.

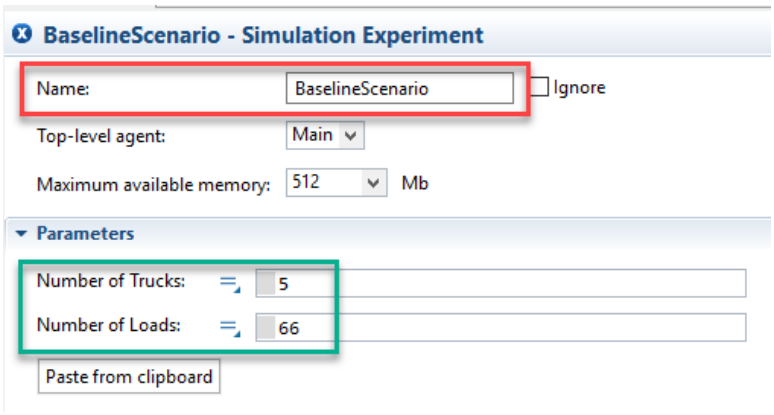


Figure 3-36: Parameters added to Main (top-level agent), available as inputs to the Simulation experiments

5. Do the following to build a new **Simulation** experiment with new parameter values (Figure 3-37):
    a. Right-click the model, select **New**, and select **Experiment**.
    b. In the **Experiment Type** area, click **Simulation**.
    c. In the **New Experiment** dialog box, in the **Name** box, enter "Scenario1".
    d. Click **Finish**.

    Figure 3-37 shows this new experiment copies its time settings from the "**BaselineScenario**" (our original **Simulation** experiment).

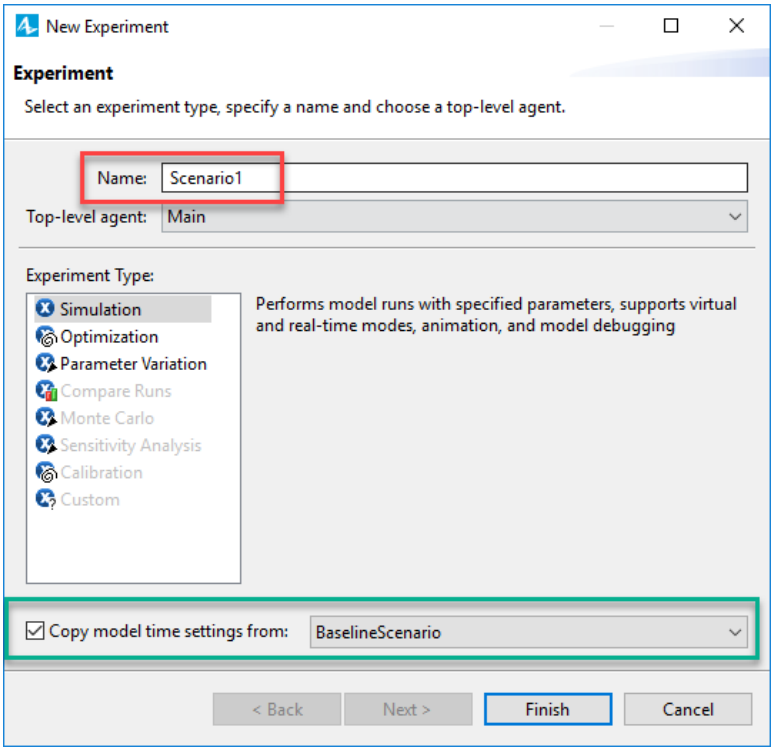


Figure 3-37: Setting up the second Simulation experiment

6. Under the Projects view, click the "**Scenario1**" experiment. In the **Parameters** section, change the value in the **Number of Trucks** box from 5 to 8 (Figure 3-38). Also, change its random number generation setting to **random seed**.

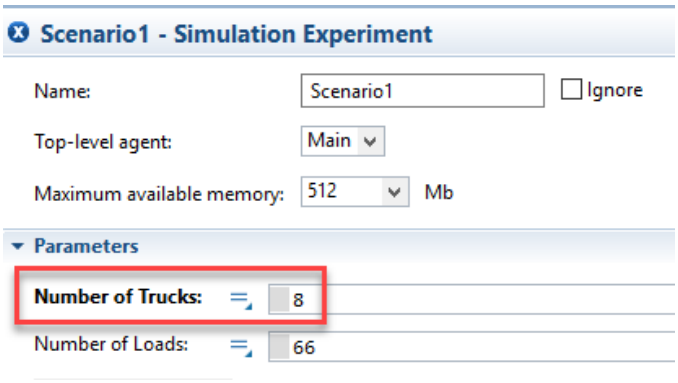


Figure 3-38: Changing the second simulation experiment's input parameter value

7. Do one of the following to run the second experiment:
   - Right-click the **Scenario1: Main** experiment and run it.
   - Click the arrow immediately to the right of the toolbar's **Run** icon and select Scenario1 (Figure 3-39).

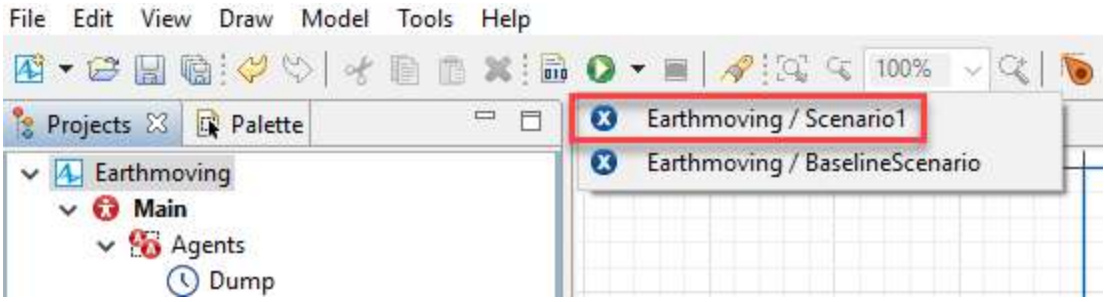


Figure 3-39: Running the experiment by the toolbar's Run icon

Figure 3-40 shows the result of this simulation run. Your outcome may differ since the simulation uses a random seed. As we expected, increasing the number of trucks increased the wheel loader's utilization and reduced the time it took to move the 66 soil loads.

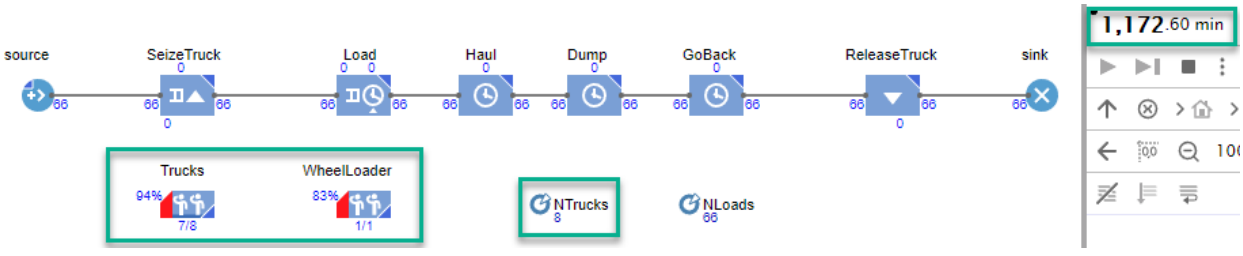


Figure 3-40: Output of the Scenario1 Simulation experiment

You can repeat this process as many times as you want to add simulation experiments and define alternative scenarios with different input parameter values. Parametrizing the model lets you have different scenarios, which are helpful for comparison.

As an exercise, build other scenarios with different values for the “Number of Trucks” and “Number of Loads” and try to explain the outcomes. You can also add more parameters to the model, such as the number of wheel loaders. It would be interesting to examine the operation finish time based on having more than one-wheel loader.

Technique 4: Using task priority

There are situations in which a resource pool is shared between two or several processes. Based on our previous discussions we know the **Seize** block is the fundamental flowchart block which associates tasks to resource units from one or several resource pools. This means a *shared* resource pool is indicative of a resource pool used by two (or more) **Seize** blocks (Figure 3-41).

AnyLogic makes it easy to implement these types of scenarios. You simply select the same **ResourcePool** block in the properties of the desired **Seize** blocks. AnyLogic doesn't require a physical link (a virtual representation of the connection) between the **ResourcePool** blocks and other blocks that use it, as it often does between other blocks. Therefore, you can assign a resource pool to as many **Seize** blocks as you want without any visual entanglement.

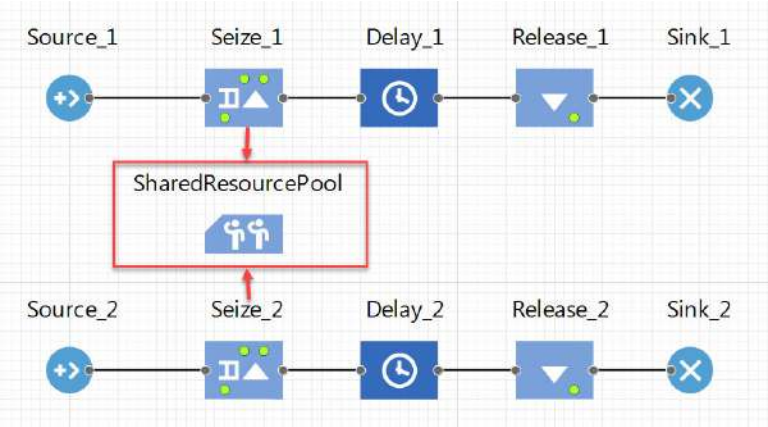


Figure 3-41: Two processes that share the same resource pool

When sharing resource pools, there's a logical implication that needs our attention. When resources are shared, it triggers competition among processes. In most real scenarios, tasks do not have the same priorities and some processes take precedence. **Seize** blocks have a feature called *task priority*, a mechanism that lets a resource pool discriminate between requests from multiple **Seize** blocks. Higher Seize block priorities result in more "attention" being paid to the block.

We need to mention other PML blocks with embedded Seize blocks (**Service**, **RackStore**, **RackPick**, and **Assembler**) also inherit the **task priority** setting from their embedded **Seize**. Our focus is on the **Seize** block, but these concepts also work in the other blocks with embedded Seize.

Before we describe how task priority works, we need to clarify some terms. When an entity enters a **Seize** block, the logical significance comes from the entity needing to complete a task that requires a resource unit. The numerical value set in the **Seize** block's **task priority** field is assigned to this incoming task to specify its priority in comparison to other tasks. After this assignment, a request for a resource unit is sent to the **ResourcePool** block which includes the priority value of the task.

If a resource unit is immediately available, the incoming task is assigned to that idle unit, regardless of the priority. However, requests that can't be immediately processed are queued at the **ResourcePool** block. It's important to note it's the request that's queued and not the entity itself. This request queue tracks the order of the requests along with the associated priority. Upon a new arrival, the request queue is

sorted by priority and secondly by when it arrived. When a resource unit becomes available, the request at the front (head) of the queue is fulfilled and its associated task is assigned to the resource unit.

To illustrate this point, Figure 3-42 shows an example with three Seize blocks (A, B, and C) with equal priority and therefore ordered purely based on their arrival time. The resource pool has one resource unit which is busy and won't be free until time 4 ("time" being a predefined unit, such as seconds). Assume that a new entity entered each one of the seize blocks sequentially in the first three time-units. In other words, an entity entered A at time 1, B at time 2, and finally C at time 3.

Since the resource pool is occupied and won't become available until time 4, these tasks can't be immediately assigned to any resource unit. This means three requests will be queued up in the resource pool. Task A came first and is sitting at the head of the request queue. Once it reaches time 4 and the resource unit becomes available, task A will be assigned to it (shown in the red dotted box in Figure 3-42 below).

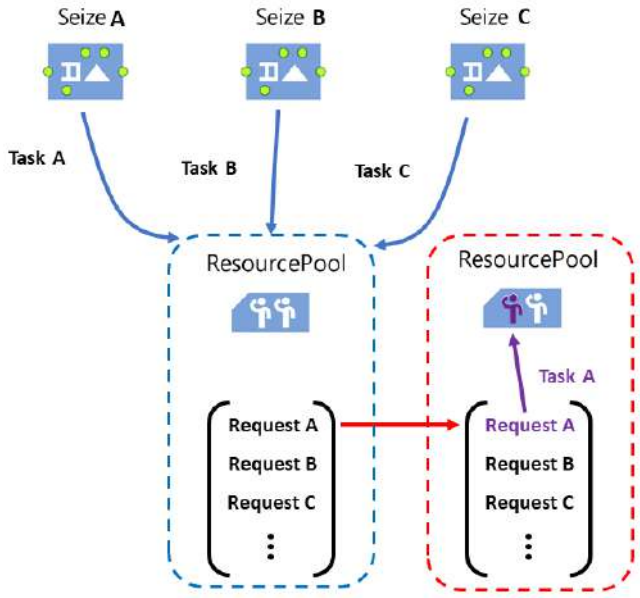


Figure 3-42: Resource requests queue, and assignment of a task to an available unit based on the first request

Since all tasks have equal priority, the request queue's queuing policy is first-in-first-out (FIFO). However, we can assign a priority value to each incoming task. Higher numeric values for a task (and its associated request) means it has a higher priority.

Revising our example, let's assume **Seize A** has a task priority of 0 (default), **Seize B** has a task priority of 5, and **Seize C** has a task priority of 10. At time 1, the first request (from **Seize A**) arrives. Since the only request is from **Seize A**, its task is first (at the head of queue). Then at time 2, request B arrives. If their priorities were the same, request B would have been placed behind request A. However, we set **Seize B**'s priority to 5. This means AnyLogic rearranges the queue and places request B at the head of the queue and request A at the end (tail).

Finally, at time 3, request C arrives with its grand priority of 10; after the queue's resorting, request C moves to the head. At the next moment (time 4), the resource unit becomes available and the request at the head of the queue (request C) is assigned to the unit (Figure 3-43).

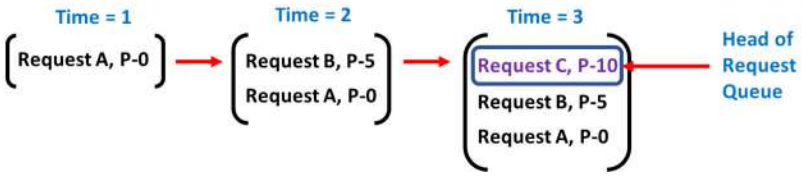


Figure 3-43: Sorting of the request queue based on incoming tasks (Different priorities and arrivals)

For a final revision, say **Seize C** has a priority of 5 (the same as **Seize B**). At time 3, request C will be inserted, and the queue will be rearranged. Since the primary sorting method is based on priority, request C will be placed higher than request A. In relation to request B however, this is where the secondary sorting method, time of arrival, comes into play. As request C came last, it will be placed after request B. At time 4, when the resource unit is available, it will first attend to request B. Afterward, it will attend to request C and then finally request A after that (Figure 3-44).

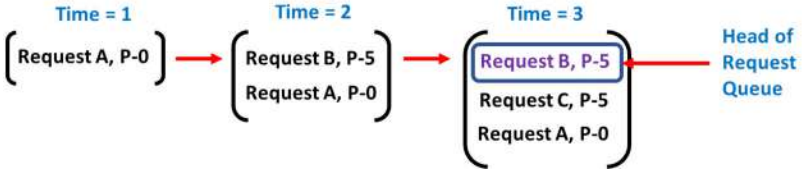


Figure 3-44: Sorting of the request queue based on incoming tasks (Similar priorities and different arrivals)

Resource assignment [to tasks] logic

Case 1: A resource is immediately available; the first task which sends a request will get the resource.

Case 2: There's no available resource and all incoming tasks (and their associated requests to the resource pool) have equal priority; they will be served based on first-come first-served bases, and time of request determines which task grabs the resource.

Case 3: There's no available resource and some (or all) of incoming tasks have different priorities. Incoming requests with higher priorities take precedence over lower priorities. Requests with equal priorities are chosen based on arrival time.

Each time a new request arrives, you can imagine the queue of requests will sort itself in two steps: first based on the request priorities, then based on arrival time for requests that have identical priorities (the actual sorting algorithm in AnyLogic may differ but the results are the same as this twostep sorting mechanism).

What happens when two tasks of equal priority arrive at the same time? Before we discuss this, let's review a prebuilt model which will help you define and experiment with task priority settings. You can find it in the book's companion materials and is helpful to better explain the nuances of task priority (and task preemption, which we'll discuss later).

Task priority (and preemption) lab

We created this model to let you test different scenarios without having to change the properties from the Properties window of each block. You're don't have to build this model, but you should use it to learn about task priority (and later preemption in Technique 6). The underlying model has two simple processes

that use a shared resource pool. We added some controls to the setup page and the runtime page to simplify your experiments. The setup page's (Figure 3-45) relevant settings are:

1. A slider to set the priority of tasks from **Seize_1**.
2. The same as 1, but for **Seize_2**.
3. An edit box to set the interarrival time, or time between each arrival, from **Source_1**.
4. The same as 3, but for **Source_2**.
5. A checkbox which lets you manually add tasks to the source blocks. Having this checked disables the ability to set the interarrival time, as those options aren't relevant with the manual setting. Internally, checking this box means each time you click the button on the runtime page, arrivals are defined by **Calls of inject() function**

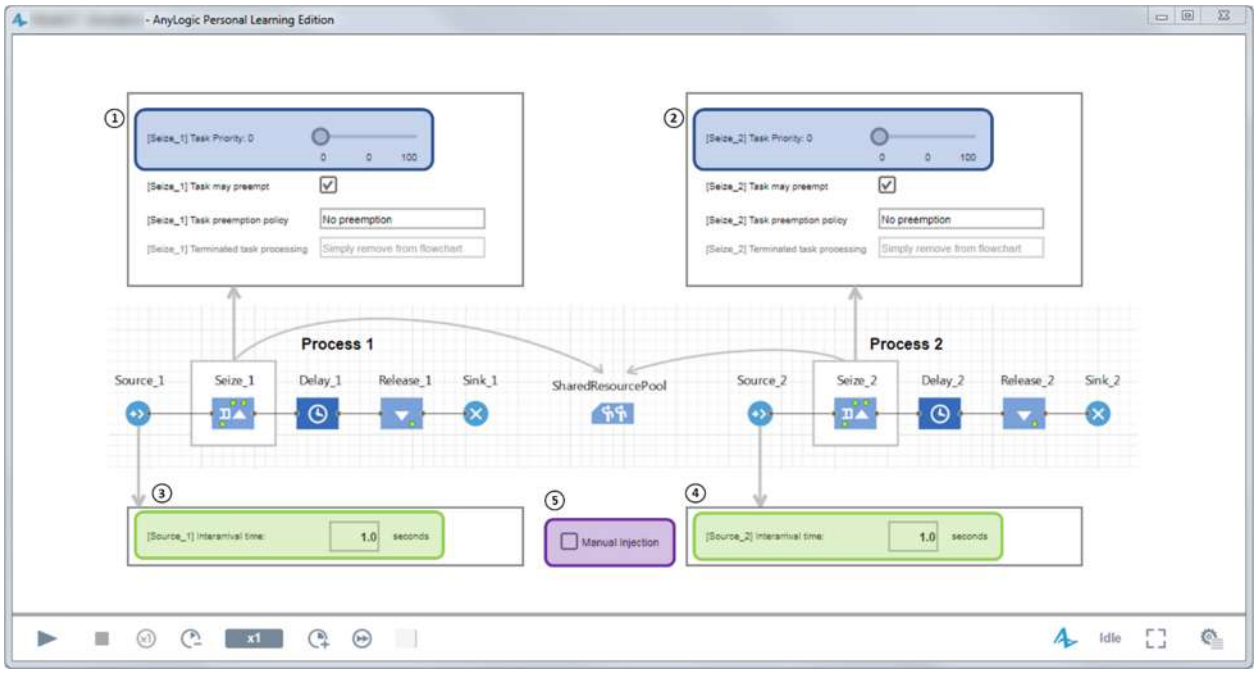


Figure 3-45: Simulator setup window (initialization of the model)

On the runtime page (Figure 3-46):

1. The inputted priority for the tasks from **Seize_1**.
2. The inputted priority for the tasks from **Seize_2**.
3. A red box that shows which process has a higher priority; if they both have the same priority, then neither process will have a red outline.
4. A green highlight showing which delay block currently has the seized resource.
5. This edit box, along with the matching one in Process 2, lets you change the interarrival time of entities injected from the relevant source block.
6. The first rectangular node in a given process showing the entities in the embedded queue of the relevant process's seize block.

7. The second rectangular node in a given process showing the entities in the relevant delay block.
8. The red ring around this task (entity) shows it's using the shared resource pool.
9. The gold fill color within this task shows that this entity is sitting at the head of queue in the relevant seize block.
10. A dynamic clock image from the Pictures panel, modified to show seconds and have its face change color when the model's run status changes. The face is white when the model is paused and green when it runs.
11. By pressing this button, you simultaneously inject one entity into **Source_1** and one into **Source_2**. The button works whether the model is running or paused. In the latter case, you won't see a change until you resume the model.

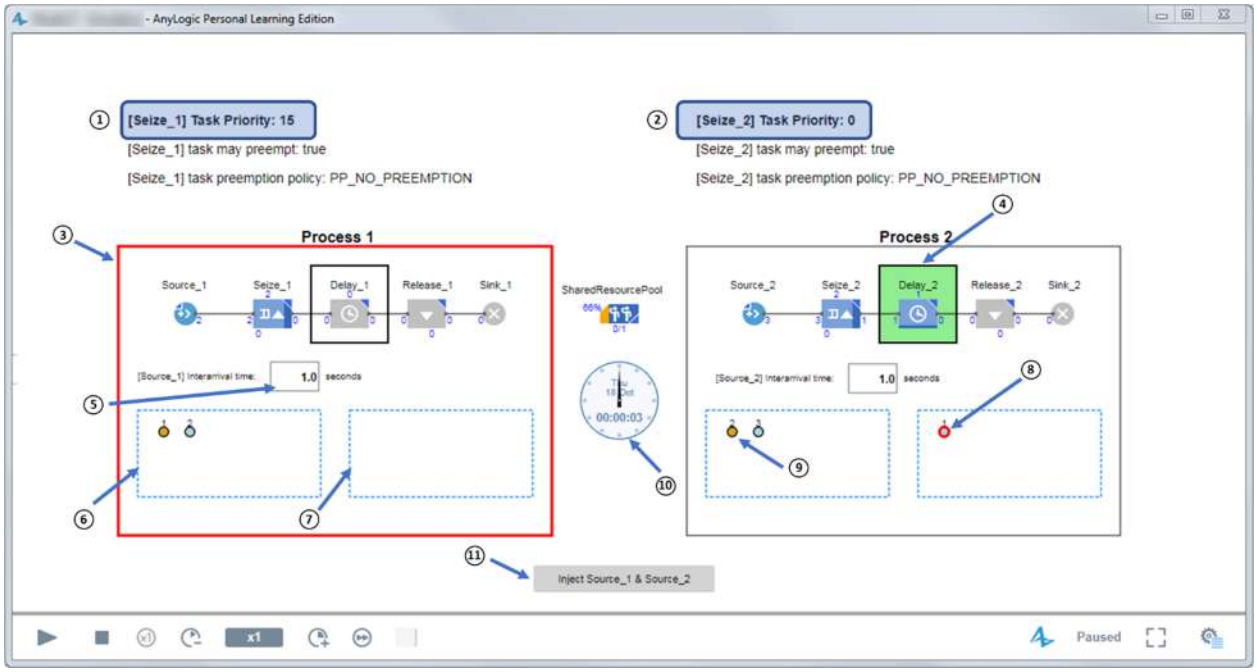


Figure 3-46: Task priority and preemption lab during runtime

The model pauses each time an entity enters or seizes a resource in either of seize blocks. It also pauses when an entity leaves the process by either of the sink blocks. You can modify the pause moments by adding or removing the `simulationPause()` function from the associated extension points. To resume the model, you must click the run button (or press space) to force the model to move forward again. *When you want to define a new scenario, you don't need to close the model; just press the Stop button and it will bring you back to the settings page to define your new scenario.*

To better understand how the task priority works, let's review some scenarios:

Scenario 1: Process 2 (Seize_2's task priority) has a higher priority compared to process 1 (Seize_1's task priority).

On the settings screen, select a value for **Seize_2** task priority that's greater than zero (for example, 10) and press the run button. After every second, two new entities - one from **Source_1** and one from **Source_2** - enter the model at the same time. As you can see in Figure 3-47, tasks from **Seize_2** always take precedence over **Seize_1**'s tasks. The tasks from **Seize_1** never get the chance to seize the resource unit since there's always a request from **Seize_2** waiting in the front (head) of the request queue to grab it as soon as it finishes its current task.

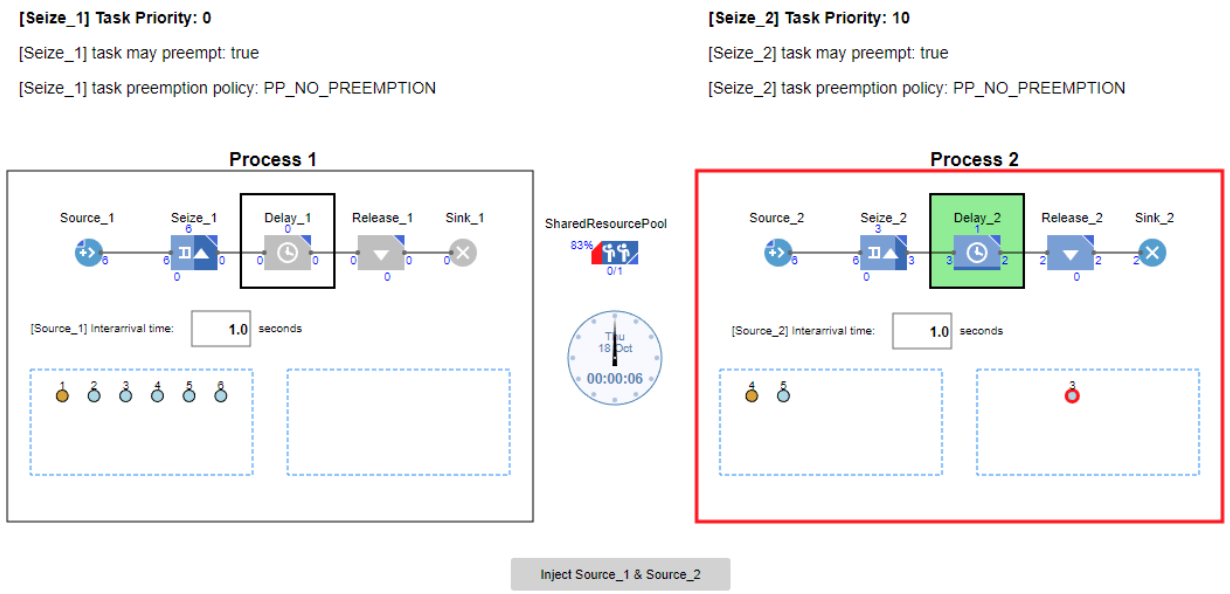


Figure 3-47: Sieze_2 has a higher priority and the model had been run for 6 seconds

Table 3-4 below shows the order of events for the first six seconds. At time 1 in this example scenario, the first two entities enter the model and one resource unit is available in the shared resource pool. Since Process 2 has a higher priority, it will get the unit. From this point on, the resource is always at work. This means all incoming tasks will send requests that are queued up in **SharedResourcePool**.

Table 3-4: Order of unit seized in scenario 1

Order of seizing the unit	Arrival time (number above circle)	Process	Task Priority (Process 1)	Task Priority (Process 1)
1st	1	2	0	10
2nd	2	2	0	10
3rd	3	2	0	10
4th	4	2	0	10
5th	5	2	0	10
6th	6	2	0	10

If you click **SharedResourcePool**, you can see the following information (as labeled in Figure 3-48):

1. Information about the current task being served by the resource unit. The text inside the inner, dark blue rectangle shows the name of the **Seize** block the task came from.
2. List of requests queued up. Each request is associated with a task and has a label with the following format: Priority [P] - *Priority Value* - Preemption [P/N] - *Preemption Policy*.

   For now, we're focusing on the first two places which is P and the value. For example, in the Figure 3-48, the head of the queue's label is P-10-P-NP. This means it has a priority of 10.
3. This is where you can find the **Seize** block (or a block that has an embedded **Seize** block) the request is associated with.

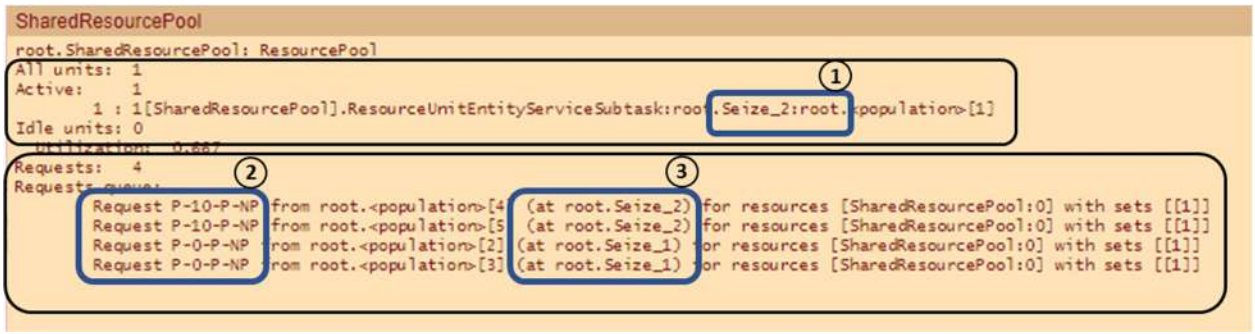


Figure 3-48: Inspect windows of the ResourcePool shows the task being served and queue of waiting requests

Scenario 2: Process 1 has a higher priority compared to process 2 (Seize_1 task priority has a higher value compared to Seize_2 task priority

This lab's settings are like the previous, but this time we swap the priorities. Run the **TASK PRIORITY AND PREEMPTION LAB** and in the setting screen, select a value for **Seize_1** task priority that's more than zero (e.g., 10) and press the run button. Like Scenario 1, two entities enter the processes 1 & 2 at time 1.

However, contrary to what you might have expected, the first task that seizes the resource unit is the one that comes into process 2 (that is, the one that has a lower task priority). If you continue to run the simulation, it works as expected and all the other tasks from Process 1 take precedence over the Process 2 tasks (Table 3-5).

Table 3-5: Order of seizing the unit in scenario 2 (first 6 entities processes)

Order of seizing the unit	Arrival time	Process	Task Priority (Process 1)	Task Priority (Process 1)
1	1	2	10	0
2	1	1	10	0
3	2	1	10	0
4	3	1	10	0
5	4	1	10	0
6	5	1	10	0

The question you might have now is what happened with the first seized task being from process 2, despite it having a lower task priority. Before we get to answer this (apparently) unexpected behavior (and as

you'll see later, is actually expected), we need to review the scenario where both processes have identical priorities.

Scenario 3: Process 1 and Process 2 have identical priorities.

Run the **TASK PRIORITY AND PREEMPTION LAB** and keep all the default values (both priorities are zero) and press the run button. At time 1, two entities enter the processes 1 & 2, just like in previous scenarios. Like the two previous scenarios, the entity that enters the Process 2 seizes the unit and finishes its process first.

It appears it doesn't matter if process 2 has the higher priority (like in scenario 1), if process 1 has the higher priority (like in scenario 2), or even if they're identical (like in this scenario); in all cases, the first entity that enters the process 2 always turns out to be the first one to seize the unit, despite both entities arriving at exactly the same model time (time 1).

Table 3-6: Order of seizing the unit in scenario 3 (first 6 entities processes)

Order of seizing the unit	Arrival time	Process	Task Priority (Process 1)	Task Priority (Process 1)
1	1	2	0	0
2	1	1	0	0
3	2	1	0	0
4	2	2	0	0
5	3	2	0	0
6	3	1	0	0

If you look at Table 3-6, you'll see that in scenario 3 the entities are processed based on their arrival time (e.g. entities that arrived at time 2 are processed before the ones that arrived at time 3). However, the order of execution between the entities (and their related task) which arrived at the same time doesn't seem to follow an obvious pattern. For example, the task from **Source_2** is first at times 1 and 3, but not at time 2.

To answer the ambiguity in the order of tasks being processed, revisit the resource assignment logic (the three cases we described earlier in this section). For the first entity that enters the model at time 1, the resource unit is idle and available, therefore it falls under Case 1; according to the description of Case 1, the first task that sends a request will get the resource. So logically we can infer that at all three aforementioned scenarios (1 to 3), the first task came from **Source_2**.

As for why the first task came from **Source_2** and not from **Source_1**, the answer is straightforward (but not necessarily simple) and comprises of two parts. The first is that:

> *Internal events (that is, events internally generated by AnyLogic blocks) that have identical model time are placed in the order their generating block was added to the graphical editor.*

In this case, this would mean simultaneous events from **Source_1** would come before **Source_2** because we dropped **Source_1** into the graphical editor before **Source_2**. Since this doesn't match the observed order of execution, that brings us to part two, which relates to how AnyLogic executes simultaneous events.

In case multiple events take place at the same model time, AnyLogic executes them according to the "Selection mode for simultaneous events" property of the experiment under the "Randomness" section. By default, AnyLogic uses a Last-In-First-Out (LIFO) policy, where the last scheduled event is the first to run; you can choose First-In-First-Out (FIFO) or random ordering as well.

If you click the **Simulation** experiment of this lab and inspect its **Randomness** section, you'll see the selection mode is set to **LIFO** (Figure 3-49).

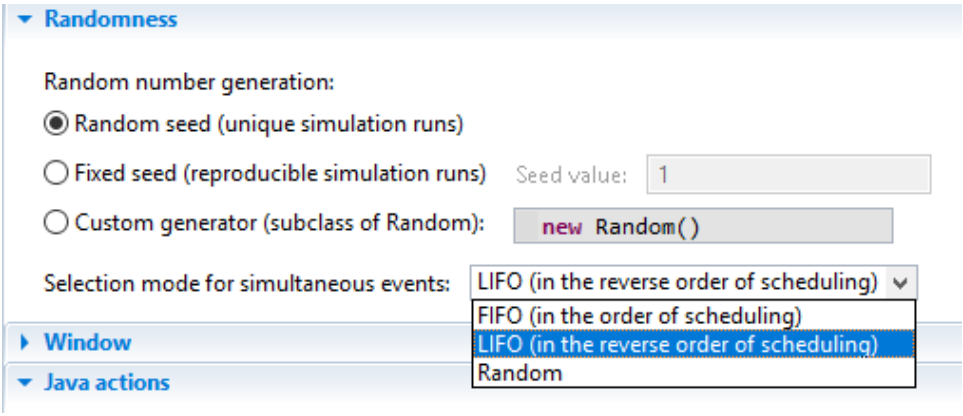


Figure 3-49: Selection model for simultaneous events

Knowing this, we can go back to the three scenarios and explain the observed behavior. These explanations simplify the steps to make them easier to understand.

Seizing behavior explanation for Scenario 1: Process 2 has a higher priority compared to Process 1

- ***At time 0:*** No arrivals have occurred, so the resource unit is idle. **Source_1** schedules its arrival event for time 1 and then **Source_2** schedules its arrival event for time 1.
- ***At time 1:*** The simultaneous events taking place at time=1 (that were scheduled at time = 0), are two arrival events from **Source_1** and **Source_2**. **Source_1** was added to the graphical interface before **Source_2**. This means the order will be event from **Source_1** and event from **Source_2**.

  Since AnyLogic's default execution policy for simultaneous events is LIFO, the program executes the arrival event related to process 2 first and the arrived entity reaches the **Seize_2** and sends the first request to the resource pool. The resource allocation at time 0 falls under case 1 of resource assignment logic. In other words, since the resource is idle when the simulation starts, this first request (from process 2) gets the resource unit. The request from process 1 will be added to the request queue.
- ***At times > 1:*** All entities that enter the model going forth will encounter a busy resource unit and have their request added to the queue. All the requests are sorted by their priority and time of arrival. This means all tasks from **Seize_2** (which has a higher priority) will always take precedence over the tasks from **Seize_1** (including the first request from process 1 added at time 1).

Seizing behavior explanation for Scenario 2: Process 1 has a higher priority compared to Process 2

- ***At time 0:*** Exactly the same as in Scenario 1.
- ***At time 1:*** The same as in Scenario 1.
- ***At time >1:*** All entities that enter the model going forth will encounter a busy resource unit and have their request be added to the queue. This means, all tasks coming from **Seize_1** which has a higher priority will always get the resource unit (including the first request from process 1 added at time 1).

Seizing behavior explanation for Scenario 3: Process 1 and Process 2 have identical priorities

- ***At time 0:*** Exactly the same as in Scenario 1.
- ***At time 1:*** Exactly the same as in Scenario 1. Since we're going to have equal priorities in this scenario, the order of arrivals (and consequently sending the request to the pool) is controlling the order in which resource units are seized. At time 1, when the arrival of **Source_2** happens, it schedules its next arrival (time 2). Then when arrival of **Source_1** happens, its next arrival (at time 2) is scheduled.
- ***At time 2:*** The arrival of **Source_1** that was scheduled after **Source_2** in the previous step (time 1) happens first (in reverse of their schedule, because of LIFO execution). Upon execution of its arrival event, **Source_1** schedules its next arrival event for time 3. Then arrival of **Source_2** happens and schedules the next arrival event for time 3.
- ***At time 3:*** The arrival of **Source_**2 that was scheduled after **Source_1** in the previous step (time 2) happens first and schedules its next arrival event for time 4. Then arrival of **Source_1** happens and schedules itself its next arrival event at time 4. A similar back and forth pattern continues to repeat thereafter.

To better follow the arrival patterns, you can look at the model log in the console window.

We clarified the reasons behind different types of behaviors your model may exhibit under different scenarios. However, knowing this brings up uncertainty of how much you need to know about the simulation engine's workings to build an effective model. Knowing all the simulation engine's intricate generation of underlying events defeats the purpose of having such an engine (which is supposed to simplify the process). This would be cumbersome and in most practical cases, it's impossible to do so for all the simultaneous events that may occur. The actual expectation is for you to be aware of the following rules, as you'll be able to more easily build and debug behaviors.

> *The order of execution of simultaneous events is based on the order the generating block was added to graphical interface. In the default LIFO policy, the last scheduled event is the first to run. When the resource is immediately available, the first task that arrives takes the resource regardless of their priority. In case of simultaneous tasks with immediately available resource, the task scheduled last seizes the resource unit. The priority of tasks coming to a shared resource pool only takes effect when a queue of requests build up due to resource unavailability. In other words, priority of tasks only works when the shared resource is busy and give the request queue a chance for reordering of the requests based on their priority.*

In general, you should follow the following rule in building your models:

The model behavior should be designed to be independent of the order of simultaneous events automatically generated by the simulation engine. If the order of simultaneous events matters to the system under study, you should model them explicitly. You do this by explicitly generating the events in the desired order and treat the order as part of the model logic; you should not let it be undefined, leaving it to the mercy of simulation engine. AnyLogic doesn't support automatic synchronization at engine-level since it reduces performance and overcomplicates the model design. However, these types of scenarios are rare and should be implemented at user-side if needed.

There are several ways to deal with a situation like Scenario 2 where automatically generated simultaneous events result in undesired behaviors. In that scenario, process 1 had a higher priority but the first entity that seized the resource unit was from process 2 (due to the order of drag-and-drop). For example, you can schedule the events manually (Inject **Source_2** & **Source_1** button in the lab), swap priorities and enable FIFO selection mode for simultaneous events or use preemption which we'll cover in Technique 6.

Example Model 3: Pizzeria operation

Prerequisites:

Model Building Blocks (Level 1): **Source**, **Queue**, **Delay**, **Sink**, **Seize**, **Release**, **ResourcePool**, **TaskMeasureStart**, **TaskMeasureEnd** blocks.

Model Building Blocks (Level 2): **Source**, **Seize**, **Schedule, ResourcePool**.

Math: Random variable, Random variate, [Poisson process]

Learning Objectives:

1. Allocation of resource sets
2. Allocation of alternative resource sets
3. Allocation of shared resources
4. Assigning task priorities
5. Using a **Schedule** for arrivals
6. Adding a dynamic clock to the model UI
7. Adding alternative views to the model UI
8. Using the built-in log
9. First order Monte Carlo experiment

Problem statement

This model represents the operation of a takeout-only pizzeria. The pizzeria's owner wants to evaluate its performance during its five hours of daily operation. Figure 3-50 shows an overview of pizzeria's operation and the way orders flow through the system. At the highest level, the pizzeria has three processes: taking orders, cooking, and delivering.

Figure 3-50: Three main processes identified for the Pizzeria's operation

Staff recorded the number of hourly arrivals for a month and rounded the average for each hour, resulting in values in Table 3-7 below.

Table 3-7: Average hourly arrival rate and interarrival time (60/average arrival rate)

Time	Time in Hour	Average arrival rate (per hour)	Average interarrival time (minute)
10:00 am to 11:00 am	0 to 1	10	6
11:00 am to 12:00 pm	1 to 2	110	0.55
12:00 pm to 01:00 pm	2 to 3	40	1.5
01:00 pm to 02:00 pm	3 to 4	10	6
02:00 pm to 03:00 pm	4 to 5	5	12

The staff also recorded the time they spent on each task, summarized in Table 3-8 below by distribution and a fixed value. The distribution is in the form of an AnyLogic function and is used for more accurate outputs of the system. To be comparable, the fixed value is based on the expected value of each distribution. We'll first use the fixed value to ensure our scenarios are deterministic and easier to compare.

Table 3-8: Tasks and their associated time (distribution and fixed values)

Task	Distribution (minute)	Fixed value
Taking Order	uniform(1, 4)	2.5
Waiting in line before leaving	uniform_discr(3, 6)	4.5
Cooking	triangular(3, 10, 5)	5
Delivering	triangular(15, 60, 30)	30

As we mentioned, let's think of this operation as three major processes:

- **Taking orders**: The pizzeria only takes orders by phone and their team of two operators only process incoming orders. These operators usually fall behind, and many customers will grow impatient and leave without placing an order.
- **Cooking**: Orders are placed in a queue and cooked on a first come first serve (FIFO) basis. The pizzeria has five wood ovens and ten pizza makers; cooking a pizza requires one oven and two pizza makers.
- **Delivering**: A fleet of 30 pizza delivery cars will deliver the pizzas. Since the crew work on commission and their task is outside of the pizzeria's operation, they're not at the center of our analysis, although their performance affects one of our key performance metrics which is the total pizza delivered in the five-hour time span.

It is extremely rare a customer cancels an order if the preparation and delivery time take too long. This means we don't need to consider these cases.

Proposed solutions

Several options for improvement have been suggested by the staff and the manager – all of which focus on more efficiently reallocating resources. The data shows the business incurs a significant loss from customers who give up on placing an order after they wait on hold for a long time. Therefore, the following two possibilities have been proposed for consideration, both to occur when incoming calls exceed the operators' capacity to answer them:

Scenario 1: Allow unoccupied pizza makers to help with the calls when there aren't any orders waiting to be cooked.

Scenario 2: Allow unoccupied pizza makers to help with the calls, whether or not there are orders in the queue. In scenario 2, pizza makers can't interrupt (preempt) their cooking to take calls; they must finish the cooking first and then see if a customer is waiting on the line.

Key metrics

The pizzeria's manager has selected the following performance metrics:

1. Total number of pizzas delivered in the pizzeria's daily five-hour operation, which the pizzeria is trying to maximize. This metric is impacted by all three steps in the operation and gives a good overall value for the pizzeria's efficiency. While we won't count enroute pizzas at the end of the five hours toward this metric, they'll add to the revenue.

2. Number of lost customers from insufficient or inefficient order taking. Although the order taking performance implicitly shows its effect in the total pizzas delivered, it is useful to know the exact number of customers lost due to the deficiency in the order taking process.

3. Average time an order takes, starting from the time after the customer placed their order and when it became ready for delivery; this is another metric that's trying to be minimized and is a good way to test the kitchen performance.

Base (as-is) scenario, Scenario 1, and Scenario 2 with deterministic times

To simplify our understanding of the system's behavior under proposed scenarios, we'll build the model with fixed values (deterministic models). These fixed values are selected to be equal to the expected values of the actual distributions. The expected values are chosen to make sure the outputs of the deterministic and stochastic versions of the model are comparable. Figure 3-51 shows the final AnyLogic model with the three major processes separated by colored rectangles.

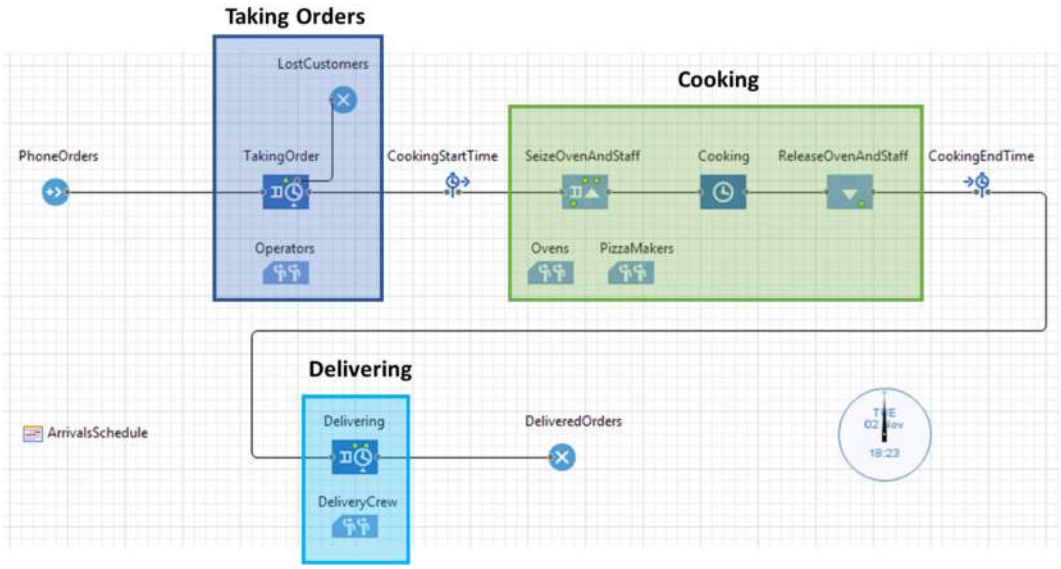


Figure 3-51: Pizzeria's operation model in AnyLogic

Building the base model and its outputs

1. Create a new model; time units: minutes.
2. Build the flowchart as shown in Figure 3-52 by dragging and dropping the needed blocks from the Process Modeling Library (PML) palette. When referencing this figure, note that field names are in black and the needed change is in blue; settings not listed should be kept at their default values. In addition, the "≈" symbol and red dotted arrow are annotations, indicating a visual continuation.

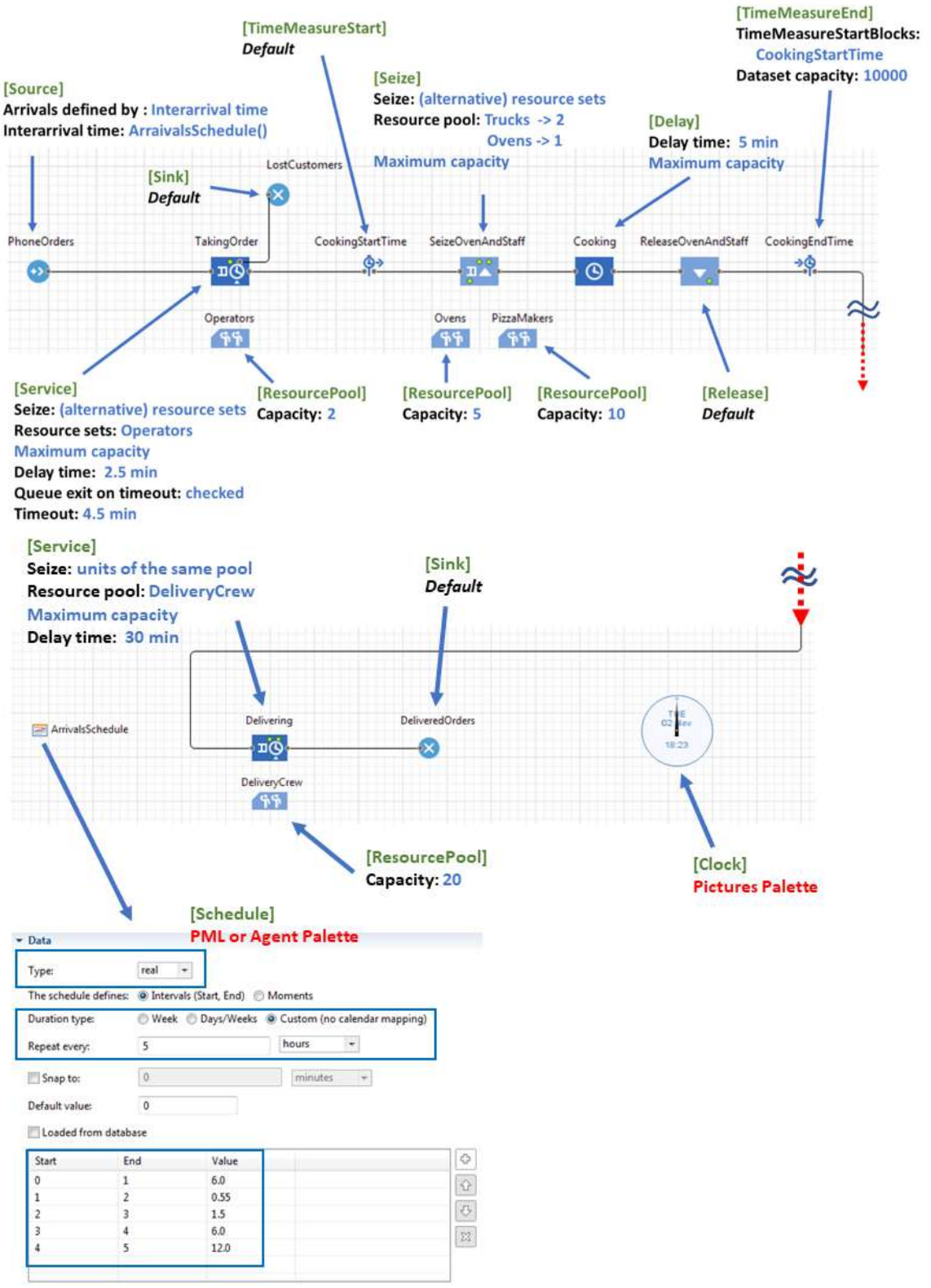


Figure 3-52: Pizzeria operation flow chart - base scenario with deterministic time values; The "≈" symbol indicating a disconnect

Before we continue, let's look more in-depth about how each of the three primary processes are implemented within this flowchart.

Taking Orders is modeled with a **Service** block that, in the base scenario, only use resource units from the **Operators** resource pool (Figure 3-53). The seize policy is set to **(alternative) resource sets** because it lets us define more than one resource pool, in cases where a resource from the primary resource pool isn't available (please refer to **MODEL BUILDING BLOCK LEVEL 2** for more information). For the base scenario, we're only using **Operators** as the sole resource set. However, we've selected this option since we want to add alternative sets in later scenarios.

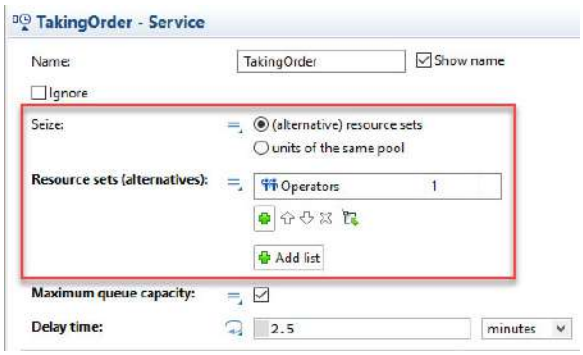


Figure 3-53: Operators as the only set of resource pool associated with the "TakingOrder" service

As an aside: having a single alternative (as shown above) and selecting **units of the same pool** with the **Operators** resource pool selected will both behave the exact same way. You may then ask, why even have the option for **units of the same pool** when a single alternative does the same thing. The simple answer is that having the **units of the same pool** option allows us to explicitly set the option for a single pool, which guarantees you can only have one pool selected at a time.

In the **TakingOrder** - **Service** block, we use the "*outTimeout*" port. To activate this port and cause entities to leave after a selected time, you must check the **Queue: exit on timeout** checkbox and provide the timeout value (Figure 3-54). This value shows the maximum time an entity will wait in the queue before it renegs – a behavior associated with queues where entities leave after a long wait. In this case, the queue is one embedded within the **Seize** block.

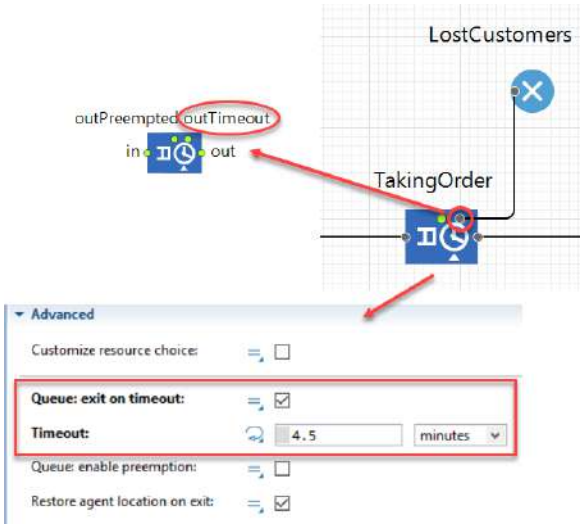


Figure 3-54: Showing what the port is and how to setup reneging behavior for the Seize block's embedded queue

To define an interarrival time that varies during the five hours of operation, we'll use the **Schedule** object from the PML or Agent palette. One way to understand how the **Schedule** object works is imagine it as a function, which takes the current time as its input and outputs a value. As shown in Figure 3-55, in the schedule setting:

a) first, we specify the type of value it returns. For interarrival times we need a real number, which is any value we can find on the number line. These include whole numbers/integers, decimal numbers, negative numbers, and irrational numbers (such as Pi).

b) During each hour (which was a start and an end time), we want the output to be the same value; therefore, we select the **Intervals (Start, End)** option. As each timespan occurs within five hours, we select the **Custom** duration time. This allows us to specify when the end of the cycle is (5 hours in our case), at which point the cycle starts again.

c) Finally, we must specify each period start time, end time, and the value the Schedule should return during that period. The unit of time used in the **Start** and **End** column matches the unit of time we specified for the cycle period (hours). The value that the schedule returns is unitless; we specify its unit in the block that calls this schedule (in this case, the **Source** block).

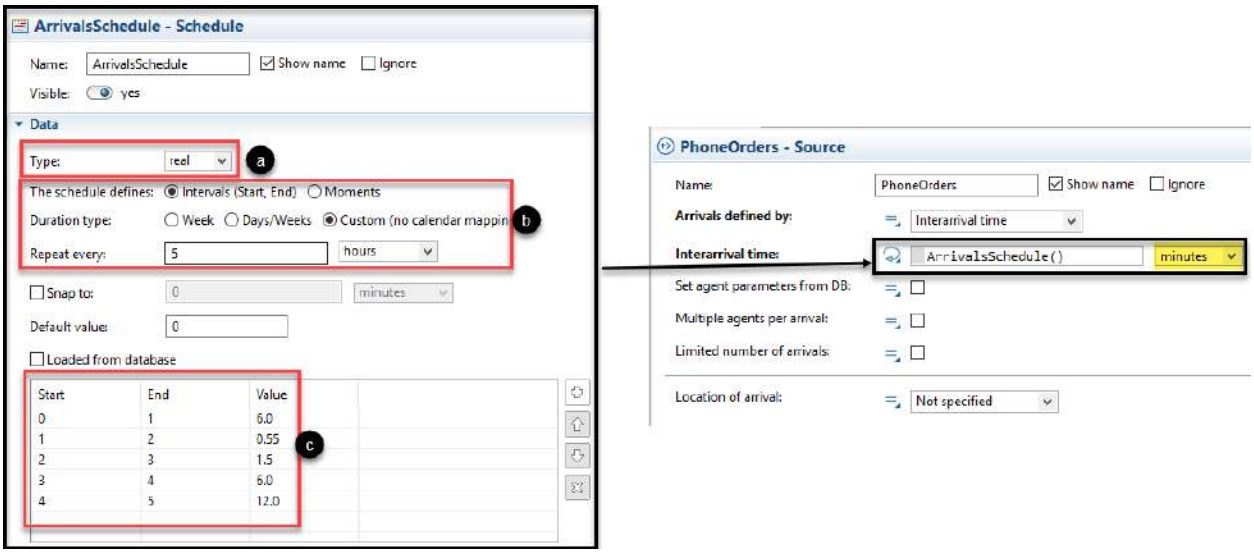


Figure 3-55: Schedule object and its application for interarrival times in the Source block

[Optional alternative option for the Calendar object (not implemented in following steps)]

Instead of the duration type to **Custom**, which has no calendar mapping, we could have used the schedule to map the interarrival times to an actual calendar. As shown in Figure 3-56 (left), each hour of a day is mapped to a real number. You should be aware that in the example shown in Figure 3-56, the Sundays and Saturdays are unchecked (no value is mapped to them), therefore the calendar returns the value in the **Default value** field which is set to 0. In case you've decided to map values to calendars, you should make sure that your experiment Start and Stop time is set correctly. Figure 3-56 (right) shows an example of a Simulation experiment in which the start date is at 12:00 am of a Monday and stops five hours later, therefore the model time covers a cycle of operation maps in the **Schedule** object.

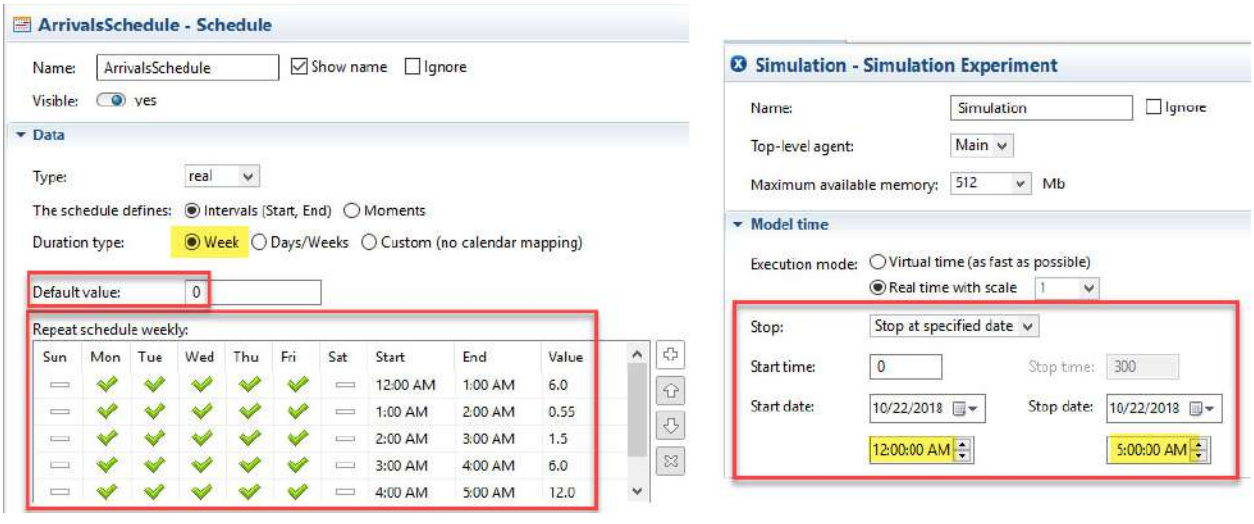


Figure 3-56: Alternative setting for the schedule object (calendar mapping) and proper setting for the Simulation experiment for this setting

Cooking is modeled with the familiar **Seize** + **Delay** + **Release** combination. The only new concept here is we've used a resource set made up of two pizza makers and one oven for each task (as shown in Figure 3-57). For the process to start, both pizza makers and an oven must be available. When going to add the ovens, make sure the click the green "+" next to the up-arrow icon; clicking the green "+" with the "*Add list*" label will add another alternate source set, which isn't what you want for this case.

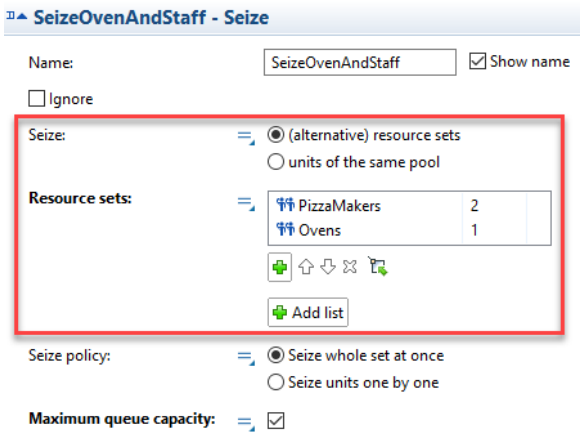
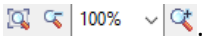


Figure 3-57: Seize setting of the "SeizeOvenandStaff" block

To record and analyze the time it takes to cook each order, we added a pair of **TimeMeasureStart** and **TimeMeasureEnd** blocks, respectively named **CookingStartTime** and **CookingEndTime**. In the properties of the **CookingEndTime**, we increase the **Dataset capacity** to 10000 (ten-thousand) to insure enough space for all the orders that will pass through this block. The actual number of orders we'll receive for this base deterministic case is 174; to be on the safe side, 10000 is used in case you later want to increase the number of arrivals.

Delivering is simply modeled with a single **Service** block that uses a dedicated resource pool of 20 delivery workers. For this block, since we know no other alternative resources will be used for any of the scenarios, the option **units of the same pool** is selected.

At this stage, we've finished our model. Now, we'll add elements from the **Analysis** palette to display the model's outputs in an understandable way. We'll also add a couple of view areas to expand the model's interface and separate it into two frames.

3. Zoom out by holding Ctrl (Mac OS: Cmd) and scroll down with the mouse wheel; alternatively, you can use the zoom controls found in top bar: 

. You should see the entire blue frame representing the model.

4. Drag and drop two **ViewArea** objects from the Analysis palette. As shown in Figure 3-58, place the first at the origin, name it **viewArea_Process** and set the title to "Process"; add the second **ViewArea** next to the first one, name it **viewArea_Stats** and set the title to "Stats".

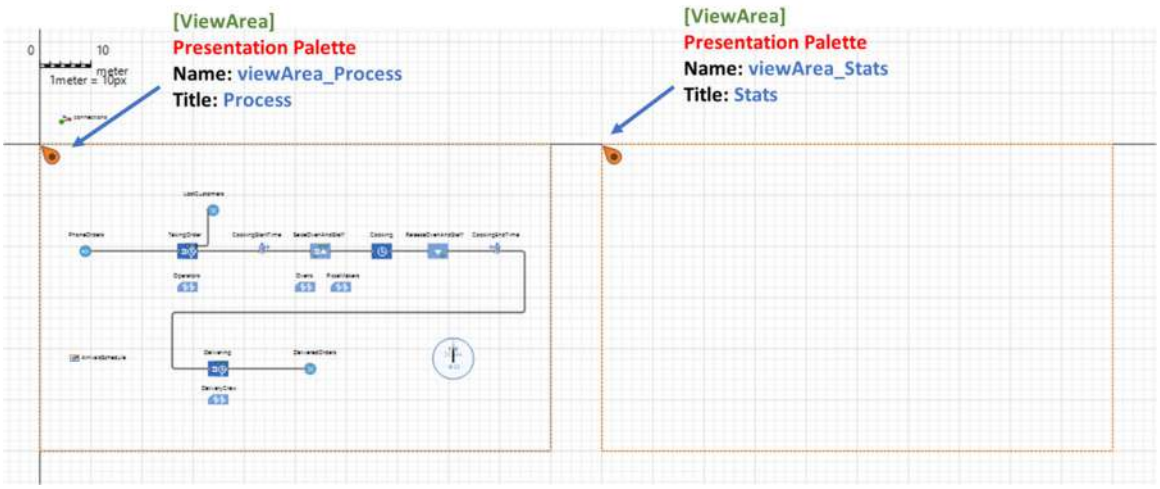


Figure 3-58: Adding two view area objects, changing their names and titles

5. Generate both the **distribution** and **datase** charts from the **CookingEndTime** block by right-clicking it and selecting the options under the **Create Chart** submenu.

    a. The distribution chart displays a histogram where the X-axis is the time spent cooking and the Y-axis is the frequency that time occurred.

    b. The dataset chart displays a line graph where the X-axis is the time an order finishes and the Y-axis is the time it took to cook the finished order.

    For both charts, AnyLogic will generate random shape to show you how the graph will look (in terms of the colors, line widths, etc.). You'll see the actual graph when you run the model.

    c. Add three **Output** elements and rename them to **TotalDelivered**, **TotalLostCustomers**, and **MeanCookingTime**. Move the graphs and **Output** element into the "Stats" view area (the one on the right), as shown in Figure 3-59.

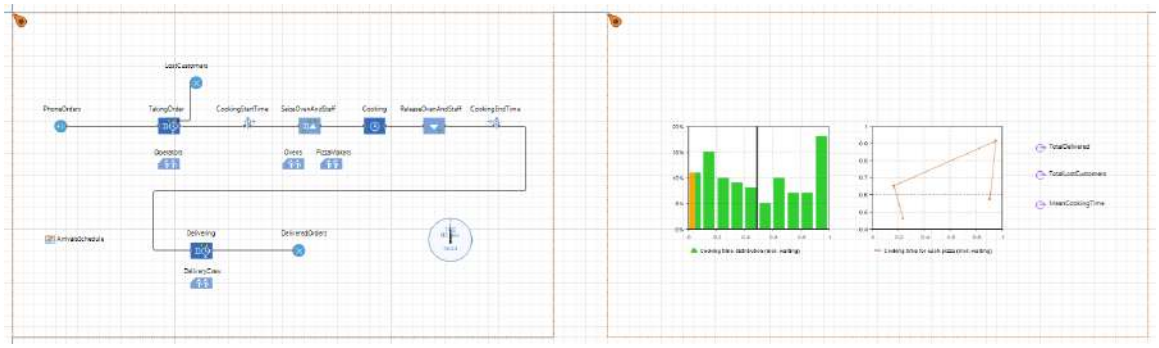


Figure 3-59: Adding charts and output objects to the stats view area

6. Modify the properties and the charts and **Output** elements according to Figure 3-60 below:

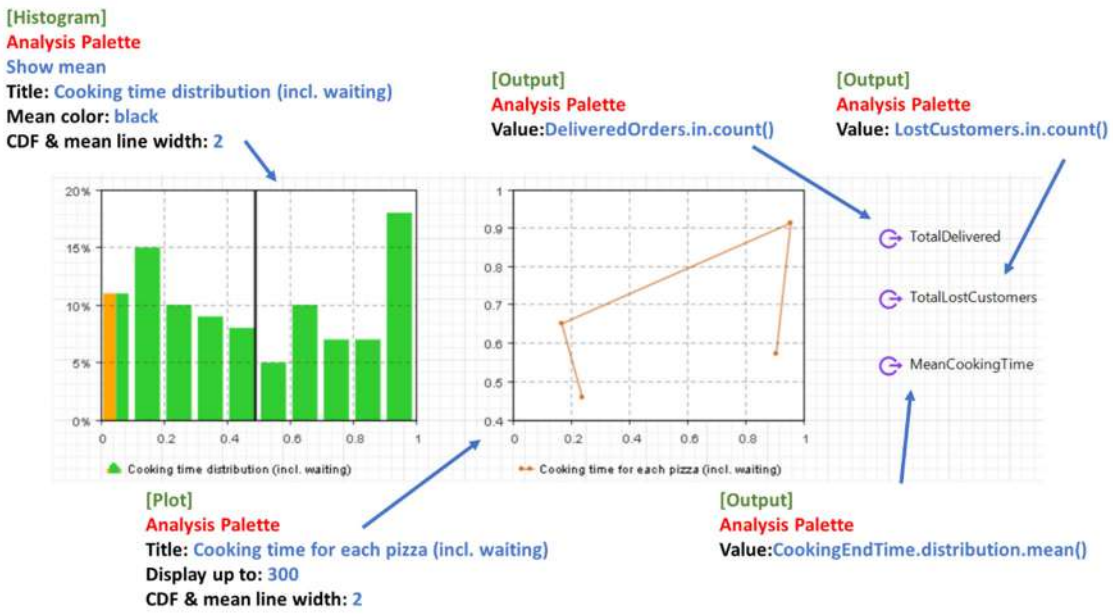


Figure 3-60: Properties of charts and output elements

Adding scenarios to the base model

Remember we want three separate versions of the same model - one as the baseline and two for the scenarios we want to test. With our base scenario complete, we must build the two scenarios. One option is to save the model under different names, resulting in three separate models. However, since our model structure is simple and its components are all in one agent ("**Main**" for now), we can use a different trick to keep all three scenarios under one model name.

To do this, we'll build three top-level agents and then build experiments associated with each of them. We'll learn about agent types in the next chapter; but for now, imagine the top-level agent as the universe that everything lives within. When we created our model, AnyLogic automatically added an agent called **Main** and a simulation experiment called **Simulation** with **Main** as its top-level agent. To build the two test scenarios, we must do the following:

1. Start by right-clicking the **Main** agent under the projects view and rename it to "BaselineScenario" (Figure 3-61). This is more aptly named and will make it clearer what its significance is in relation to the other scenarios we'll soon create. After doing so, you'll notice that AnyLogic refactored the name of the associated Simulation experiment to **Simulation: BaselineScenario** to match the top-level agent's name.

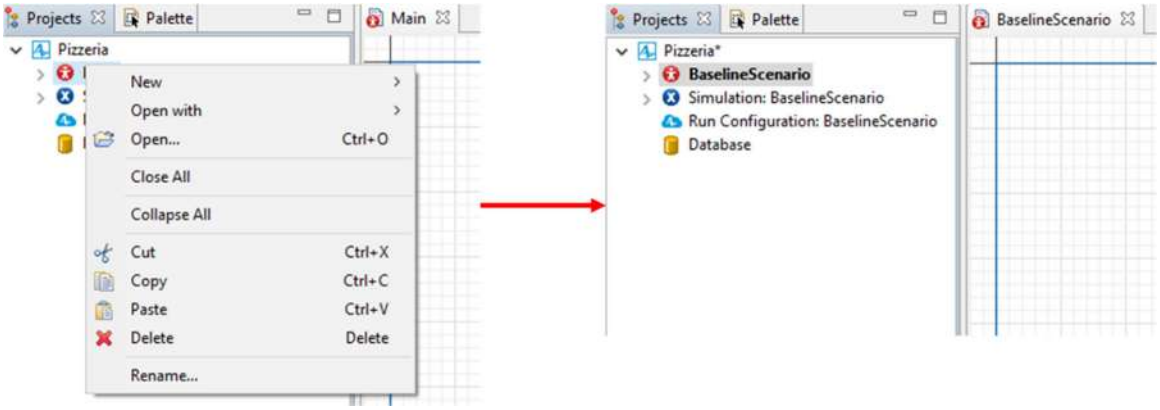


Figure 3-61: Renaming the "Main" agent type to "BaselineScenario"

2. Now we're ready to make two more copies. Right-click the **BaselineScenario** agent type (red icon) and select **Copy**. Then click the model's name (for example, Pizzeria) and paste it – do this a second time, so you have three agent types.

3. Rename the two newest agent types "Scenario1" and "Scenario2". Your Projects panel should look like the right picture in Figure 3-62 with three appropriately-named agents and one simulation experiment.

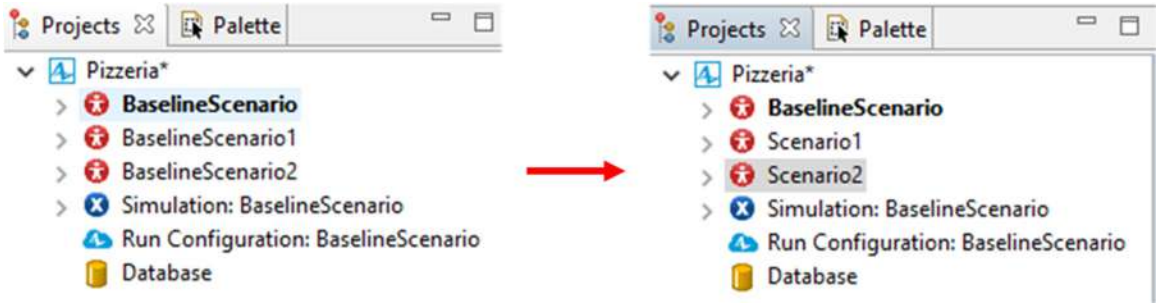


Figure 3-62: Copy pasting the agent types two times and rename them to represent the two test scenarios

4. Click the **Simulation: BaselineScenario** experiment and check its properties. Like in Figure 3-63, you should see the top-level agent is **BaselineScenario**.

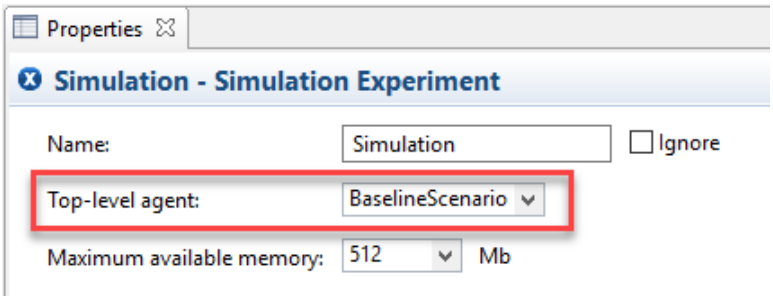


Figure 3-63: Top-level agent of "Simulation: BaselineScenario" experiment

5. To build the simulation experiments for "Scenario1" and Scenario2", start by right-clicking the model name (for example, Pizzeria), select **New** and then **Experiment** (Figure 3-64).

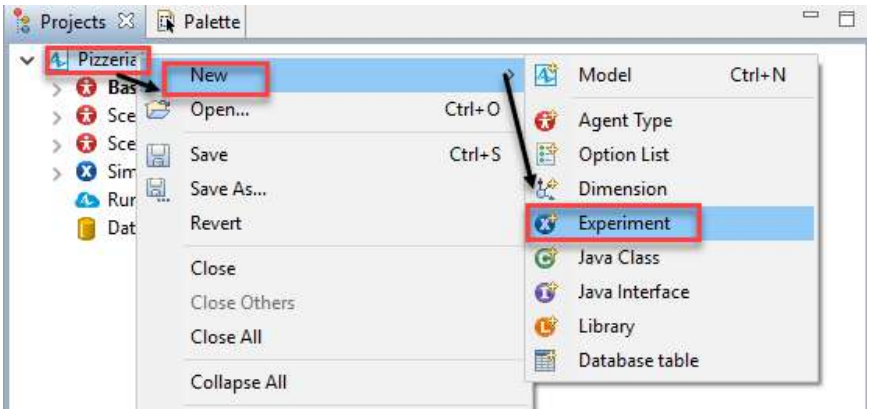


Figure 3-64: Steps of building a new Experiment in a model

6. In the **New Experiment** window, change the top-level agent to Scenario1 and press **Finish** (Figure 3-65).

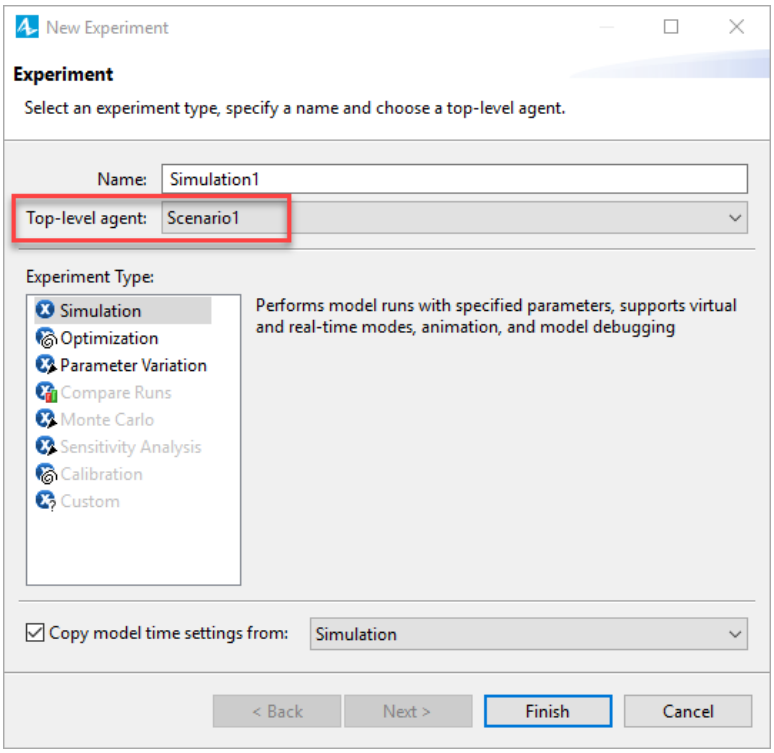


Figure 3-65: Building a new Simulation experiment for scenario 1

7. Repeat the previous two steps for Scenario2. Afterwards, you show have three agent types (**BaseScenario**, **Scenario1**, **Scenario2**) and three Simulation experiments (**Simulation**, **Simulation1**, and **Simulation2**), such as in Figure 3-66.

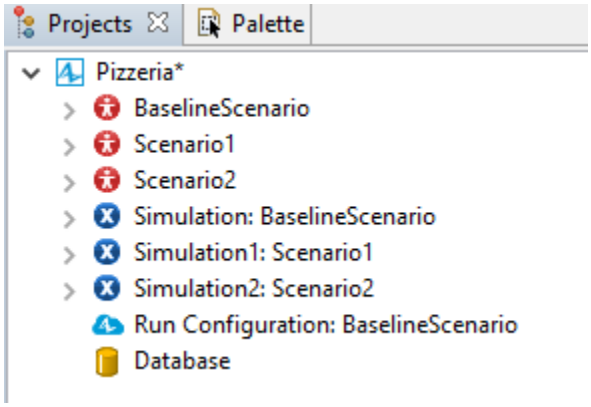


Figure 3-66: Three to-level agents and their associated simulation experiments

8. To modify the first scenario, double-click the **Scenario1** agent type in the Projects view. Click the **TakingOrder** block and in its properties add an alternative resource set by clicking the plus sign with the "*Add list*" label.

   Having a resource pool in a separate list - as opposed to one – means an incoming entity will try to seize the first set and then the others based on their order under the resource set field. Assign **PizzaMakers** with a quantity of one as the alternative resource set (Figure 3-67). With the pizza makers in a separate list, incoming calls will first try to seize an available operator. If one isn't available, it will try to seize an available pizza maker. If neither operators or pizza makers are available, the entities must wait in the service block's embedded queue.

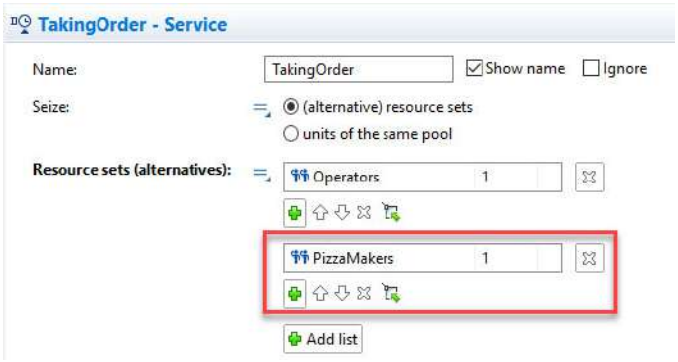


Figure 3-67: Changing the "TakingOrder" property in Scenario1

Setting up the second scenario begins similarly to the first scenario: First double-click the **Scenario2** agent type in the **Project** view. Click the **TakingOrder** block and in its properties add an alternative resource set by clicking the plus sign with the "*Add list*" label. Assign **PizzaMakers** with a quantity of one as the alternative resource set (Figure 3-68). Also, In the **Priorities/preemption** section, change the value of **Task priority** to 1, as shown in Figure 3-68. Increasing the task priority will result in this task (**TakingOrder**) gaining precedence over the **SeizeOvenAndStaff** block in using the shared resource (that is, the pizza makers).

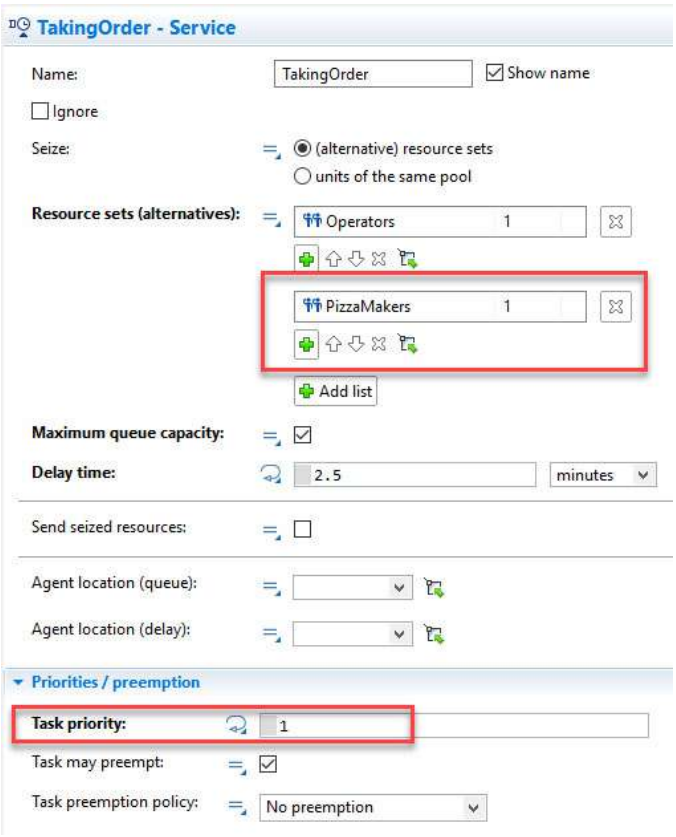


Figure 3-68: Changing the "TakingOrder" property in Scneario2

Outputs of the Base (as-is) scenario, Scenario 1, and Scenario 2 with deterministic times

With these steps complete, we can try to run each scenario. For each experiment, right-click the **Simulation** experiment, then select **Run**. Within the model window, click the icon in the lower right corner to toggle to the **Developer** panel (green box in Figure 3-69). On the developer panel, click the drop-down menu (red box in Figure 3-69) and select the **Stats** view area. Now, we're ready to run each scenario and compare their outputs.

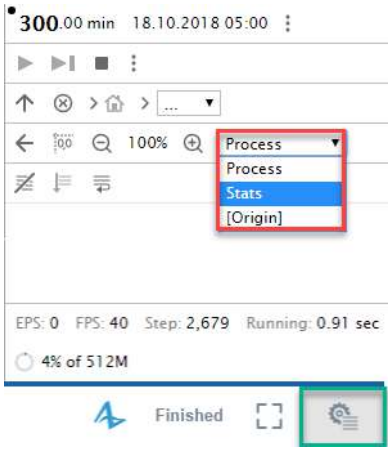


Figure 3-69: Navigating to the "Stats" view area

In our baseline scenario (results shown in Figure 3-70 and Figure 3-71), pizza makers don't answer the phones. As a result, the ovens and pizza makers are only active 38% of the time. The fact each pizza was cooked in 5 minutes shows there were never a shortage of ovens or pizza makers. But the numbers of operators was clearly inadequate, as 59 customers abandoned their calls after they waited for more than 4.5 minutes.

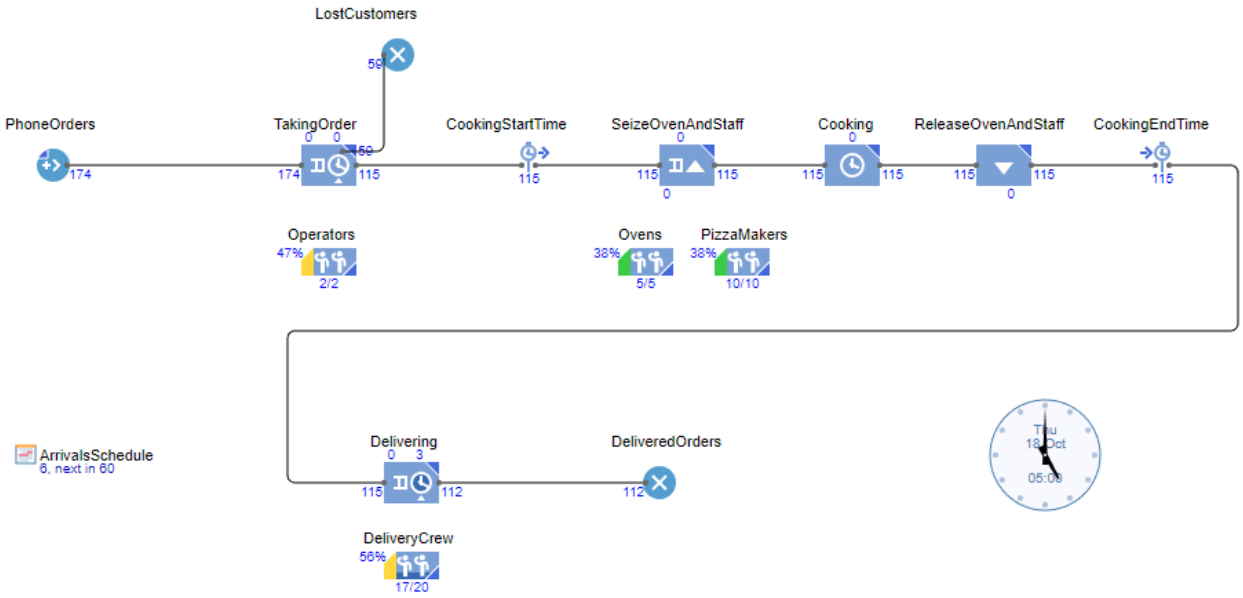


Figure 3-70: Outputs of base scenario (process view)

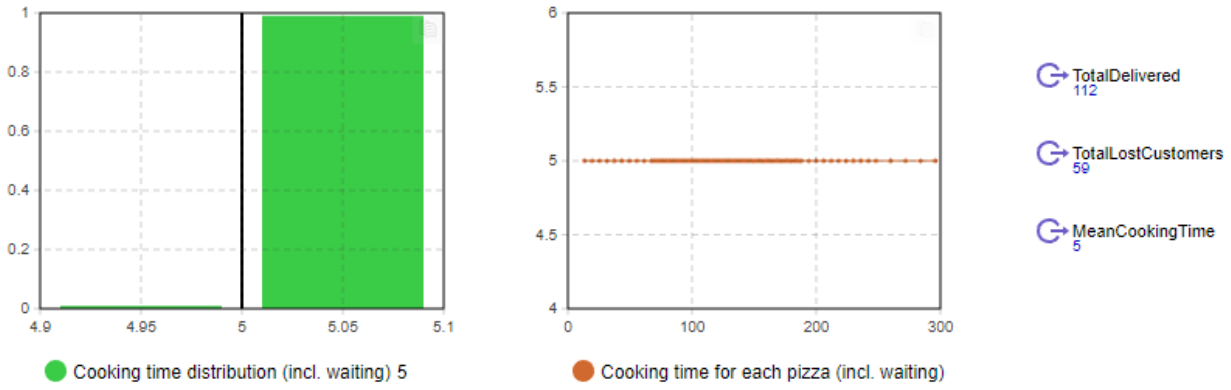


Figure 3-71: Outputs of base scenario (stats view)

With scenario 1, pizza makers will answer calls if all the following conditions are true:

- All phone operators should be busy
- There are orders (calls) waiting to be answered in the embedded queue of the **TakingOrder** block
- There are no orders in the embedded queue of the **SeizeOvenAndStaff** seize block (that is, there are no pizzas waiting to be cooked).

This scenario's outcome (Figure 3-72 and Figure 3-73) reduced the number of lost customers, but it also increased the mean cooking time. This is expected since in peak times, pizza makers who answer the phone will be unavailable for the duration of taking the order (2.5 minutes), resulting in pizzas having to wait to be made.

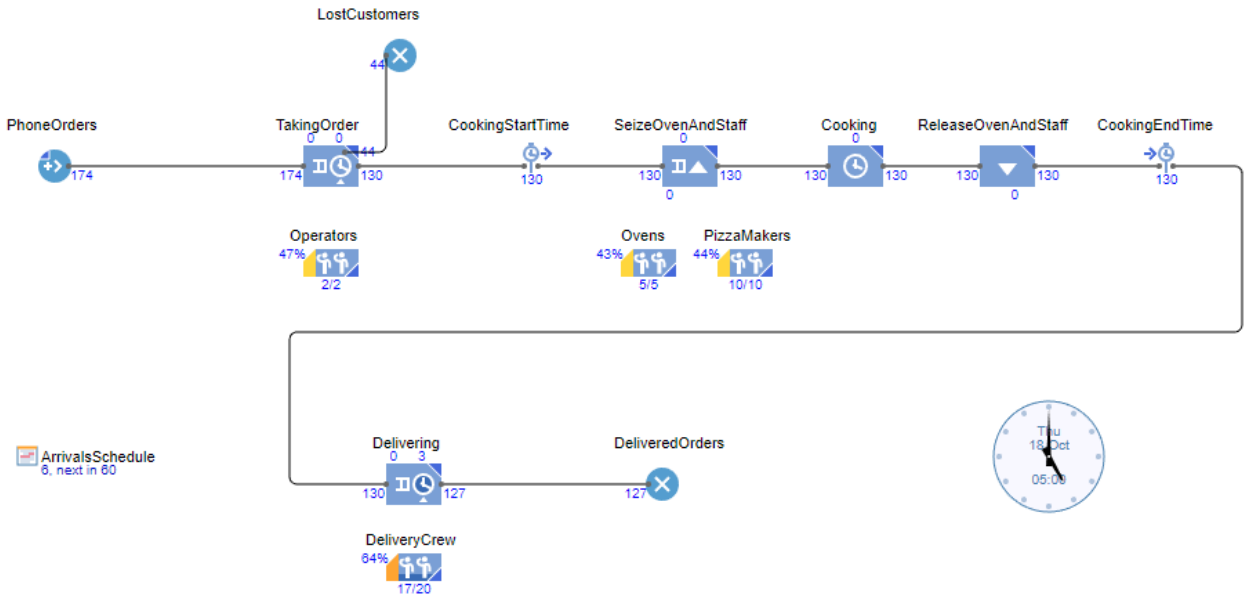


Figure 3-72: Outputs of scenario 1 (free pizza makers help the operators) – Process view

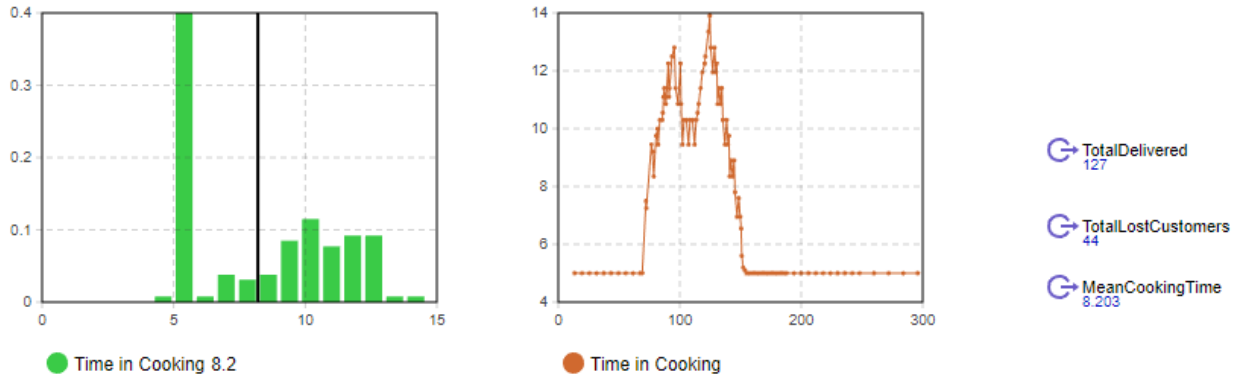


Figure 3-73: Outputs of scenario 1 (free pizza makers help the operators) – Stats view

Finally, in Scenario 2 not only do pizza makers help the operators, they give priority to doing so. For a pizza maker to answer a call in this scenario, all the following conditions must be true:

- All phone operators should be busy
- There are orders (calls) waiting to be answered in the embedded queue of the **TakingOrder** block
- The pizza maker should not be in the middle of making a pizza

If there are orders in the embedded queues of **TakeOrder** and **SeizeOverStaff** competing for **PizzaMakers**, the makers will help with the calls, since that block has a higher priority. In other words, when the phones

are ringing and all operators are busy, a pizza maker who isn't baking will take the order (call) rather than start baking a pizza. It's important to note we didn't set a preemption policy. This means a cook who has started to bake a pie won't answer the phone. We'll talk about preemption policy later (TECHNIQUE 6).

This scenario's outcome (Figure 3-74 and Figure 3-75) shows a drastic reduction in lost customers – all hungry customers have placed their orders! Unfortunately, customers who order during peak times must wait over an hour for their pizza to get cooked.

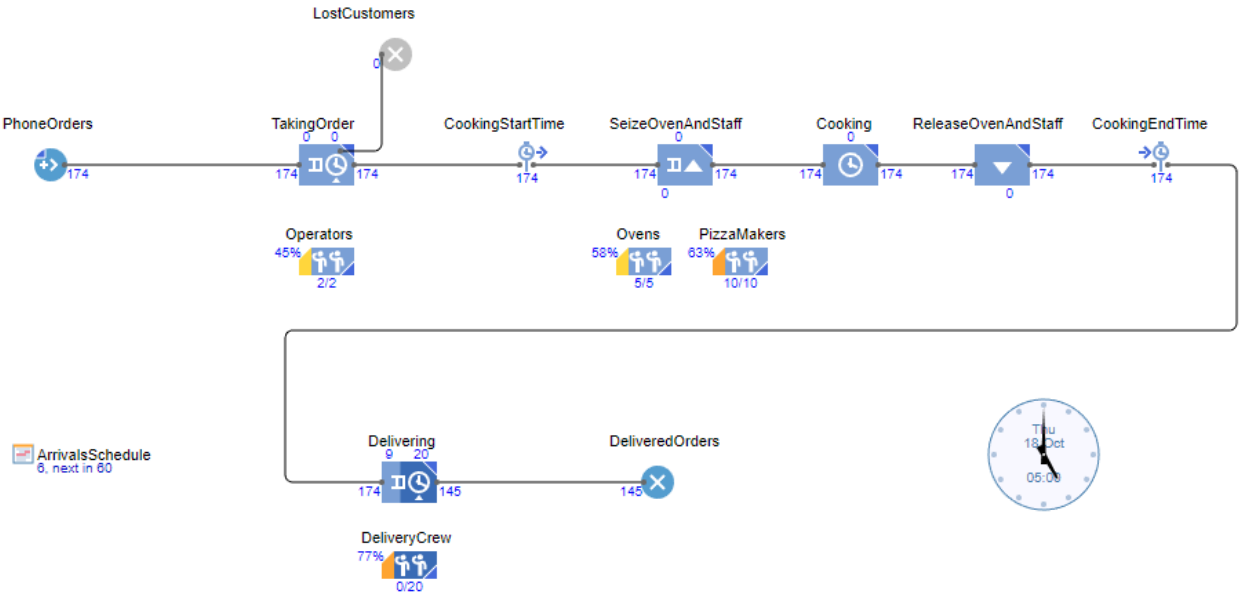


Figure 3-74: Outputs of scenario 2 – Process view

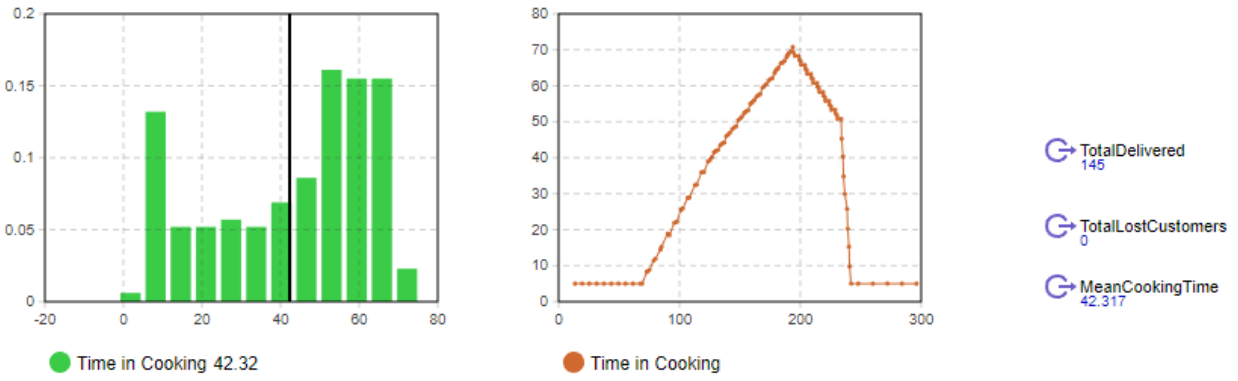


Figure 3-75: Outputs of scenario 2 – Stats view

When you use Table 3-9 to compare the key metrics from the three operation policies and assume our goal is to minimize lost customers, Scenario 2 is the definite winner. However, the baseline model where pizza makers never answer the phone offers a far better compared to Scenario 2. As you can see, selecting a proper performance indicator is important part of comparing alternative scenarios.

Table 3-9: Outputs of three scenarios (deterministic)

Scenario	Total Delivered	Total Lost Customers	Mean Cooking Time (min)
Baseline	112	59	5
Scenario 1	127	44	8.203
Scenario 2	145	0	42.317

One key point to consider with the total delivered metric is it only covers the five hours of the pizzeria operation. However, in all the above scenarios there are still undelivered pizzas by the end of five hours of operation. If you look at the top of the Delivering block in each scenario, you'll see two numbers that represent: the number of orders in the embedded queue, and the number of entities actively using a resource (from left to right, respectively).

For the baseline and first scenario, there are no pizzas waiting to be delivered. However, there were three orders being delivered. The second scenario's case is the worst – there are nine pizzas in the embedded queue waiting for delivery and 20 being delivered. In all, we still have 29 cooked pizzas in progress.

While we improved the processes for taking orders and cooking, we also shifted the bottleneck to the delivery crew. All our scenarios require the crew to deliver pizza outside of working hours. That's why our performance metric is the number of pizzas delivered during the pizzeria's daily five-hour operation. Setting the metric this way made sure those undelivered pizzas won't have a positive impact.

Remember our task is to simulate the proposed scenarios and report their results. The pizzeria's management will review the outputs and will use their objectives and constraints to decide. Our task isn't complete though; the current model is over-simplified, as we used deterministic times for each task to complete. This was useful for comparing scenario runs quickly and easily, but it wasn't an accurate portrayal of the system.

Activating the built-in Log to record model outputs automatically

Before we add randomness to our model, let's review the AnyLogic's built-in log. In this example, we've recorded custom outputs with a **Histogram**, **Time Plot**, and three **Output** objects. AnyLogic also allows you to automatically log certain outputs of blocks and objects in your model. To enable this feature, you must:

1. Click the **Database** entry in the **Projects** panel and in its Properties, check the **Log model execution** checkbox (Figure 3-76). This setting will let AnyLogic know you want to record the default log during the simulation run.

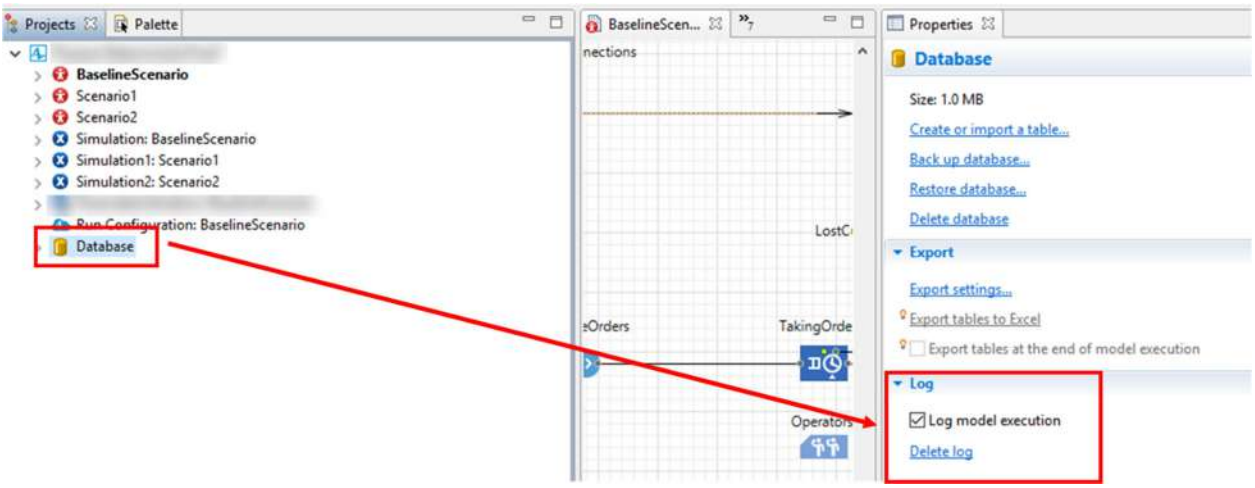


Figure 3-76: Activating the built-in Log for a model

2. Test this by running the simulation for **BaselineScenario** until the end, then close the run time window and return to the **Projects** panel in the development environment. You'll see a small expansion arrow to the left of the **Database** icon.
3. Click the arrow to expand the view. You'll see many tables (views) but for this example, we're only interested in **dataset_log**, **histogram_log**, **resource_pool_utilization_log**, and **resource_unit_utilization_log** (Figure 3-77).

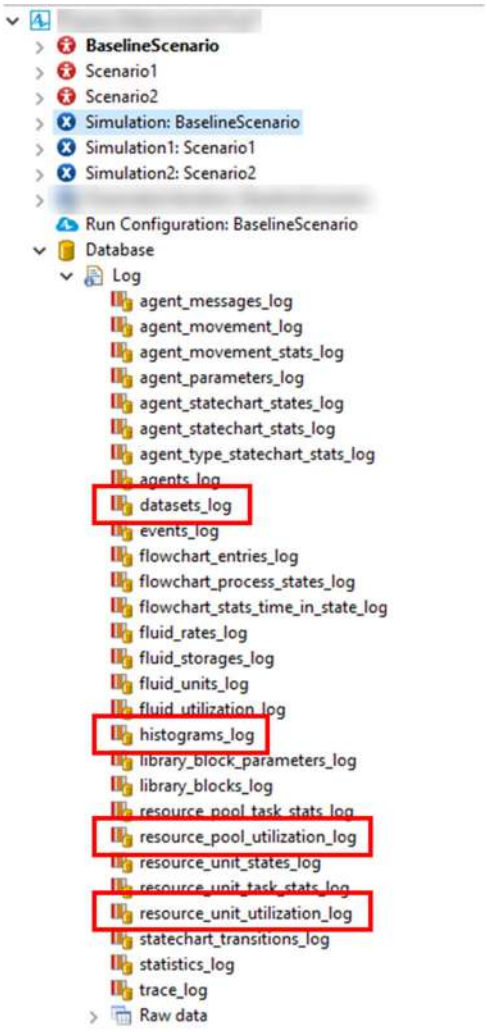


Figure 3-77: Expanded results of the database log

4. If you double-click any of the views in the log, you can open them to see the outputs; below is an explanation of the ones we're interested in. Some of the logs will be empty – indicating your model didn't perform the required actions needed to log. E.g., we didn't use the fluid library at all, so any log prefixed by "fluid" will be empty.

   dataset_log shows every entry for each dataset in you model. To filter the log content by items in a column, click the empty cell below its label and select the items you do or don't want to see, then click Ok to confirm your choice (Figure 3-78).

   Our example only has one dataset (the embedded dataset of the **TimeMeasureEnd** block, **CookingEndTime**) and therefore there's only one entry for "*agent_type*", "*agent*", and "*name*" columns (Figure 3-78). The **dataset_log** also shows every single entity that has passed through a dataset, indicated by columns "*x*" and "*y*", which respectively show the entity's entry and elapsed times.

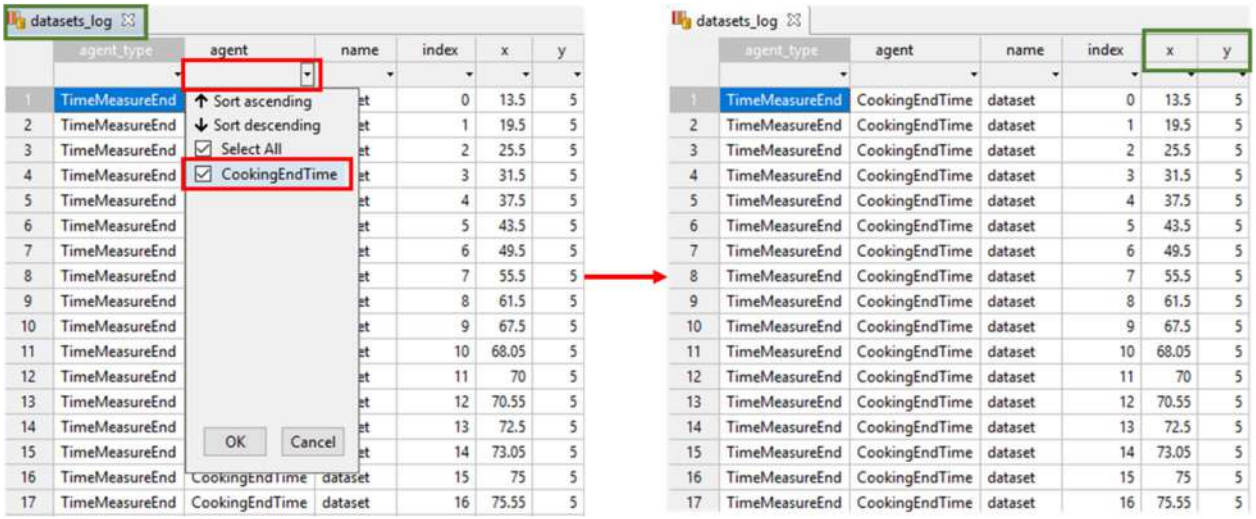


Figure 3-78: "dataset_log" view of the AnyLogic Log

histograms_log shows, for each histogram bin's, the start and end values, the probability density function value, and the cumulative distribution function values (Figure 3-79).

histograms_log

	agent_type	agent	name	start	end	pdf	cdf
1	TimeMeasureEnd	CookingEndTime	distribution	4.9	5	0	0
2	TimeMeasureEnd	CookingEndTime	distribution	5	5.1	0	0
3	TimeMeasureEnd	CookingEndTime	distribution	5.1	5.2	0	0
4	TimeMeasureEnd	CookingEndTime	distribution	5.2	5.3	0	0
5	TimeMeasureEnd	CookingEndTime	distribution	5.3	5.4	0	0
6	TimeMeasureEnd	CookingEndTime	distribution	5.4	5.5	0	0
7	TimeMeasureEnd	CookingEndTime	distribution	5.5	5.6	0	0
8	TimeMeasureEnd	CookingEndTime	distribution	5.6	5.7	0	0
9	TimeMeasureEnd	CookingEndTime	distribution	5.7	5.8	0	0
10	TimeMeasureEnd	CookingEndTime	distribution	5.8	5.9	0.009	0.009
11	TimeMeasureEnd	CookingEndTime	distribution	5.9	6	0.991	1
12	TimeMeasureEnd	CookingEndTime	distribution	6	6.1	0	0
13	TimeMeasureEnd	CookingEndTime	distribution	6.1	6.2	0	0
14	TimeMeasureEnd	CookingEndTime	distribution	6.2	6.3	0	0
15	TimeMeasureEnd	CookingEndTime	distribution	6.3	6.4	0	0
16	TimeMeasureEnd	CookingEndTime	distribution	6.4	6.5	0	0
17	TimeMeasureEnd	CookingEndTime	distribution	6.5	6.6	0	0
18	TimeMeasureEnd	CookingEndTime	distribution	6.6	6.7	0	0
19	TimeMeasureEnd	CookingEndTime	distribution	6.7	6.8	0	0
20	TimeMeasureEnd	CookingEndTime	distribution	6.8	6.9	0	0

Figure 3-79: "histograms_log" view of the AnyLogic Log

resource_pool_utilization_log shows the utilization of each resource pool (average of all units, if more than one in the pool) and the size (Figure 3-80).

resource_pool_utilization_log

	resource_pool	utilization	size
1	DeliveryCrew	0.568	20
2	Operators	0.479	2
3	Ovens	0.383	5
4	PizzaMakers	0.383	10

Figure 3-80: "resource_pool_utilization_log" view of the AnyLogic Log

resource_unit_utilization_log shows the utilization of each resource unit in each resource pool. The values in the "*unit*" column, in the order they appear, are the name of the population (in this case, the default population), the index or numbered position in that population, and an ID AnyLogic automatically assigns to each unit. These values are generally for AnyLogic's internal use and the way it distinguishes units.

Each item under the "*utilization*" column is the utilization of one resource unit, as opposed to the average of all units, such as in the **resource_pool_utilization_log**. If you look at the two operators (lines 21 and 22 in Figure 3-81), there are two operators with utilization of 0.475 and 0.483. Referring back to Figure 3-79, the utilization for their resource pool (**Operators**) is: $\frac{0.475+0.483}{2} = 0.479$.

resource_unit_utilization_log

	unit_type	resource_pool	unit	utilization
1	Agent	DeliveryCrew	<population>[18] : 366	0.6
2	Agent	DeliveryCrew	<population>[17] : 365	0.6
3	Agent	DeliveryCrew	<population>[16] : 364	0.6
4	Agent	DeliveryCrew	<population>[15] : 363	0.6
5	Agent	DeliveryCrew	<population>[21] : 369	0.6
6	Agent	DeliveryCrew	<population>[22] : 370	0.6
7	Agent	DeliveryCrew	<population>[23] : 371	0.6
8	Agent	DeliveryCrew	<population>[24] : 372	0.6
9	Agent	DeliveryCrew	<population>[25] : 373	0.6
10	Agent	DeliveryCrew	<population>[20] : 368	0.6
11	Agent	DeliveryCrew	<population>[26] : 374	0.6
12	Agent	DeliveryCrew	<population>[28] : 376	0.553
13	Agent	DeliveryCrew	<population>[29] : 377	0.513
14	Agent	DeliveryCrew	<population>[30] : 378	0.5
15	Agent	DeliveryCrew	<population>[31] : 379	0.5
16	Agent	DeliveryCrew	<population>[32] : 380	0.5
17	Agent	DeliveryCrew	<population>[33] : 381	0.5
18	Agent	DeliveryCrew	<population>[34] : 382	0.5
19	Agent	DeliveryCrew	<population>[27] : 375	0.593
20	Agent	DeliveryCrew	<population>[19] : 367	0.6
21	Agent	Operators	<population>[36] : 384	0.475
22	Agent	Operators	<population>[35] : 383	0.483
23	Agent	Ovens	<population>[14] : 362	0.383
24	Agent	Ovens	<population>[13] : 361	0.383
25	Agent	Ovens	<population>[12] : 360	0.383
26	Agent	Ovens	<population>[11] : 359	0.383
27	Agent	Ovens	<population>[10] : 358	0.383
28	Agent	PizzaMakers	<population>[9] : 273	0.383
29	Agent	PizzaMakers	<population>[8] : 272	0.383
30	Agent	PizzaMakers	<population>[7] : 271	0.383
31	Agent	PizzaMakers	<population>[6] : 270	0.383
32	Agent	PizzaMakers	<population>[5] : 269	0.383
33	Agent	PizzaMakers	<population>[4] : 268	0.383
34	Agent	PizzaMakers	<population>[3] : 267	0.383
35	Agent	PizzaMakers	<population>[2] : 266	0.383
36	Agent	PizzaMakers	<population>[1] : 265	0.383
37	Agent	PizzaMakers	<population>[0] : 264	0.383

Figure 3-81: "resource_unit_utilization_log" view of the AnyLogic Log

You can run the model in different scenarios and observe their logs. Depending on the blocks and objects you've added to your model, different views will be populated. After each run, AnyLogic will replace the previous logs with information from the new run. These logs can be backed up or exported by using the

relevant settings in the Properties panel shown after clicking the Database entry on the Projects panel. Please refer to AnyLogic Help to find out about how to export tables from your log to Excel.

You should also be aware of the memory and computational burden of logging everything, especially in bigger models. One remedy is to select a log in the Projects view and, on its Properties panel, check the box which states **I do not need this type of log** This will cause the name to become gray in the **Projects** panel and will not be logged to during future runs. You can also hold Control (or Command for Mac OS) or shift to select multiple files; selecting the appropriate checkbox will then cause all to not be logged to.

A second remedy is for cases when you have logging enabled and are not interested in the information from a few selected blocks. For each block, you can select it and navigate to the bottom of its **Properties** panel, then uncheck the box for the entry **Log to database** (Figure 3-82).

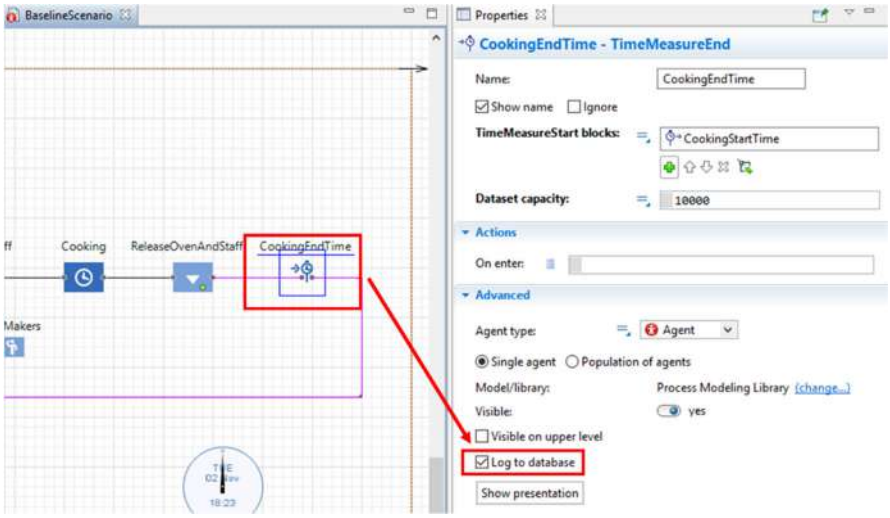


Figure 3-82: Activating or deactivating a block's log

Base (as-is) scenario, Scenario 1, and Scenario 2 with stochastic (random) times

Now, we'll change five parts of our model to make it random: arrival times of phone orders (a non-stationary Poisson process), the time to take an order, the time at which customers waiting on the phone hang up, cooking time of pizzas, and delivery times.

To perform these changes:

1. Make a copy of the deterministic version of the model. From **File** menu, select **Save as…** and save the model with a different name to keep the original model intact.
2. Open the **BaselineScenario** agent and make the changes shown in Figure 3-83 below.
3. Repeat this process for **Scenario1** and **Scenario2**.

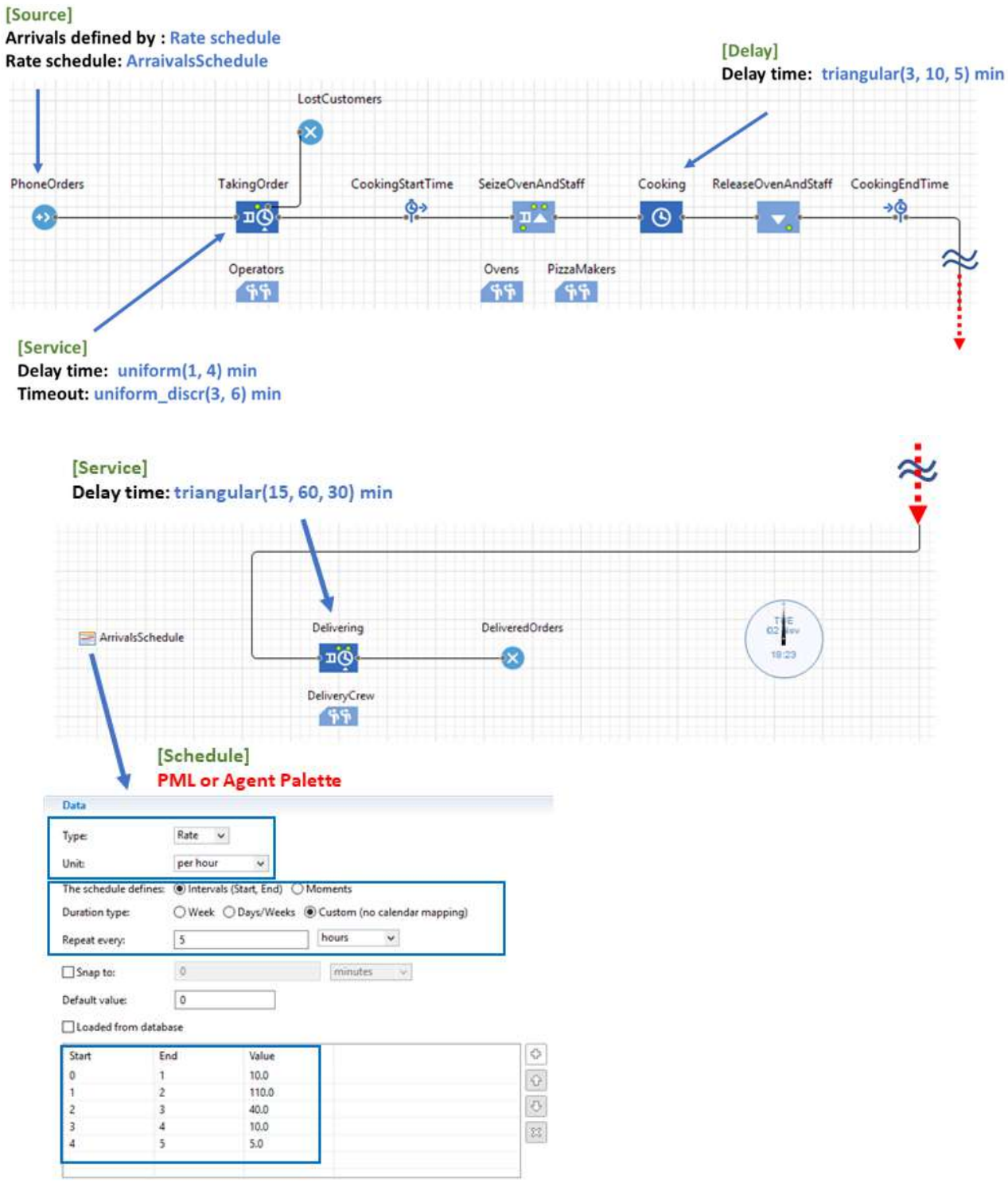


Figure 3-83: Changes to the Pizzaria operation flow chart (baseline, scenario1 & 2) with stochastic time values; note the "~" shows a continuation

4. To make each run unique, we must use a random seed. Click each simulation experiment (**Simulation: BaselineScenario**, **Simulation: Scenario1**, and **Simulation: Scenario2**) and in their **Properties** window, under the **Randomness** section, select the option for **random seed**.

5. Run each scenario and observe their differences. Figure 3-84 and Table 3-10 below show sample realizations of a single run. For each scenario, you find similar outputs across multiple runs. However, any two runs will almost never be the same due to the randomness between them.

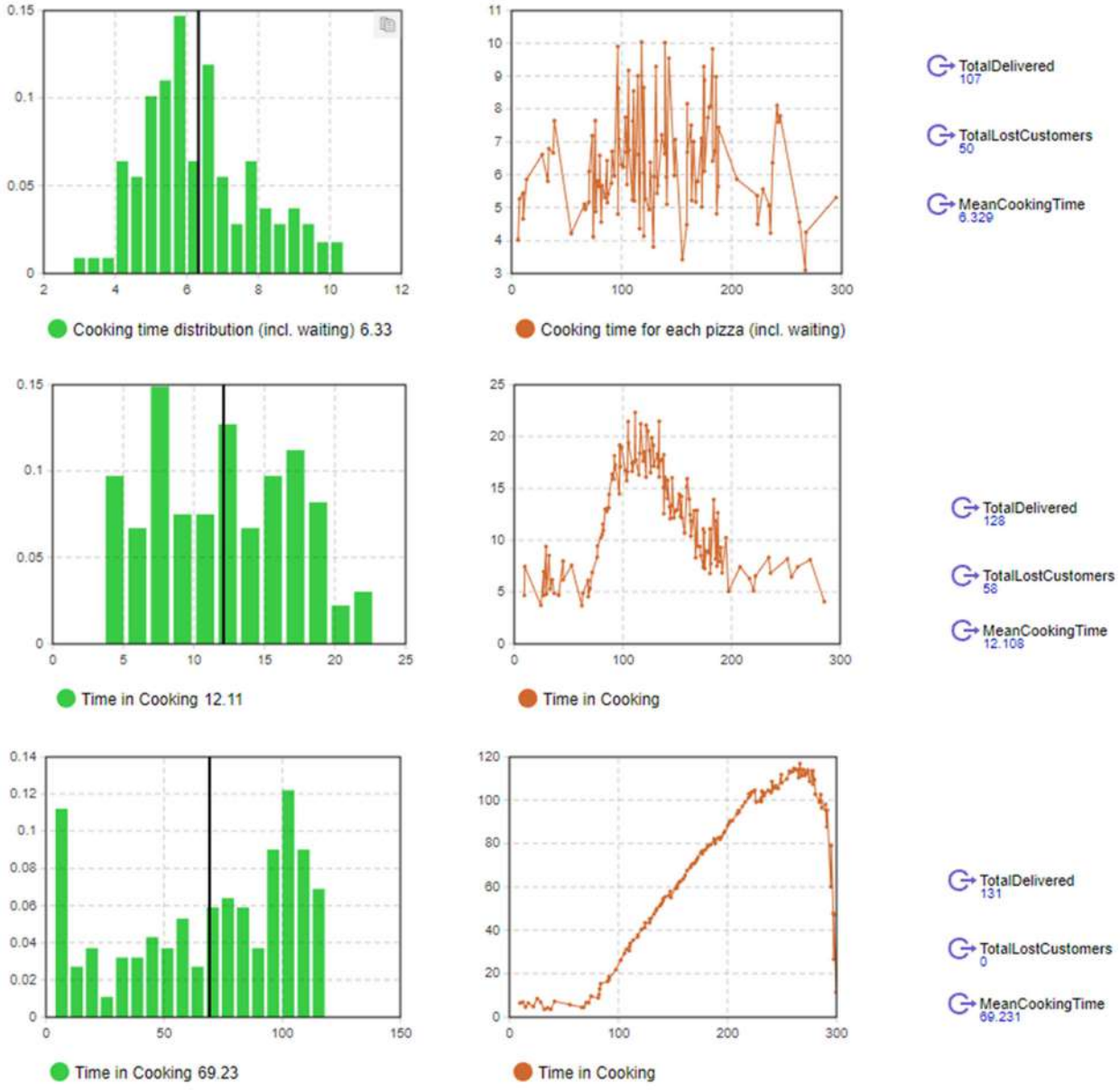


Figure 3-84: Statistical output of single runs from each of the three scenarios (stochastic)

Table 3-10: Outputs of single runs from each of the three scenarios (stochastic)

Scenario	Total Delivered	Total Losts Customers	Mean Cooking Time (min)
Baseline	107	50	6.33
Scenario 1	128	58	12.11
Scenario 2	131	0	69.23

We won't discuss the formal and proper statistical methods of comparing these stochastic scenarios in this chapter. However, we can simply run each scenario several times and compare the averages of their output.

1st order Monte Carlo experiment for the base scenario, Scenario 1, and Scenario 2

To better understand each scenario's outputs and have (relatively) stable values for comparison, we'll run a 1st order Monte Carlo experiment. In these experiments, inputs such as the number of pizza makers or operators are constant. This means the variability of the outputs comes from each process' internal stochasticity (time needed for cooking, number of arrivals per hour, etc.).

AnyLogic Professional has a dedicated Monte Carlo experiment (1st and 2nd order Monte Carlo). Since we can't access it in the PLE version, we'll use the Parameter Variation experiment with fixed input values but with several replications, which are simply several simulations with the same exact input values, repeatedly executed for a set number of times. Replications are usually needed in some experiments (for example, Optimization) and we'll discuss their use later.

By running a Parameter Variation experiment with fixed input but several replications, we'll perform a 1st order Monte Carlo experiment. We'll build three Parameter Variation experiments for the three scenarios.

To do this, you must:

1. Right-click the model's name, select **New**, and select **Experiment** (as shown in Figure 3-85).

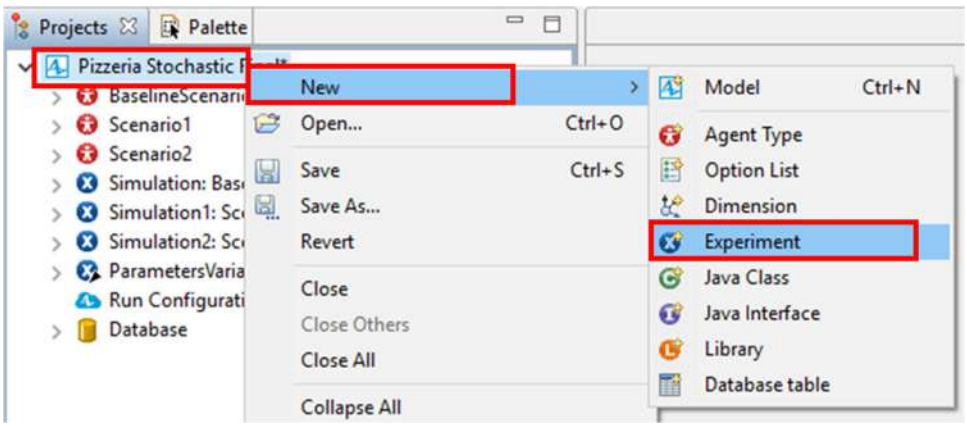


Figure 3-85: Building a new experiment

2. In the **New Experiment** dialog box, do the following:
    a. In the **Experiment Type** box, select **Parameter Variation**.
    b. In **Top-level agent**, select **BaselineScenario**.
    c. In **Copy model time settings from**, select **Simulation**.

       This allows the new experiment to copy its time settings from **Simulation**, the experiment related to the **BaselineScenario** (Figure 3-86).
    d. Click **Finish**.

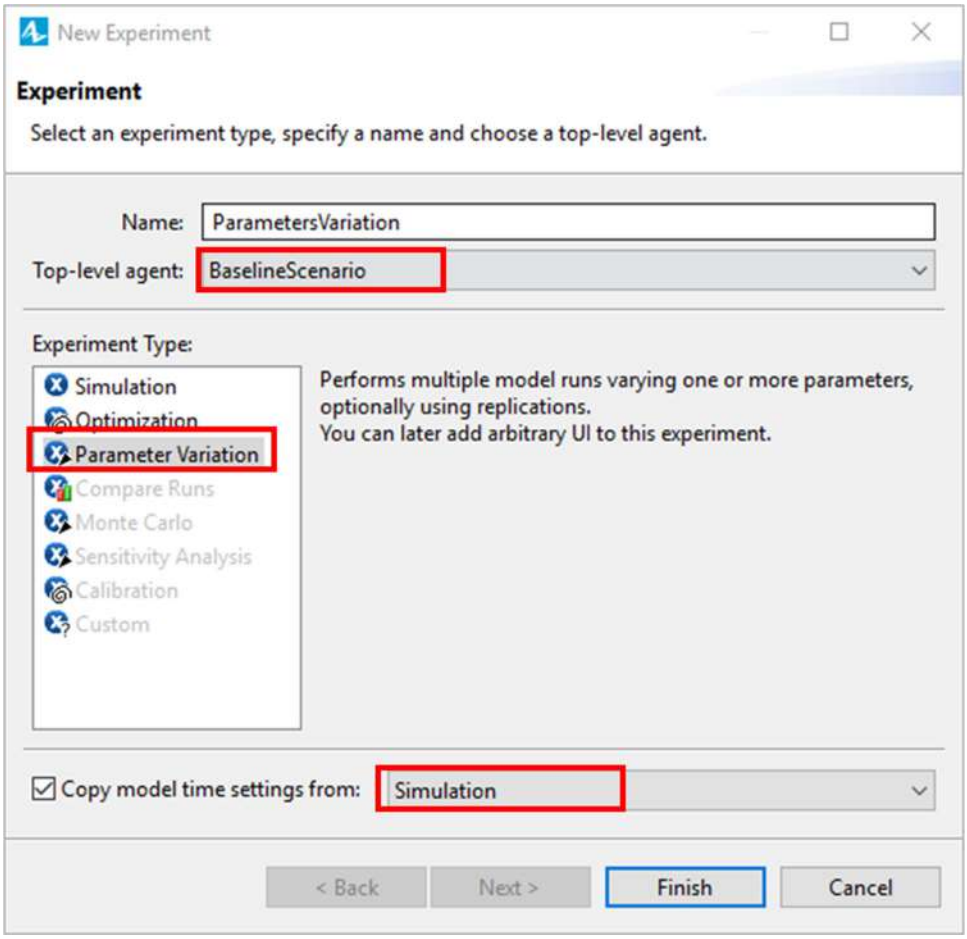


Figure 3-86: Initial setup of a new Parameter Variation experiment

3. AnyLogic adds this experiment to your **Projects** view and opens an empty graphical editor. If it doesn't, double-click the experiment's name to open it (Figure 3-87).

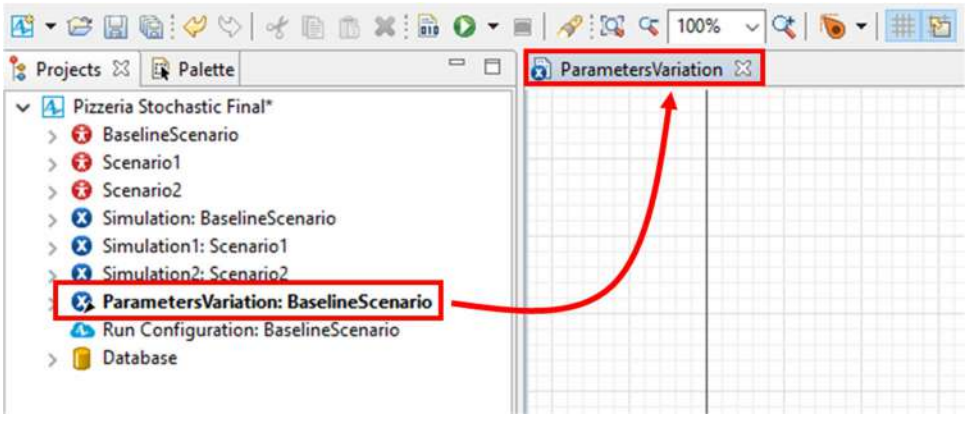


Figure 3-87: Opening the experiment's graphical editor by double-clicking it under the Projects window

4. From the analysis palette, drag and drop three **Histogram Data** objects and then rename them according to the top half of Figure 3-88.
5. Do one of the following to add three **Histogram** [Chart] objects:
    - Drag and drop them from the analysis palette
    - Right-click each **Histogram Data** object and select **Create Chart**.

    Modify their properties to match the bottom half of Figure 3-88.

    By doing it the second way, AnyLogic will automatically connect the generated chart to the **Histogram Data** object you right-clicked.

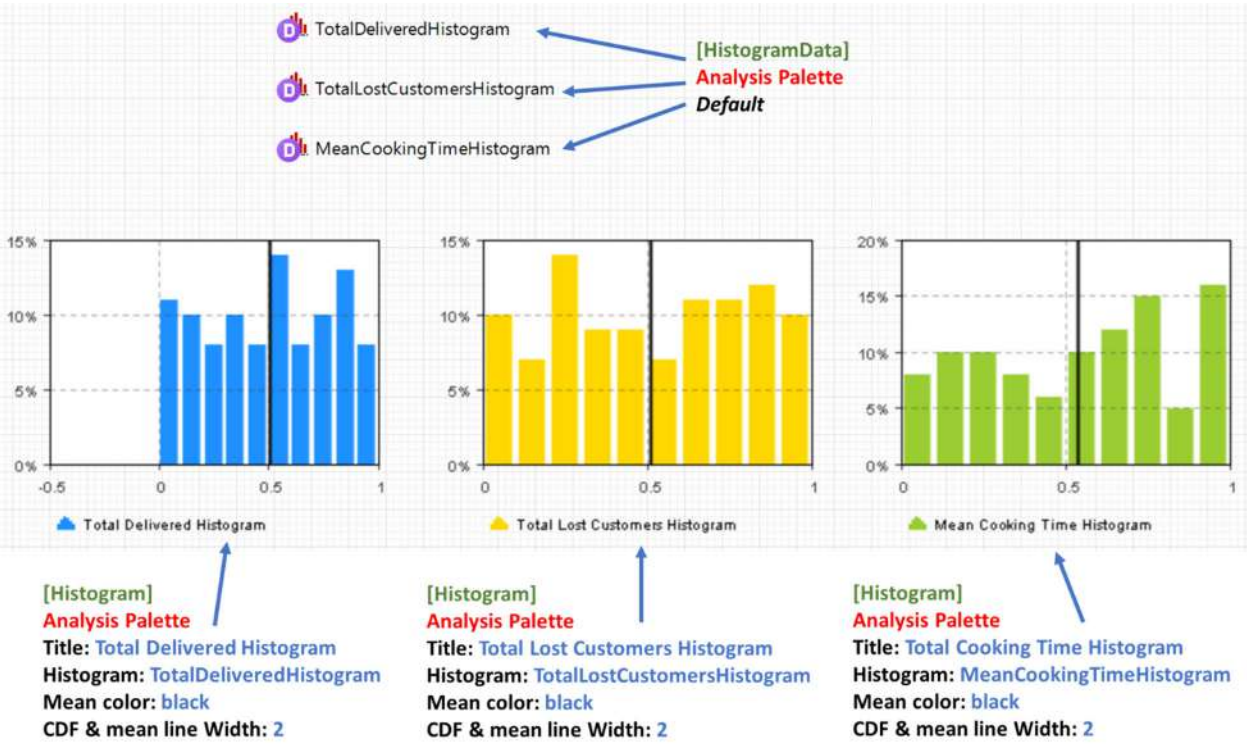


Figure 3-88: Histogram data and histogram charts added to the experiment

If you click the **ParametersVariation** experiment in the Project view or click any blank space in its graphical editor, you can use the **Properties** panel to review the experiment's current settings. You'll see the model stop time (300) and the seed selection (random) copied from the simulation experiment.

6. In the Replications section, check the **Use Replications** checkbox and enter 100 in the **Replications per iteration** box (Figure 3-89).

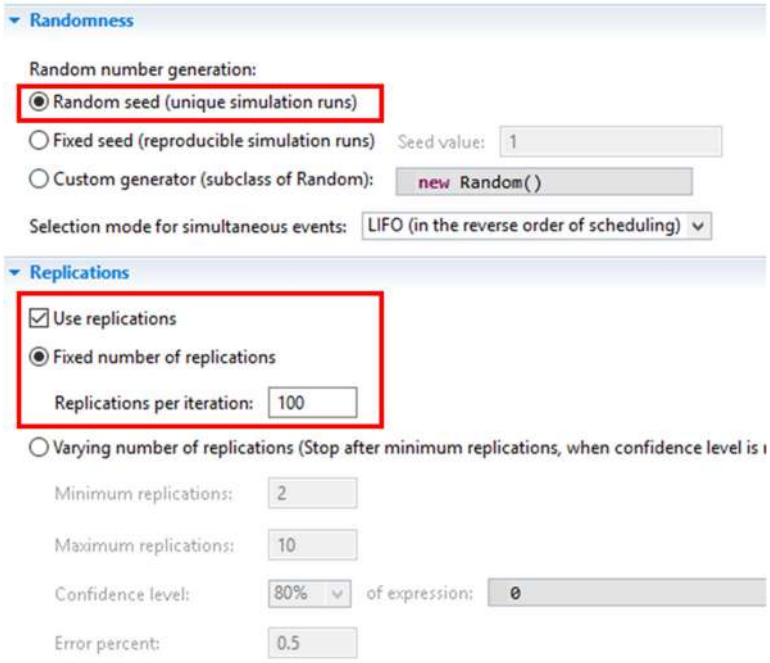


Figure 3-89: Randomness and Replication setting of the parameter variation experiment

7. In the experiment's **Java actions** section, add the three lines of code shown in Figure 3-90.

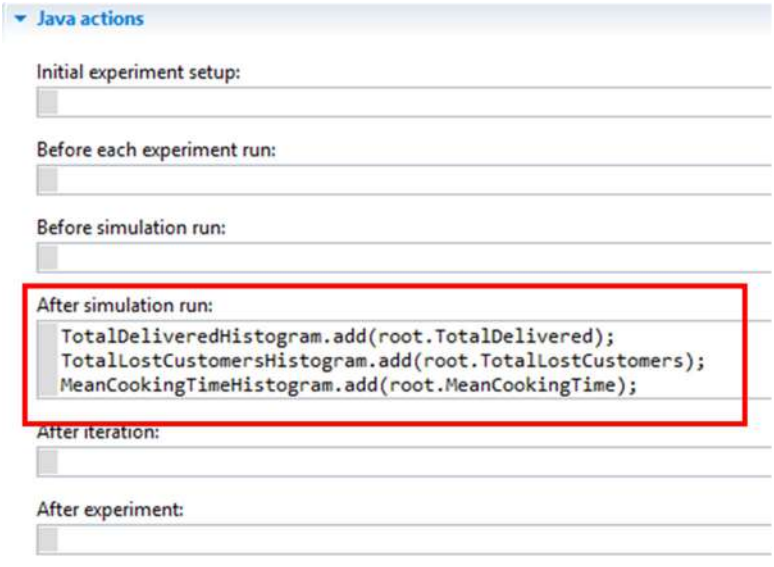


Figure 3-90: Adding the output of each simulation run (each replication) to the histogram data object in the parameter variation experiment

Each time the simulation finishes running (which is after each replication), this code takes the three scaler outputs and adds them to their respective **Histogram Data** objects within the parameter variation experiment. However, **Output** objects are in the top-level agent, not the current experiment; we resolve this by using the AnyLogic keyword "root" to reference the top-level agent.

For example, refer to the first line shown in Figure 3-90 above. We first reference the Histogram Data object, **TotalDeliveredHistogram**, by name. We then call this object's add function which allows us to append a value into the histogram's dataset. In this case (first line), we want to add the resulting value of the total number of pizzas delivered, which is the top-level agent's **TotalDelivered Output** object.

If you look at the top of the experiment **Properties** window (Figure 3-91), you'll see it says the top-level agent is the **BaselineScenario**, which is what is referenced by the "root" keyword.

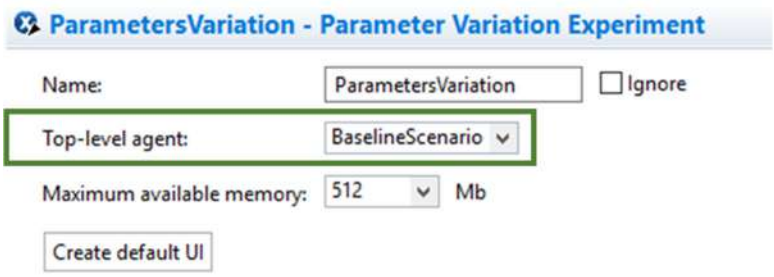


Figure 3-91: Top-level agent setting of the experiment

100 simulation runs will occur in each run of the parameter variation experiment. The results (**Output** objects) of these runs are added to the datasets within parameter variation experiment (**HistogramData**). The code which does this is added to the **after simulation run** field from the Java actions section. It means each simulation (1 to 100) should run until completion, then add its result by the code to the **Histogram Data** object in the **Parameter Variation** experiment. This is visually described in Figure 3-92 below.

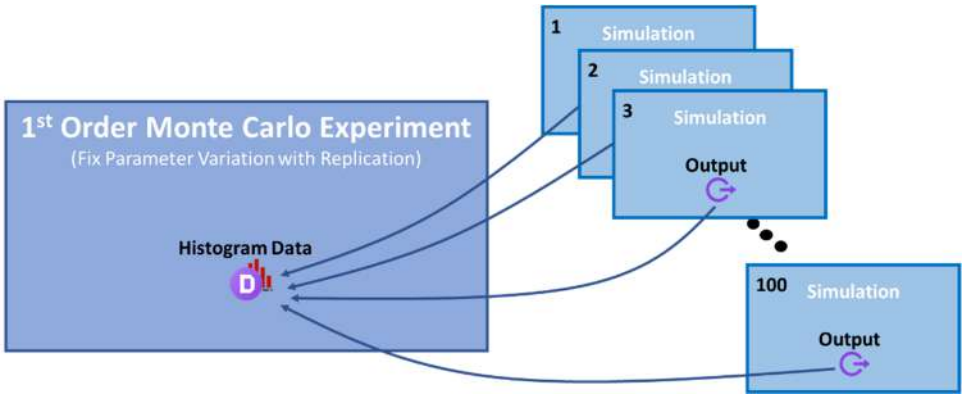


Figure 3-92: Illustration of how the outputs of each replication (each replication is one simulation run) is added to the Histogram data in the Parameter Variation experiment

8. Build two more Parameter Variation experiments for Scenario 1 & 2 by right-clicking the **ParametersVariation: BaselineScenario** and selecting copy. Then, click the model name under the **Projects** view, right-click and paste two copies of the experiment (paste it twice).

   AnyLogic will automatically build separate variation experiments named **ParameterVariation1: BaselineScenario** and **ParameterVariation2: BaselineScenario**. As you see, AnyLogic added a number 1 (and 2) to the end of the experiment name to distinguish it from the original. Click one

of the newly copied experiments and change its **Top-level agent** to **Scenario1** (Figure 3-93); do the same for the second pasted experiment but changing the top-level agent to be the second scenario.

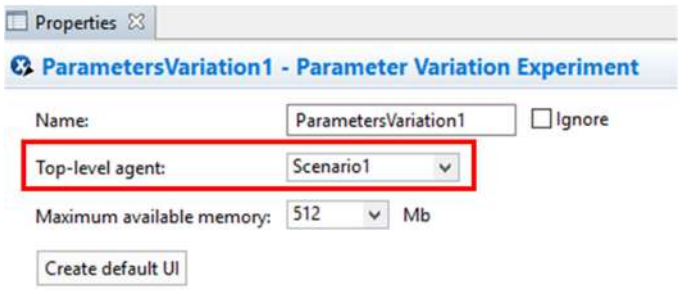


Figure 3-93: Changing the Top-level agent

By this point, you should have three Parameter Variation experiments that run the Baseline, Scenario1, and Scenario2, each for 100 times each (1st order Monte Carlo experiment).

9. Run and observe the outputs for each of the three Parameter Variation experiments (1st order Monte Carlo experiment) for BaseScenario, Scenario1, and Scenario2. Figure 3-94 and Table 3-11 show sample realizations.

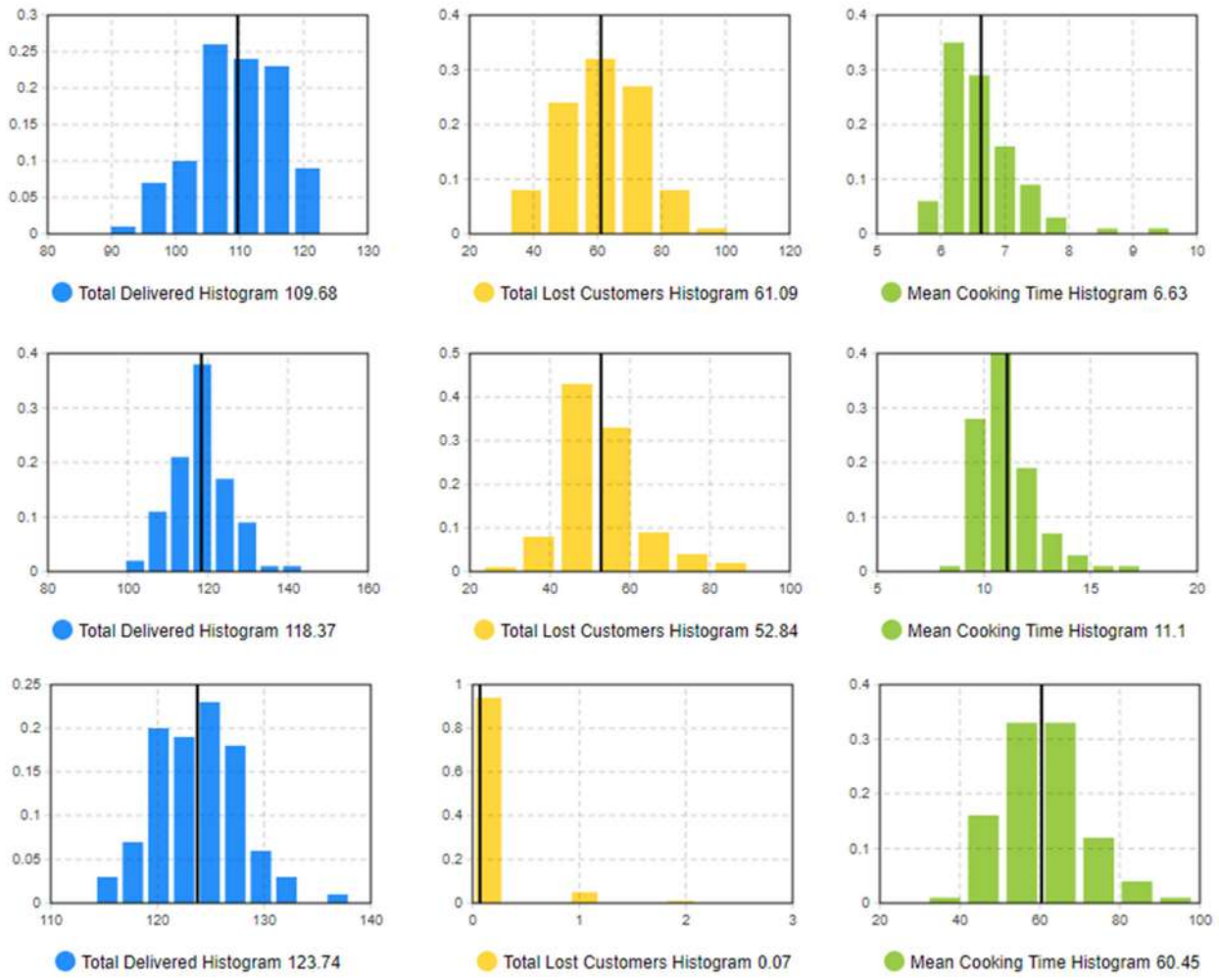


Figure 3-94: Monte Carlo experiment output of the three scenarios

Table 3-11: Mean values of 100 replications of the three scenarios

Scenario	Total Delivered (mean of 100 runs)	Total Losts Customers (mean of 100 runs)	Mean Cooking Time (min) (mean of 100 runs)
Baseline	109.68	61.09	6.63
Scenario 1	118.37	52.84	11.1
Scenario 2	123.74	0.07	60.45

Although the outputted values for the three scenarios are the means from 100 runs – which are closer to the expected values than running each scenario one time - we didn't ensure they're statistically representative. If you run these experiments again, you may get different values. You could increase the likelihood you have stable mean values by running the Parameter Variation experiments with a higher number of replications (for example., 1000). However, determining the proper number of replications is outside of our scope.

For now, we'll use the results of our 100 replications to compare the scenarios. The results in Table 3-11 shows assigning pizza makers to answer calls only if there's no order to be cooked (Scenario1) doesn't have a large impact on the number of lost customers.

However, when pizza makers give a higher priority to taking orders (Scenario 2), almost no customers are lost. A by-product of this is that the average cooking time has increased significantly (more than 9 times) compared to the base scenario. This tells us giving higher priority to order taking reduced our pizza makers ability to cook orders.

It's clear there's a trade-off between taking more orders and the time it takes to complete them. In contrast, the proposed scenarios haven't increased the number of delivered orders. This tells us our decision to take more orders and cook more pizzas shifted the bottleneck to the delivery crew.

Like most real-life situations, there will always a trade-off between different objectives (for example, , maximizing revenue, maximizing service quality, minimizing waiting time, and minimizing overhead). What's most important to remember? Our simulation models should be designed in way that cover important aspects of the operation and produce performance metrics decision makers will find useful in finding better solutions.

Building animation for process-centric models

AnyLogic's Process Modeling Library (PML) can create sophisticated animations of process models. To better understand how we animate entities and resource units in a process, we'll first categorize the animating procedure into four main use types:

I. **No animation:** This model type neither has a physical level movement nor any illustrative animation. It only includes blocks and other non-animation related elements.

II. **Illustration only**: This type of animation exists only to add an illustrative visualization for the process and doesn't affect the model definition or its quantitative outputs. This type of animation applies to process (or subsection of processes) that don't include explicit modeling of entities or resource units' movements at the physical level.

III. **Physical movements are part of model definition**: Beside visualizing the process, these animations exist as part of the model definition by contributing to the calculations related to physical movements of entities and recourse units. In other words, some of the model definition is delegated to spatial components.

For example, we base the process that uses forklifts to unload inbound trucks on the physical paths we see in a warehouse. In these physical level models, the model's animated components significantly contribute to its quantitative outputs. We use this type of animation when we should model physical movement of entities or resource units explicitly.

IV. **Combination of Type II and III**: In these models, we model some part of the process for illustrative purposes that doesn't have anything to do with physical movement (Type II), and some parts are part of the model definition and contribute to the model results (Type III).

One of AnyLogic's unique features is the design of its animating procedure; the program separates the components that represent the model's logic and physical space (or illustrations). In the PML, the elements in the **Blocks** section define the model's logic and the elements in the **Space Markup** section define the model animation (Figure 3-95).

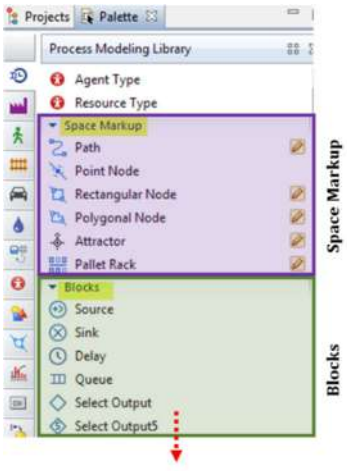


Figure 3-95: Space Markup and (logical) Blocks in the PML

The Space Markup elements allow us to animate process models. Before we explain how you can build animations, let's discuss how we draw the space markup elements and connect them to form networks.

Space markups of PML

Space Markup elements define the locations of entities and resource units within your model's environment. We can categorize these elements into three main categories:

- **Nodes** define the places in the environment where entities or resource units can reside or move toward (destination of a movement).
- **Paths** define the routes between nodes.
- **Network** is a set of interconnected nodes via paths. Unlike nodes and paths which the user draws, AnyLogic builds networks automatically. This means you won't find a network element in the PML's **Space Markup** section.

Node

The PML has three types of nodes: **Point**, **Rectangular**, and **Polygonal** (Figure 3-96).

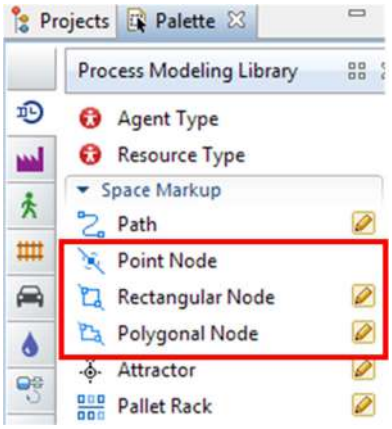


Figure 3-96: Point, Rectangular, and Polygonal Nodes (space markup section of PML)

- **Point Node**: Modelers use this type of node as a single location to hold entities (or resource units). This means it's usually a destination point in a network. Since a point node can hold more than one entity or unit, multiple entities or resource units set to a single point node will have their animation representations rest on top of one another. Having them all at the same location in the 2D space means you'll only see the animation of the one on top.
    - To draw a Point node, drag it from the PML and drop it on to the graphical editor.
- **Rectangular node**: We use this type of node to draw rectangular areas that allow one or many entities/resource units to use them as their location in the 2D space.
    - To draw a rectangular node, open the PML library and drag it from the PML library on to the graphical editor. After you've drawn the default rectangular node, you can use the handles to resize it. As a preferred alternative when you see a pencil icon to the element's right, double-click the entry. In the resulting drawing mode, you can draw a rectangle by holding the left-mouse button down, dragging your mouse to the desired lower right corner position and then releasing the button (Figure *3-97*).

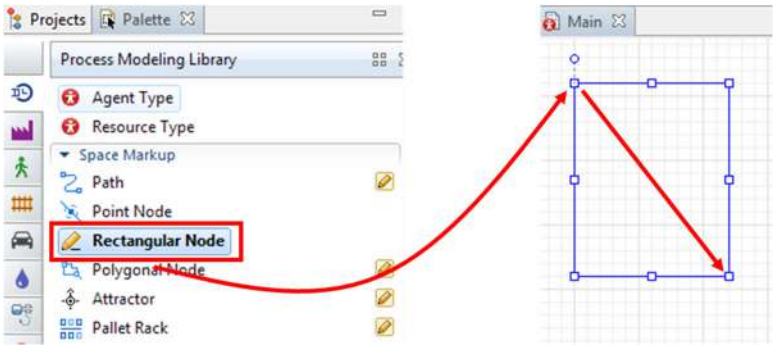


Figure 3-97: Drawing a rectangular node (in drawing mode)

Since a Rectangular Node is an area, we can specify the exact location of entities or resource units that reside inside it. You have three options for this setting: random layout, arranged layout, or attractors. You can select one of them by selecting the rectangular node object, opening the **Properties** panel, and choosing the **Locations** layout from the dropdown box (Figure 3-98). No matter how you set the layout of positions inside the node, there's no limit on the number of entities or resource units AnyLogic can put inside a Rectangular node.

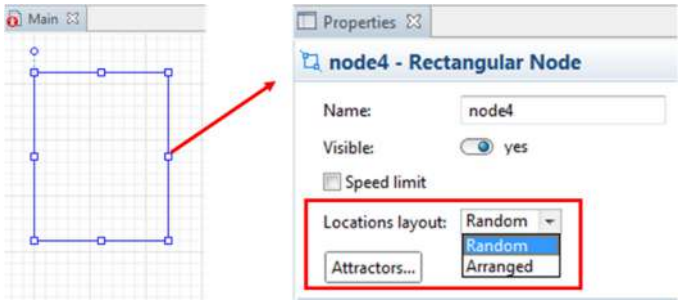


Figure 3-98: Selecting a specific layout for the rectangular object

i. **Random:** Animations of entities or resource units will reside at random positions inside the node (Figure *3-100*).

ii. **Arranged:** Animations of entities or resource units display arranged in an invisible grid. In most cases, the associated logical block's capacity dictates the number of positions (Figure *3-100*).

iii. **Attractors:** Anylogic uses on the positions of Attractor objects to display the animations of entities or resource units.

You can automatically generate multiple attractors by clicking the **Attractors…** button (Figure *3-99*) and then specify the number of attractors you want. The dialog box that opens offers three options (Figure *3-99*): explicitly specifying the **Number of attractors**, filling the node with attractors while preserving a specified **Space between attractors**, or placing them in a **Grid** with a set number of rows and columns.

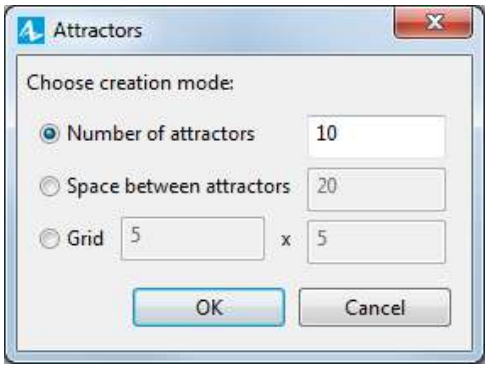


Figure 3-99: Dialog box for automatic generation of attractors inside a node

Alternatively, you can manually add the attractors into the rectangular node by drag-and-drop Attractor objects into the rectangular node. If you want to add several attractors manually, you can double-click the Attractor element to enable drawing mode which lets you place several attractors by clicking any location inside the rectangular node.

You should also know that beside specifying an exact position inside a rectangular (or polygonal) node, you can also use attractors to set the entity or unit's orientation at that position. No matter how you generate the attractors (automatically or manually), you can click each attractor to move or rotate.

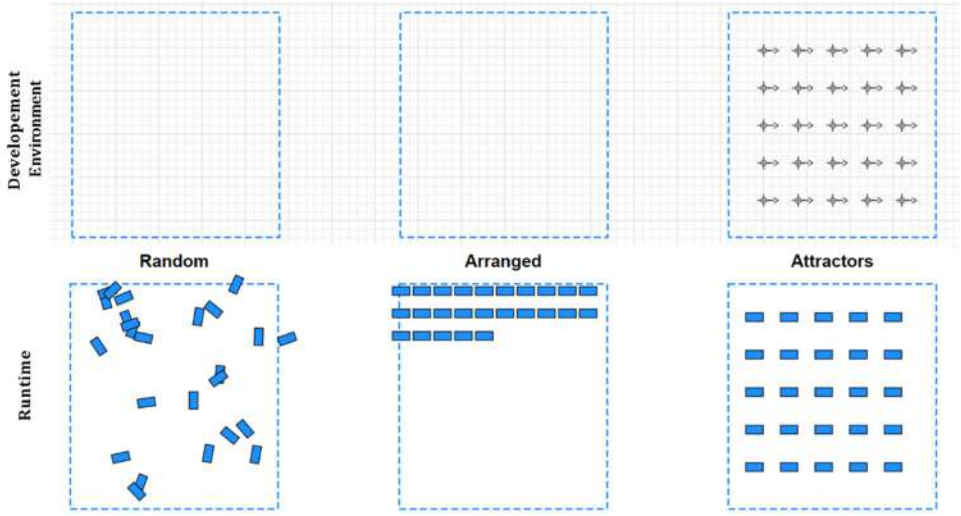


Figure 3-100: Entities in three rectangular nodes with different layout settings

- **Polygonal node**: All Polygonal node features, use cases, and attributes are identical to the Rectangular node. The only difference is you can draw more complex areas (shapes) with this type of node.

Path

Path elements define routes between nodes. The primary use is to define and display the movements of entity or resource units in Type III and IV animations. They represent the physical paths between nodes and their length (in combination with the entity or resource unit's speed) specifies the time it takes to

move between them. When an entity or resource unit moves along a path, it's automatically oriented to align with the path.

It isn't common, but you can use a path for illustrative animation. When we use a path for the location of entities in a **Queue** block (or other blocks containing a **Queue** inside such as **Seize**, **Service**, **Batch**, or **Match**), the entities stay on the path and don't move along it. Here, the entity's position on a path depends on its position in the **Queue**. AnyLogic displays the entity at the queue's head (index 0) at the path's endpoint. The distance between the two subsequent entities is set to the path length divided by one less of the **Queue**'s capacity. If your model animation involves actual physical movement, we recommend you use Nodes rather than paths as the entity's location.

When we assign the entities inside a **Delay** block to a path, each entity appears at the start of the path and then moves along it. AnyLogic adjusts the entity's speed in way that forces it to reach the end of the path at the same time its delay time finishes. In other words, the **Delay** animation behaves like a **MoveTo** block, but its action reflects a travel time rather than the entity's speed.

To draw a path, double-click the PML's **Path** element to enable the drawing mode. Each subsequent left-click adds a point in the graphical editor, and you can double-click to add the last point. Paths are by default bidirectional, but you can uncheck the **Bidirectional** checkbox from their **Properties** windows to make them unidirectional. A unidirectional path has arrowheads which show their current direction. If you want to change the direction, right-click the path and select **Change Direction**. Whenever you connect a path to a point in the middle of another path, AnyLogic automatically adds a point at the intersection. If you connect a path to any two ends of another path, the program automatically merges the two paths.

Network

As we mentioned, AnyLogic automatically builds a Network from interconnected Node and Path elements. Starting with AnyLogic 8.4, single Path markups are also assigned to a generated Network, but we still can have isolated nodes that don't belong to any network. In general, depending on the number of groups of paths and nodes (or isolated paths), you could have multiple networks in one environment.

To see the generated networks, go to the **Projects** panel, expand the agent type that contains the networks (for example, Main), and expand the **Presentation** field. You'll see the networks and other space markup elements. When you click a Node or Path object that is part of a network, your first left-click will select that space markup element; a second left-click will select the network. If you select the network, you can configure the visibility or Z level of the network and all its elements.

While AnyLogic automatically generates networks for all animation types (I, II,III, or IV), it only uses them in models where physical level movement is part of the model definition (Type III or IV). In these cases, the moving entity or resource unit considers the full network. In other words, the entity or resource unit scans the entire network between its current node and the destination node to find the shortest path.

In Type I models (animation is illustrative) we only show the entities or resource units when they're inside specific blocks; AnyLogic will automatically generate a network even when your model won't use it. Besides the network topology, each moving entity or resource unit has a speed that can affect the time it takes to complete the movement. In the PML, it's assumed segments have unlimited capacity and entities or resource units that move along a segment aren't aware of one another. Said another way, they can pass through one another. For spatially aware movements such as AGVs, you should use the **MATERIAL HANDLING LIBRARY'S (MHL)** transporters.

Defining scale for animation of an environment

AnyLogic's graphical interface defines length in *pixels*. If you zoom out in the active graphical editor (for example, **Main**), you'll see a blue frame and a **Scale** object above it (Figure 3-101).

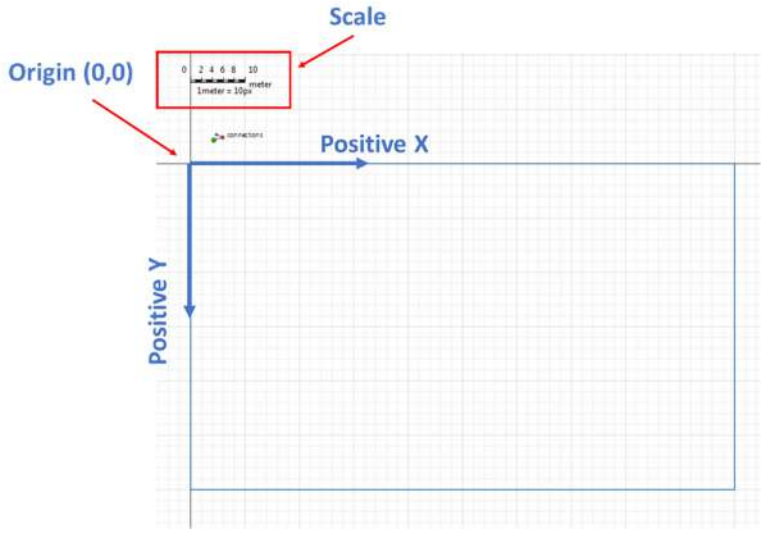


Figure 3-101: Graphical editor of Main (or any other Agent type)

The environment's origin lies at the intersection of the horizonal (X axis) and vertical line (Y axis). By default, the blue frame's upper-left corner is also at the origin. It's important to note the positive direction of the Y-axis is downward. The status bar (below the graphical editor), shows the current **Scale**'s setting and your cursor's current position in pixels (Figure 3-102).

Figure 3-102: Current setting of the Scale and current position of the cursor

Each markup element (Path, Point Node, Rectangular Node, Polygonal Node) has a **Position [and Size]** section that shows the current location of the space markup in the graphical editor (Figure *3-103*). There might be other spatial information, such as the radius for the Point Node or the coordinates of each point in a Path). Since this information is in pixels, it may be hard to convert to the corresponding physical length or coordination.

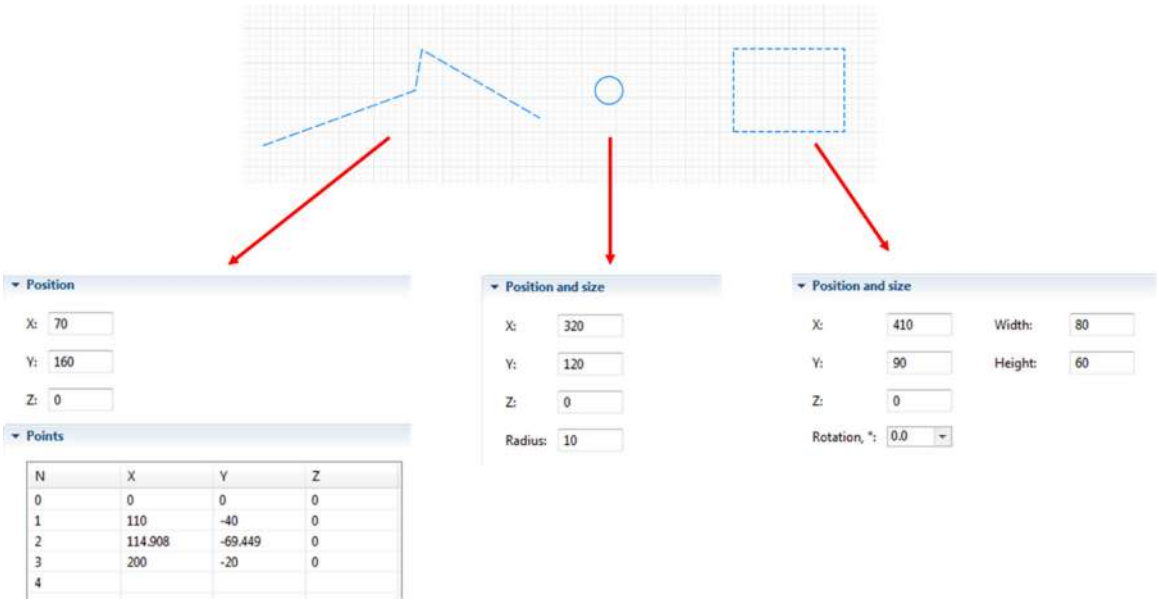


Figure 3-103: Position, Point, and Position and Size sections in Properties window of Space Markup elements

When you use the Space Markup elements to draw nodes and paths, you can define a scale, which is the number of pixels per unit of length. The graphical editor in each model's building environment (for example, Main) has a **Scale** on top of its blue frame (Figure 3-101).

While you can set the scale explicitly, we recommend you use a background layout and a scale you define graphically based on a reference in the layout to draw space markups. You can use the Presentation palette's **Image** or **CAD Drawing** objects to add the model's background. To scale the environment graphically, your background layout should have a reference line with a known length. You should then click the **Scale** object, move it close to the reference line and resize it by dragging one of its ends to match the reference length. Finally, you can assign the reference's actual length to the **Scale** object in its **Ruler length corresponds to** field (Figure 3-104). Afterward, any space markup you draw on top the layout will have a correctly-scaled length and location.

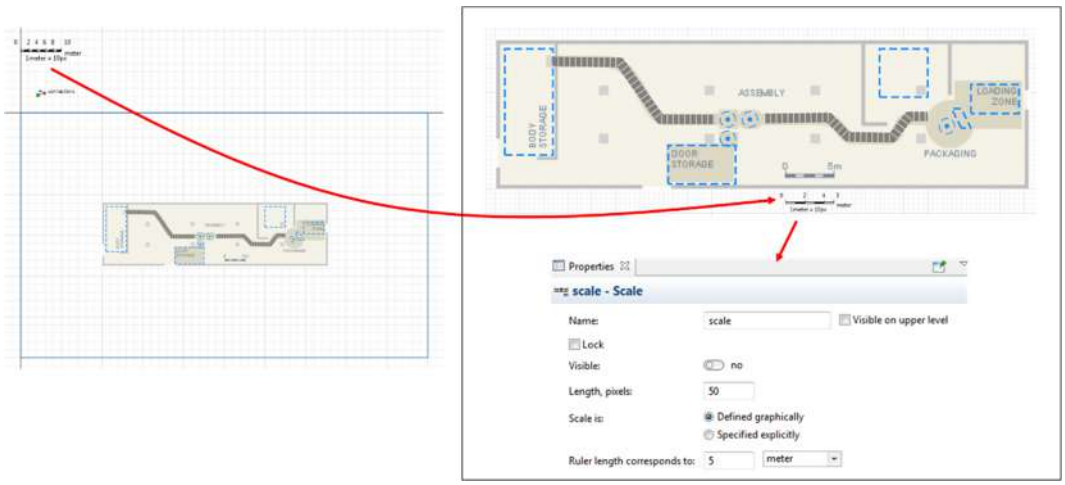


Figure 3-104: Moving the Scale object closer to the reference length on the Image, resizing it and set its corresponding length

Steps needed to build animation for a process-centric model (Type II, III, or IV)

Now that we know about different elements of PML space markups, we can review (at high-level) the steps you take to build an animation for entities and resource units in a process-centric model:

1. In most real process-centric projects, physical movements occur in a large and intricate environment such as a warehouse, factory or hospital. In these cases, we usually import an image or a CAD file that represents the facility's layout and draw the space markups on top of it. We use the imported plan in the background as a guide to help us draw our space markups. As we mentioned, it makes sense to use the background layout and the **Scale** object to set the correct scale for the environment. If you don't have an image or CAD file of the layout, you can start drawing the space markups without a background layout.

2. Using the PML or Space Markup palette's space markup elements, we draw the nodes and paths; networks are generated if we have interconnected nodes and paths. Note the space markup elements in the PML and from the Space Markup palette are the same; the only difference is the Space Markup palette's space markup elements are more comprehensive and entail all the other types of space markups available in AnyLogic (for example, ones used for GIS, Pedestrian, Rail, Road, Material Handling, and Fluid libraries for models that have physical movements [Type III or IV]).

3. Associate your logical blocks of PML (the flowchart blocks) with the space markup elements through the options in the Properties panel.

Associating the PML blocks with space markup elements

As mentioned in step 3 of the previous section, we associate PML logical blocks with the drawn space markups in the graphical editor. Before we discuss how different types of models (Type I, II, III or IV) are animated, we must understand that different PML blocks have different animating features depending on their functionality.

Regarding animation, PML blocks can be categorized into four categories:

- **Type A**: Blocks that build new entities or resource units and add them to the environment
- **Type B**: Blocks that set the location of contained entities
- **Type C**: Blocks that set the destination of moving entities or resource units
- **Type D**: Blocks that cannot keep an entity (or resource unit)

A) Blocks that are building new entities or resource units and adding them to the environment

These blocks shown in Table 3-12 have settings in their properties that allow you to set an entity or resource unit's first (initial) location.

Table 3-12: Blocks that assign animation location to entities or resource units at the time they're added to the model

Block name	PML Icon	Setting	Description
Source		Location of arrival	The location of newly generated entities
Enter		New location	The new location for the existing entities that entered the flowchart via this block

Split		Location of copy	Where to place the new entity (that is, the copy)
Batch		Location of batch	Where the new batched entity will be placed
Assembler		Location of assembled agent	Where the new assembled entity will be placed
Combine		Agent location (combined)	The location of the combined entity while it hasn't yet been consumed by the subsequent block. This setting is only active if the Combine block builds a new entity. It won't be available if the output of the combine block is one of the original inputs.
ResourcePool		Home location (nodes)	The nodes that will be used as the home location for the resource units in this pool

Depending on the block, the options available inside the **Properties** view are different. For example, in the **Location of arrival** field of a **Source** block, you can select a Node, an Attractor, x-y-z coordinate, GIS Point, longitude-latitude, or the name of a geographical place. In contrast, in the **Home location** filed of a **ResourcePool**, you can only select Node objects as the home location of newly built resource units. Table 3-13 shows all options for Type A blocks and Properties we discussed in Table 3-1.

Table 3-13: Options available in the animation related fields of Type A blocks

Options	Source	Enter	Split	Batch	Assembler	Combine	ResourcePool
Not specified	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Node *(with or without attractors)*	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Attractor *(inside a Node)*	Yes	Yes	Yes	Yes	Yes	No	No
Path	No	No	No	No	No	Yes	No
x, y, z	Yes	Yes	Yes	Yes	Yes	No	No
Network / GIS node *(in GIS Map)*	Yes	Yes	Yes	Yes	Yes	No	No
latitude, longitude *(in GIS Map)*	Yes	Yes	Yes	Yes	Yes	No	No
Geographical place *(in GIS Map)*	Yes	Yes	Yes	Yes	Yes	No	No
Not specified: No arrival place for entities are specified, and thus not displayed. This is the default setting and is chosen in models of Type I (with no animation) or Type II (only illustrative animation).							

Node *(with or without attractors):* A node (Point, Rectangular, or Polygonal) in the environment; if the selected node is Rectangular or Polygonal and has attractors, those attractors specify the exact location after the entity enters the node.

Attractor *(inside a Node):* Entities are assigned to a *specific* attractor inside a Rectangular or Polygonal node; this is in contrast to the "Node" option in which assignment to attractors are automatic.

Path: Selecting a Path is only available for the **Combine** block; since only one entity can stay there, it will be at the end point of the path.

x, y, z: The new (or newly entered) entities will appear in the point with the specified coordinates (in pixels).

Network / GIS Node *(in GIS Map)*: Entities appear in the given GIS point or GIS region. This option is only available if the environment contains a GIS map and there are existing GIS points or regions defined inside of it.

latitude, longitude *(in GIS Map)*: Entities appear in the given point on the GIS map with the specified latitude and longitude. This option is only available if the environment contains a GIS map.

Geographical place *(in GIS Map):* Entities appear in a named location on the GIS map. AnyLogic puts the entity on the first result of map search for the name provided in this field. This option is only available if the environment contains a GIS map.

B) Blocks that set the location of contained entities

These blocks assign a location to the entities inside them and are listed in Table 3-14. In almost all cases, for type B blocks it's a better design to treat their assigned location as a temporary illustration of the entities inside the block, and then restore the entity's previous (real, physical) location when it leaves the block. AnyLogic restores the location by default by enabling the checkbox for the **Restore agent location on exit** setting in the properties of type B blocks. This ensures the entity or resource unit will treat its location inside a Type B block as temporary and illustrative; it then restores its actual physical location upon leaving the block. Clearing the **Restore agent location on exit** checkbox is equivalent of forcing the entity or unit to jump (at zero time) to the location set in the Type B block's **Agent location** field.

To clarify, the **Combine**, **Batch**, and **Assembler** blocks have two types of location fields. The first sets the first location (Type A) and the second sets the location while the entity resides in them (Type B). This is why we showed them under both types in Table 3-12 *and* Table 3-14.

Table 3-14: Blocks that set the location of contained entities

Block name	PML Icon	Setting	Description
Delay		Agent location	The location of entities while being delayed
Queue		Agent location	The location of entities while in the queue
Seize		Agent location	The location of entities while in the embedded queue

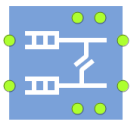
Service		Agent location (queue), Agent location (delay)	The location of entities while being in the embedded queue or delay
Match		Agent location (queue 1), Agent location (queue 2)	The location of entities while waiting in either queue to be matched
Combine		Agent location (queue 1), Agent location (queue 2), *Agent location (combined)*	The location of the first, original entity that arrived from either input port while waiting for the other entity to arrive; "Agent location (combined)" option is assigning a location to a new entity and is described in Type A group (Table 3-12)
Batch		Agent location, *Location of batch*	The location where the entities wait to be batched
Assembler		Agent location (delay), Agent location (queue 1-5), *Location of assembled agent*	The location of entities while in any of the five embedded queues, or the delay; "Location of assembled agent" option assigns a location to the newly assembled entity and is described in the Type A group (Table 3-13).

Table 3-4 shows all available options for Type B blocks and their properties we discussed in Table 3-14.

Table 3-15: Options available in the animation related fields of Type B blocks

Options	Delay	Queue	Seize	Service	Match	Combine	Batch	Assembler
Not specified	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Node *(with or without attractors)*	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Path	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Not specified: No arrival place for entities are specified, and thus not displayed. This is the default setting and is chosen in models of Type I (with no animation).

Node *(with or without attractors)***:** A node (Point, Rectangular, or Polygonal) in the environment; if the selected node is Rectangular or Polygonal and has attractors, those attractors specify the exact location after the entity enters the node.

Path: If used as the setting in a **Queue** block (or other blocks containing a queue inside such as **Seize**, **Service**, **Batch**, **Match**, **Combine**, **Batch**, **Assembler**), the entities jump to their location on the path and don't move along it. Here, the position of an entity on the path depends on its position in the queue. The entity at the head of the queue (index 0) is displayed at the path's end point. The distance between the two subsequent entities is set to the path length divided by one less of the Queue's capacity. If the entities inside a **Delay** block are assigned to a path, each of them appears at the start of the path and then move along the path. The entity's speed is adjusted in a way that forces it to reach the end of the path at the same time its delay is finished.

C) Blocks that set the destination of moving entities or resource units

For Type C blocks that move entities or resource units, you need to know the movement's destination. Movement modeled by these types of blocks mimicks actual physical movement and therefore part of the model definition (Type III or IV).

Table 3-5 shows the blocks that make up this type. Note we categorize **Release** and **ResourceTaskEnd** under Type C although their properties don't set a new destination. Instead, they move the released resource units back to their original "home" location set by their **ResourcePool** block.

Table 3-16: Blocks that set the destination of entities or resource units (physical models)

Block name	PML Icon	Setting	Description
MoveTo		"Destination"	Destination that the moving entity or resource unit should move to.
Seize		"Destination is"	Destination that the seized resource units should be sent to (only available if "Send seized resources" is checked). The movement of units is modeled by an embedded **MoveTo** block inside the Seize.
Service		"Destination is"	Destination that the seized resource unit should be sent to (only available if "Send seized resources" is checked). The movement of units is modeled by the embedded **MoveTo** block in the embedded **Seize** block.
Resource Send To		"Destination is"	Defines the destination that the seized resource units should be sent to (by a preceding block).
Release		"Moving resources" *When "Return to home location" is selected (radio button)*	Doesn't define a new destination here; rather, it uses the "home" location set in the **ResourcePool** block to return the released units. The return movement of units is modelled by an embedded **MoveTo** block.
ResourceTaskEnd		*When "Move resource to its home location" (checkbox) is selected*	Doesn't define a new destination here; rather, it uses the "home" location set in the **ResourcePool** block to return the released units. The return movement of units is modelled by an embedded **MoveTo** block.

Table 3-6 shows all available options for destinations of Type C blocks and the properties discussed in Table 3-16.

Table 3-17: Options available in the animation related fields of Type C blocks

Options	MoveTo	Seize	Service	Resource Send To
Node *(with or without attractors)*	Yes	Yes	Yes	Yes
Attractor *(inside a Node)*	Yes	Yes	Yes	Yes
Agent	Yes	Yes	Yes	Yes
Other seized resource unit	No	Yes	No	Yes
Seized resource unit	Yes	No	No	No
Home of seized unit	Yes	Yes	No	Yes
x, y, z	Yes	Yes	No	No
Node + (x,y,z)	Yes	No	No	No
[res.] Agent which possesses me	Yes	No	No	No
[res.] Other unit seized by my agent	Yes	No	No	No
[res.] Other unit's home	Yes	No	No	No
Network / GIS node *(in GIS Map)*	Yes	No	No	No
latitude, longitude *(in GIS Map)*	Yes	No	No	No
Geographical place *(in GIS Map)*	Yes	No	No	No

Node *(with or without attractors):* A node (Point, Rectangular, or Polygonal) in the environment; if the selected node is rectangular or polygonal and has attractors, those attractors specify the exact destination after the entity enters the node.

Attractor *(inside a Node):* Entities destinations are assigned to a *specific* attractor inside a Rectangular or Polygonal node; this is in contrast to the "Node" option in which assignment to attractors are automatic.

Agent: For the **MoveTo** block, the destination agent could be any agent (entity or resource unit). For the other Type C blocks (**Seize**, **Service**, and **ResourceSendTo**), the "agent" is the entity inside the block that has seized a resource unit.

Other seized resource unit: For the **ResourceSendTo** block, moving units selected in the **Resource to send** field will be moved toward the units in the pool selected in the **Resource** dropdown menu. For the **Seize** block, it will be the resource pool selected in the **Resource** dropdown menu seized by a previous block. In other words, we should have nested **Seize** blocks for this option to work.

Seized resource unit: This option is only available in a **MoveTo** block and sets the destination to the resource unit that was seized in a previous block. Since the entity might have seized units from multiple resource pools, you must select the specific pool from the **Resource** drop-down menu.

Home of seized unit: For **Seize** and **ResourceSendTo**, a seized unit is set to move toward its home location. For the **MoveTo** block, the entity inside the block will be set to move towards the home

location of a previously seized unit. Since the entity might have seized units from multiple resource pools, you must select the specific pool in the "Resource" drop-down menu.

x, y, z: The entity (for **MoveTo**) or the moving resource unit (for **Seize**) will move to the point at the specified coordinate.

Node + (x,y,z): The entity or resource unit first moves to the specified node, then to the point at the specified *absolute* coordinates. This option sets two consecutive destinations at once. If you want the x,y,z coordinates to be relative to the node's location, you can set the coordinates by an relative offset (for example, setting the value for X to: `node.getX() + offset`).

The next three destinations are for resources (abbreviated as "res.") used in the preparation and wrap-up flowchart branches. In these branches, the seized unit behaves like the entity in a regular process; we assign the destination from the perspective of the unit that flows in that branch.

[res.] Agent which possesses me: The seized resource unit moves to the entity (in the primary branch) that seized (possesses) it.

[res.] Other unit seized by my agent: This option only works when the **Seize** block has seized more than one resource units. You also must select the **specific resource** option in the **ResourceTaskStart** block and assign a specific pool to the preparation branch. Then, in a **MoveTo** block that is in the same preparation branch, you can select this option and choose one of the other pools in the **Resources** drop-down menu.

[res.] Other unit's home: This is very similar to the previous option; the only difference is that the unit in the preparation branch, or the selected **specific resource** option in the **ResourceTaskStart** block, will move toward the "home" location of the other unit that is inside a pool selected in the **Resources** drop-down menu.

Network / GIS Node *(in GIS Map):* Entities move toward a given GIS point, or GIS region.

latitude, longitude *(in GIS Map):* Entities move toward the given coordinate on the GIS map.

Geographical place *(in GIS Map):* Entities move toward a named location on the GIS map. AnyLogic uses the first result found from searching the named location database to choose the location.

D) Blocks that cannot keep an entity (or resource unit)

In these blocks, the entity or resource unit passes through in zero model time and thus cannot be animated. These blocks include: **SelectOutput**, **SelectOutput5**, **ResourceAttach**, **ResourceDetach**, **Hold**, **Sink**, **Exit**, **Unbatch**, **Pickup** (entity waits in the Queue connected to it), **Dropoff**, **RestirctedAreaStart**, **RestirctedAreaEnd**, **TimeMeasureStart**, **TimeMeasureEnd**, and **ResourceTaskStart**. It could be argued **Release** and **ResourceTaskEnd** could also fall under this category, since they technically don't set a new destination for the return movement. A counter-point to this is they're still setting movements with an implicitly defined destination.

How to build type I, II, III, or IV models

Now we'll review how you can use the PML blocks we categorized as types A, B, C and D to build Type I, II, III or IV models.

Type I (no animation): These models only have logical blocks and don't have any animation. Your model will work if you keep your PML blocks' default settings and don't use **MoveTo** and **ResourceSendTo** blocks (since they're type C blocks and only work in models with physical movement). You can use **Seize** and **Service** blocks if you don't use their internal **Send seize resources** setting.

Type II (illustration only): In these models, animations are only illustrative. You usually start by drawing path and nodes with the space markups for locations that visually represent where the entities are. AnyLogic will automatically generate networks for interconnected path and nodes (or any isolated path), but you won't use them. You can then associate any of the Type A and B PML blocks to the space markups.

As a reminder, Type A blocks specify the first location of entities and resources. Type B blocks specify their location, providing they reside in the block and assuming you kept the default **Restore agent location on exit** checked. In these type of models, the animation doesn't affect the quantitative outputs. This means the modeler doesn't have to worry their animation might affect the model's computations. For example, you can partially animate the process for a selected group of blocks.

> *One important caveat in Type II animations: you can assign illogical animations such as assigning two blocks to one node without causing a quantitative flaw. In other words, the mistakes you make as you create an animation won't lead to error messages. This means in case you encounter unexpected behaviors in your model animation you should look for errors in animation setup before moving to verify the model's logic.*

Type III (physical movements are part of model definition): In these types of models, the movements in the environment are part of the model definition and they directly affect the model outputs. Like Type II, we must draw the environment with space markup elements and then associate the logical blocks with those elements. The moving entities or resource units use the network to find the closest path between two nodes. Since the physical space is part of the model definition, you must use the **Scale** element to scale the environment.

After you use space markups to draw the environment, you must use Type A blocks to set the first location of newly built (or entered) entities or resource units. This allows each entity or resource unit to know where it should be when it appears. You should use Type C blocks (which specify destination locations) to govern all other movements of entities and resource units.

Essentially, you leave the Type B block's **Agent location** fields untouched (by selecting the **Not specified** option). This doesn't apply to location of agents in **Combine**, **Batch**, and **Assembler** blocks, which we categorize under Type B even though a subsection of their properties behaves like Type A blocks.

> *If you assign a node or path to a Type B block's **Agent location** setting and uncheck its **Restore agent location on exit** checkbox, it's equivalent to forcing the entity or resource unit to jump (that is, in zero time) to the location set in the **Agent location** field. While this is possible, it isn't realistic for a physical thing to change its location in zero time. If you must animate the location of agents when they reside inside a Type B block, we recommend you use Type IV models.*

In summary, the entities or resource units in Type III models start at a specific location. They only change their animation's location when a real, physical movement happens in the model. The position of the entity or resource unit at any moment represents its actual location in the physical environment since it's either its initial location or the result of prior movements modeled by a Type C block.

> *In Type III animations, the model's movements and animations are part of the logic definition and to the outputs. This means any movement setting that is logically impossible will lead to error messages.*

Since these models have physical level movements, entity speed and resource unit speed are important. You can set the initial speed of entities in **Source** and change their speed in the **MoveTo** blocks when they enter it. It's important to note updating the speed of an entity or a unit in a **MoveTo** block persists after exiting the block. For resource units, we set their initial speed in the **ResourcePool** block's properties.

Type IV (combination of Type II and III): For this type, consider it more of Type III than Type II. This is because in these models, physical movement is part of the definition. In addition, and as an auxiliary illustration, the location of entities inside Type B blocks (those which show contained entities) are also animated. As we mentioned, the location of entities or moving units inside Type B blocks isn't part of the model logic (in almost all cases). It's Type C blocks that with their movements control the position of entities and units. You can essentially consider Type IV models as Type III models that have illustrative animations for entities (or resource units) while they reside in Type B blocks.

Since we want the animation inside Type B blocks to be a temporary illustration for almost all Type IV models, we should keep the **Restore agent location on exit** checkbox enabled. The only exception is the **Assembler** block that doesn't have the checkbox and won't restore the entity's location after exit. You can leave the agent's location (for the delay or for any of the five queues) blank and it won't assign any illustrative animation.

The **Restore agent location on exit** checkbox means that while AnyLogic may display the entered entity or resource unit in this Type B block in a node or path, the shown position is temporary. The previous actual physical position, set by a preceding Type A or C block, will be restored upon exiting the block. This behavior allows you to separate the animation for the physical movement from the animation that is illustrative.

> *Assume a scenario where you have several Type B blocks connected in a series and you've selected the **Restore agent location on exit** checkbox in all of them. In this scenario, as they exit the last block, the entities or resource units will return to the last node or path that represented its last physical location and was assigned by a Type A or C block.*

To clarify how Type IV models work, let's review a simple example model (Figure 3-105) that shows the effect of **Restore agent location on exit** in three **Delay** blocks (Type B) connected in tandem. This model also has a **Source** (type A) and two **MoveTo** blocks (Type C) that animate real, physical movements. You don't have to build this model; we offer these descriptions to clarify the settings and outputs. To simplify the animation and explanation, only one entity arrives (at time 0), and all nodes are **Point Node** objects.

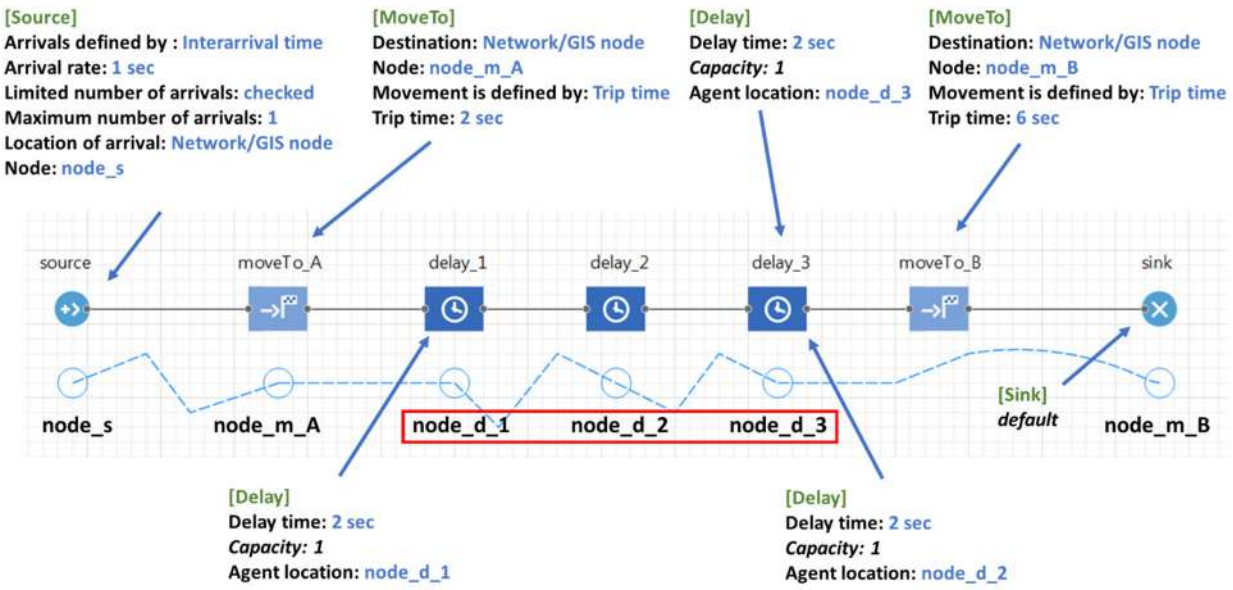


Figure 3-105: Example of a Type IV model with the "Restore agent location on exit" checkbox in a series of Type B blocks

The following actions will occur during the simulation:

At time 1: The entity enters the model. Since the **Location of arrival** field of the **Source** block is set to **node_s**, the entity appears at that node (Figure 3-106: time = 1). Since **Source** is a Type A block, it sets the actual physical location of the arriving entity.

Between time 1 to 3: The **moveTo_A** block is set to move the entity based on the 2 seconds trip time. This means the move that began at time 1 will finish at time 3 (Figure 3-106: time = 1 to 3).

At time 3: **MoveTo** is a Type C block and models a real physical movement, so at time 3 the actual location of the entity would be at **node_m_A** (Figure 3-106: time = 3).

Between time 3 to 5: Right after the **moveTo_A** block, the entity enters **delay_1** and stays there for two seconds. Since **Delay** is a Type B block and its **Restore agent location on exit** checkbox has been checked, although we'll see the entity on **node_d_1** while it is inside the **delay_1** but this illustration is temporary, and the entity's location will be restored upon leaving the delay (Figure 3-106: time=1 to 5).

At time 5: As soon as the entity leaves **delay_1**, the entity restores its previous real location and is shown at **node_m_A** (Figure 3-106: time = 5).

Between time 5 to 7: This is similar to what had happened between time 3 to 5; but this time, **delay_2** and its associated agent location node that is **node_d_2** (Figure 3-106: time = 5 to 7).

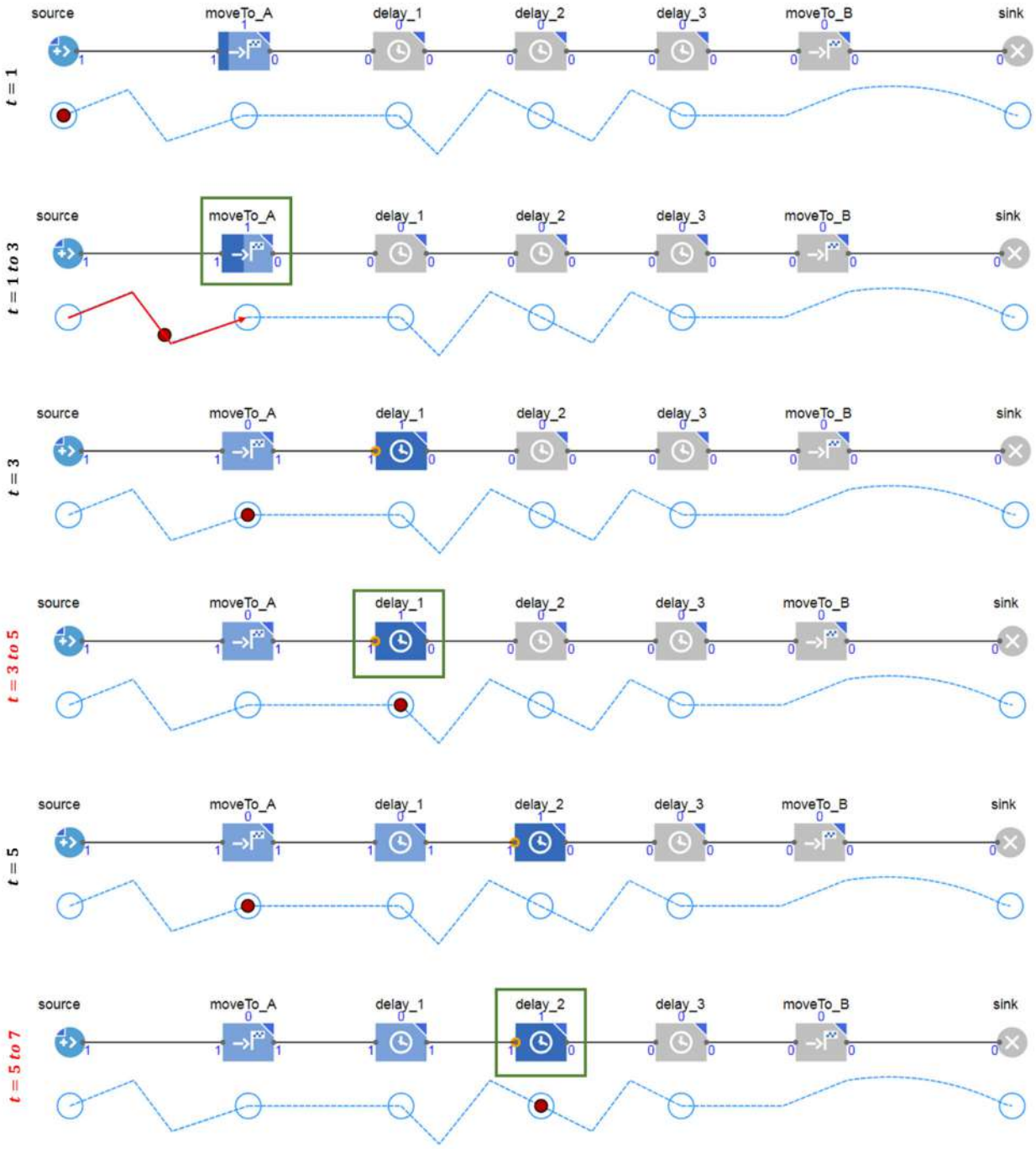
At time 7: As soon as leaving **delay_2**, the entity restores its previous real location, **node_m_A**. The leaving entity tries to restore its location to the one before entering this block, which was at **node_d_1**. However, that location will be restored to the entity's previous location, which leads back to **node_m_A** (Figure 3-106: time = 7).

> *If you keep the "Restore agent location on exit" checkbox of all Type B blocks checked, the entity or unit will restore to its last actual physical location that was set by a Type A or C block.*

Between time 7 to 9: Similar to what had happened between 3 to 5 and 5 to 7, the entity enters **delay_3** and is animated on the associated agent location node, **node_d_3** (Figure 3-106: time = 7 to 9).

At time = 9: The entity restores its location again to **node_m_A**, its last real location (Figure 3-106: time = 9).

Between time 9 to 15: The entity starts moving from its last real location (**node_d_3**) to **node_m_B**, which is the destination that that is set by the **MoveTo_B** block (Figure 3-106: time = 9 to 15).



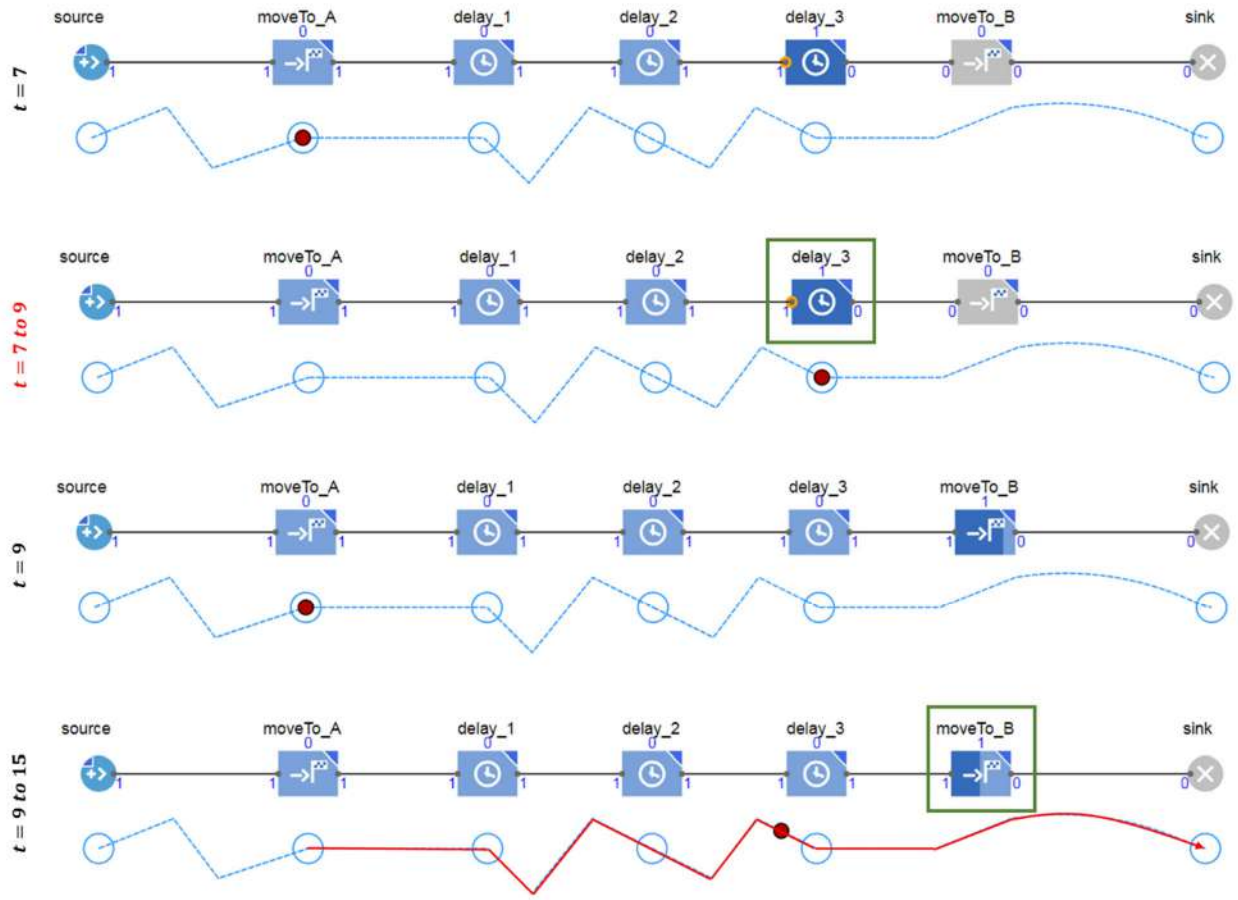


Figure 3-106*: Animation of the simulation model shown in Figure* 3-105 *at run time and at different model times*

Hopefully, after seeing the abovementioned example, the differences between Type A, B, and C blocks are clear. You should also have a clear understanding of what will happen when you mix all types of PML blocks and have real physical movements and illustrative animation in one model (Type IV models).

Example Model 4: Clinic

Prerequisites:

Model Building Blocks (Level 1): **Source**, **Queue**, **Delay**, **Sink**, **Seize**, **Release**, **ResourcePool**, **TaskMeasureStart**, **TaskMeasureEnd** blocks.

Model Building Blocks (Level 2): **Source**, **Seize**, **Release**, **Schedule**, **ResourcePool**, **MoveTo**, **PMLSettings**.

Math: Random variable, Random variate, Poisson process

Learning Objectives:

1. Using preparation and wrap-up process branches
2. Building networks using space markups elements
3. Adding custom animation for entities and resource units
4. Using **MoveTo** block to move entities and resource units
5. Sequential resource use (nested **Seize**-**Release** pairs)
6. How to setup the Optimization experiment

Problem statement

Electrocardiography (ECG or EKG) is the process of using an instrument to records the electrical impulses that travels through the heart in each heartbeat. Ultrasound imaging (US), which is sometimes called Sonography, is a procedure in which sound waves are used to produce pictures of organs inside the body. In this example we're modeling an ultrasound clinic. Specifically, we're focusing on a part of this clinic where incoming patients first get recordings of their heart's electrical activity via an ECG and then get an ultrasound (US). At a high-level of abstraction, this can be described as patients coming in, going through two processes in sequence, and then leaving (Figure 3-107).

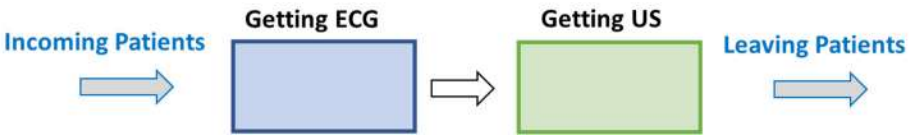


Figure 3-107: Overview on the clinic operation (high abstraction)

Building a model at this level of abstraction is useful as a starting point. However, the management of the facility is looking for a more detailed simulation of the operation. Based on their experience, the high-abstraction analysis of the operation would not be enough to answer their questions, which will discuss shortly. Our more in-depth review of the processes revealed that each of these processes (performing the ECG and US) could be highly dependent on the actual movements in the facility. As a side note, it could be said that the ECG or US processes are either being *performed* by the resources (technicians/doctors) or the patients are *getting* them - the perspective you take can change how you name the processes. In this example, processes are named from the entity's perspective and therefore the processes are named as *getting* ECG and *getting* US.

The following section provides detailed descriptions of the entities that receive both processes:

Getting ECG: When a patient enters the clinic, they must wait in a waiting area (#1 in Figure 3-108) for their name to be called. A patient whose name is called will walk to the "ECG Room", chooses a free spot (one of the #2s in Figure 3-108) and will wait there for a technician. The ECG technicians are in their own room, "Technician Room", (#4 in Figure 3-108) and will be alerted when the name of the patient is called. They will first go to the "ECG Storage" room and pick up an ECG machine (#3 in Figure 3-108) and then they will bring it to the waiting patient in the ECG room. Due to this, the patient must wait in the ECG room for a short period before the technician shows up with the ECG machine and begins the procedure.

Based on historical data for the ECG procedure, we suggest you use a triangular distribution with a minimum of five minutes, maximum of ten minutes, and mode of seven minutes (Table 3-18). When the ECG procedure is finished, the technician returns the ECG machine to storage and walks to the technician room. After each procedure, technicians rest for five minutes before they accept a new patient. A patient who has completed the ECG won't leave the ECG room until there's an available pair of ultrasound machine and its nearby bed (Any pair of #5 and #6 in Figure 3-108).

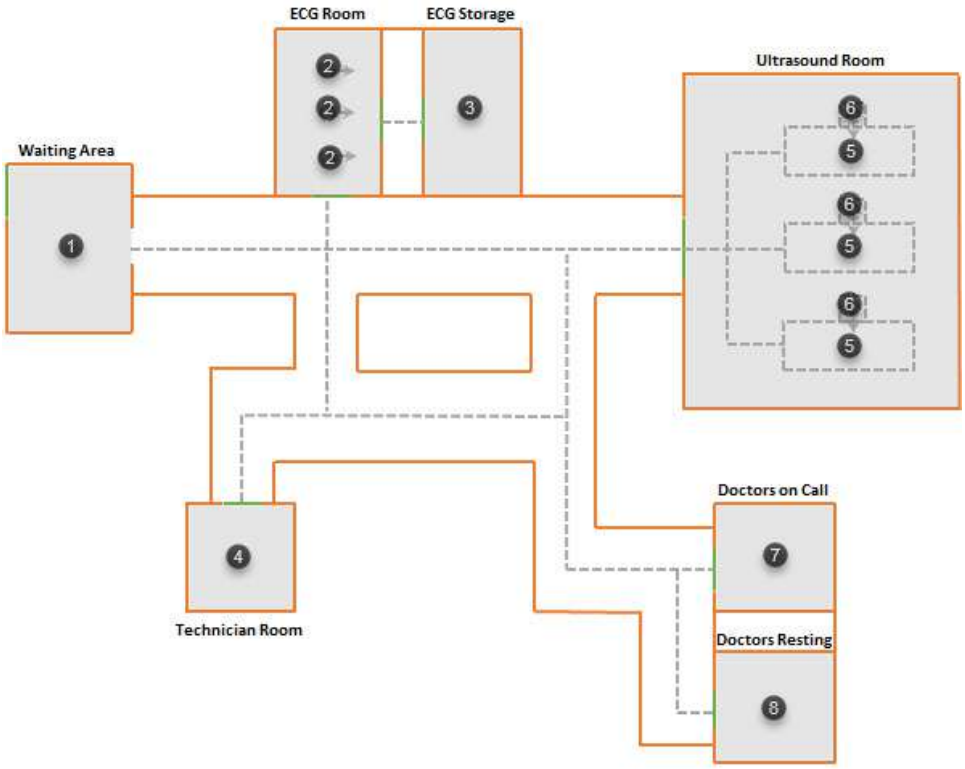


Figure 3-108: Clinic layout plan

Getting US: At the start of this procedure, a patient is finished with the ECG but still waiting in the ECG room (one #2 in Figure 3-108) for a free spot in the "Ultrasound Room". Each US spot is a combination of a US machine (one #6 in Figure 3-108) and the bed it's assigned to (one #5 in Figure 3-108). When there's an available spot, the patients name is called; they walk to the assigned bed and lay down on it. At this point, the patient is ready and a request for a doctor is sent. If there's no doctor available (that is, they're resting or with other patients), the patient must wait until a doctor becomes available. When an on-call doctor is readily available, which is if they're in the "Doctors on Call" room (#7 in Figure 3-108), they will walk to the patient and begin the procedure. We suggest a truncated triangular distribution for the time

needed for the US procedure (this distribution is like a regular triangular distribution, but whose ends are cut off based on the given min and max values). This distribution consists of a minimum of five minutes, maximum of twenty-five minutes and a mode of ten minutes that is then truncated at seven and twenty minutes (as shown in Figure 3-109).

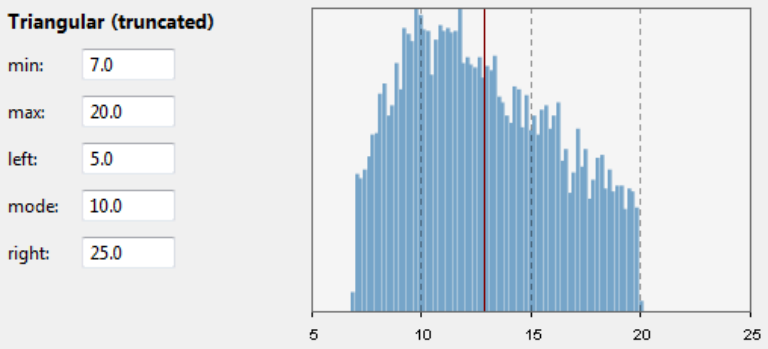


Figure 3-109: Truncated triangular distribution of the US procedure time (in minutes)

After the completion of the US procedure, the patient immediately leaves the clinic (leaving "immediately" means that the time it takes for a patient to leave the clinic is out of the scope of our model and isn't considered in our analysis). A doctor that has finished an ultrasound procedure will walk to the "Doctors Resting" room (#8 in Figure 3-108) and rest for some time - Doctors are supposed to rest for only ten minutes, but historical data shows that they spend up to five more minutes; therefore, the doctor's resting time is selected to be a uniform distribution with a minimum of ten and maximum of fifteen minutes. After resting, doctors return to the "Doctors on Call" room and wait for the next patient.

Table 3-18: Clinic tasks and their associated times

Task	**Distribution (minute)**
Getting ECG	triangular (5, 10, 7)
Getting US	triangular (7, 20, 5, 10, 25)
Doctors Resting	uniform_discr (10, 15)
Technician Resting	5 *(fixed)*

Walking paths: The management wants the clinic layout and its walking pathways to be incorporated into the operation of this model. They have provided the plan of the clinic and walking paths of patients, technicians, and doctors along with the following information:

Patients walking paths: (each of the two steps labeled in Figure 3-110 below):

1. The path taken from the waiting area to the ECG room when the patient's name is called for the ECG procedure. There are three spots in the ECG room and the patient will choose one based on availability.
2. The path taken from the ECG room to an ultrasound bed when the patient's name is called for the US procedure. There are three beds in the Ultrasound room and the patient will walk toward a free one.

As we discussed, the model doesn't have to keep track of patients after the US procedure is finished. This means it doesn't include a path from the bed to the outside of the clinic. In reality,

the patients walk back to the waiting area and leave from the same door. Since we aren't using this model to evaluate this behavior, following the patients after they finish the procedures doesn't add any value to our analysis.

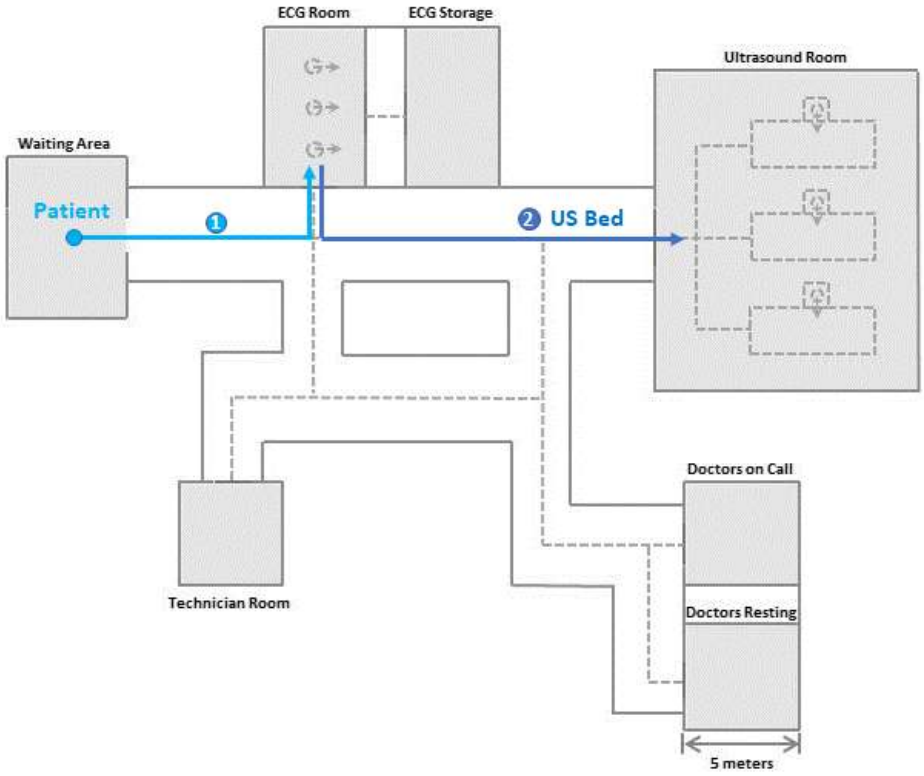


Figure 3-110: Patients' walking paths

Technicians walking paths: (each of the four steps labeled in Figure 3-111 below – paths shown in the figure are illustrative examples of the procedure and may vary for each instance)

1. The path taken from the technician room to an available ECG machine in the storage room.
2. The path taken from the ECG storage room to the patient who is waiting in the ECG room. After the completion of this path, the technician is ready for the ECG procedure to start.
3. The path taken after the procedure is finished to return the ECG back to the storage room. Note the locations shown in Figure 3-111 are example locations; since there's no dedicated location in the room for the machines, the technician can leave them in any empty spot (which may be different from the spot they originally took the machine from).
4. The path taken from the ECG storage room back to the technician room. After returning the ECG machine back, technicians go back to their room to rest and get ready for the next patient.

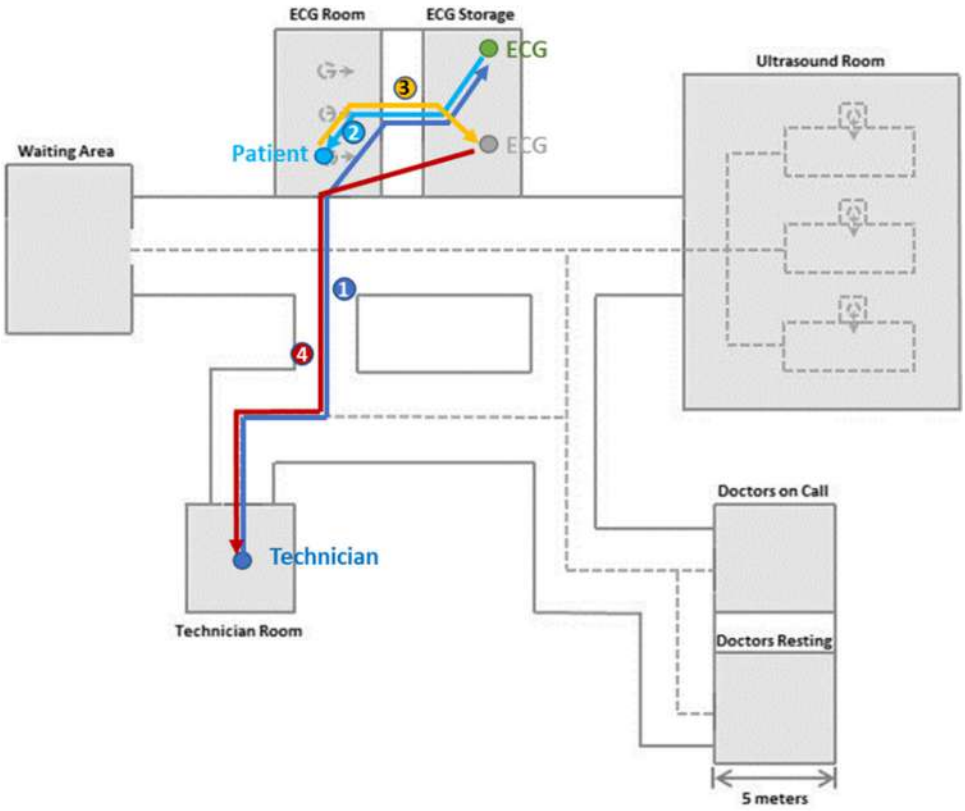


Figure 3-111: Technicians' walking paths

Doctors walking paths: (each of the three steps labeled in Figure 3-112 below)

1. The path taken from the on-call room to the US room. When a patient is laying on a bed and is ready for the US procedure, one of the on-call doctors will start walking toward that patient. The doctor will go to a spot in front of the patient's bed and to the left of the US machine (example shown in zoomed area of Figure 3-112).

2. The path taken after the completed procedure from the US room (specifically, the spot beside the US bed) to the doctors' resting room, where the doctor will rest for some time.

3. The path taken from the resting room to the on-call room. They will wait here, ready for the next patient.

Note that, like the ECG storage room, there are no designated locations inside the doctors' resting or on-call rooms. This means doctors will select a random spot inside these rooms when they enter and stop there until the next movement.

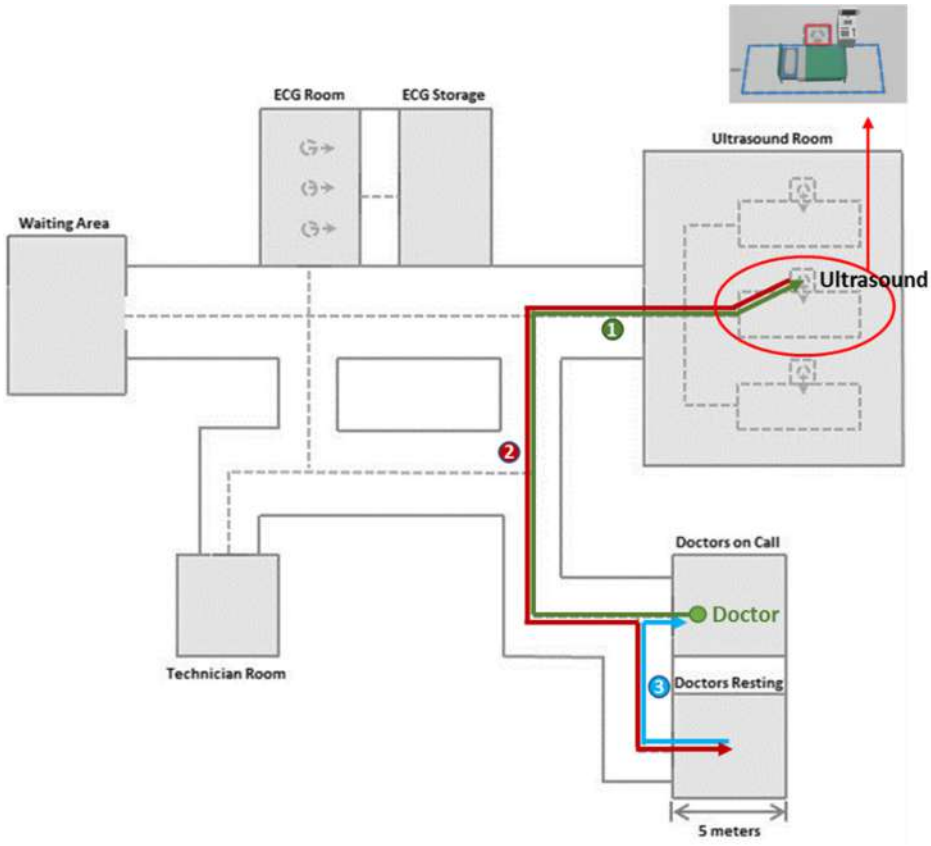


Figure 3-112: Doctors' walking paths

To assign a reasonable walking speed to each type of entity (patients, technicians, and doctors), we gathered data. Based on what we found, we'll let moving entities use the moderately conservative speed of 1 meter/second (~3.28 Foot per second).

Required solution

Since the clinic was renovated recently, the clinic's management won't make fundamental changes to its layout or equipment. This means the number of ECG spots or ultrasound beds (three, in both cases) can't change. However, since the clinic is a subsidiary of a large medical university, they can easily change the number of ECG technicians and doctors.

The clinic's management has the following objectives:

- Minimize the time to process patients - this is the most important indicator of patients' satisfaction, apart from the quality of procedures
- Process 50 patients during each operating day
- Complete the ECG and US procedures for all admitted patients by the end of the clinic's daily working hours (within eight hours) and without the need for after hour work

Adding more staff is will add to overall operating cost of the clinic. However, due to the educational nature of the operation, the cost of adding doctors and technicians isn't part of the decision criteria. Any extra overhead related to hiring more staff is assumed to be absorbed by the educational benefits of training

provided at the clinic. Therefore, we can add as many staff as we need without worrying about the associated cost.

Key metrics

Managers typically use their objectives and requirements to develop the metrics that will gauge the system's performance. This means we should use the following metrics as the simulation model's outputs:

1. Average processing time for the patients who received ECG and US in a day of operation.
2. Utilization of resources (technicians and doctors)
3. The departure time of the last patient (used as a measure of daily operation duration)

Examining possible model designs

It's evident the entities in our model are the patients passing through the process. For the resource pools, there are clear candidates: physicians, doctors, ECG machines, and US machines. Other resources that may not be as obvious are the spaces in the ECG room and the beds in the US room.

As you see, physical spaces could be considered as resources that are seized, used, and released upon the completion of tasks. Defining physical spaces as a pool of resource units lets us easily keep track of spaces occupied and unoccupied (under use and free). For example, in our clinic, patients need a spot in an ECG room to receive the ECG procedure. We know there's a maximum of three spots for ECG machines and we can easily model those spaces as a resource pool with a capacity of three. An entity (patient) can seize one of those resource units if there's at least one available, receive its ECG and then release the spot back to the pool for other patients who are waiting.

Now that we know which entities and resource pools we need, we can shift our focus to building the two processes shown in Figure 3-107. We're going to build the model with three levels of abstraction and progressively increase the details:

Version 1 (High abstraction level): Focus is on the two main processes (getting ECG and US) but we keep details to the minimum. In our first iteration, like our previous examples, we only focus on the operation at a logical level (only the process) and won't model the physical space (the path network).

Version 2 (Medium abstraction level): We'll incorporate the clinic's layout into the model's definition. We'll define the paths and nodes and substitute simple **Delays** with **MoveTo** blocks from PML library that let the patients, technicians, and doctors move along their respective paths. The moving entities or resource units will automatically analyze the network (or the network of paths and nodes that defines the clinic's layout) and find the shortest path to a specified destination.

Version 3 (Low abstraction level): The process will be completed by adding preparation and wrap up branches that explicitly model tasks related to the technicians and doctors. The preparation process includes technicians' movements in bringing the ECG machine to the ECG room. The wrap-up branch models the process of returning of the ECG machine to the storage room and starting the technicians' resting time. We'll also add a wrap-up branch for the doctors to model their resting time.

By the end of version 3, we'll have a very detailed model of the clinic operation. We'll use this final, low abstraction version for our further experimentation and final recommendations.

Version 1 (high abstraction level): No physical space; 50 patients admitted daily with staff consisting of three physicians and three doctors

Our focus here is on the two main processes: getting ECG and getting ultrasound (Figure 3-113). As we've discussed previously, patients are the entities passing through the flowchart and receive service from six resource pools: technicians, doctors, ECGs, ultrasounds, ECG room spaces, and ultrasound beds. We'll record the time each patient spends in the process to find the average processing time. We also record the time the last patient departs the clinic.

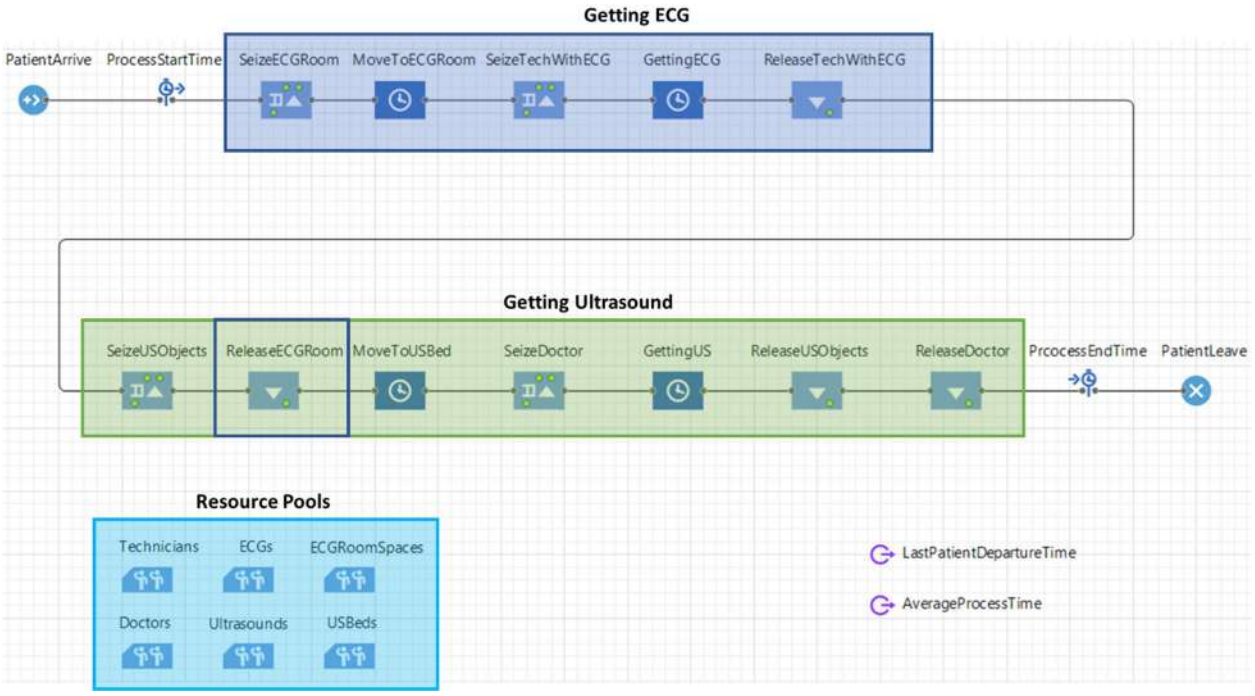


Figure 3-113: Overview of the flowchart blocks of version 1

To build the first version of this model:

1. Create a new model; time units: minutes.
2. Add six **ResourcePool** blocks to the lower left corner of the frame (blue rectangle) as shown in Figure 3-113; rename them according to Figure 3-114 and change their capacities to three (all six pools have the same capacity).

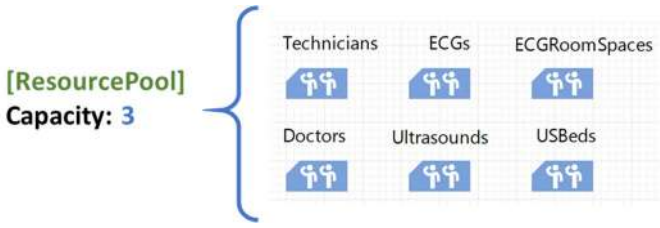


Figure 3-114: Six resource pools with similar settings

3. Build the following flowchart by dragging and dropping the needed blocks from the Process Modeling Library (PML) palette, as shown in Figure 3-115. Afterward, modify the properties of each according to Figure 3-115 and the following written instructions (renaming each block according to the screenshots). Each block has several attributes and settings; you only need to change the settings we mention. Note the green text inside brackets are the PML block's labels (type), the black text are the attributes you need to change, and the required settings are in blue.

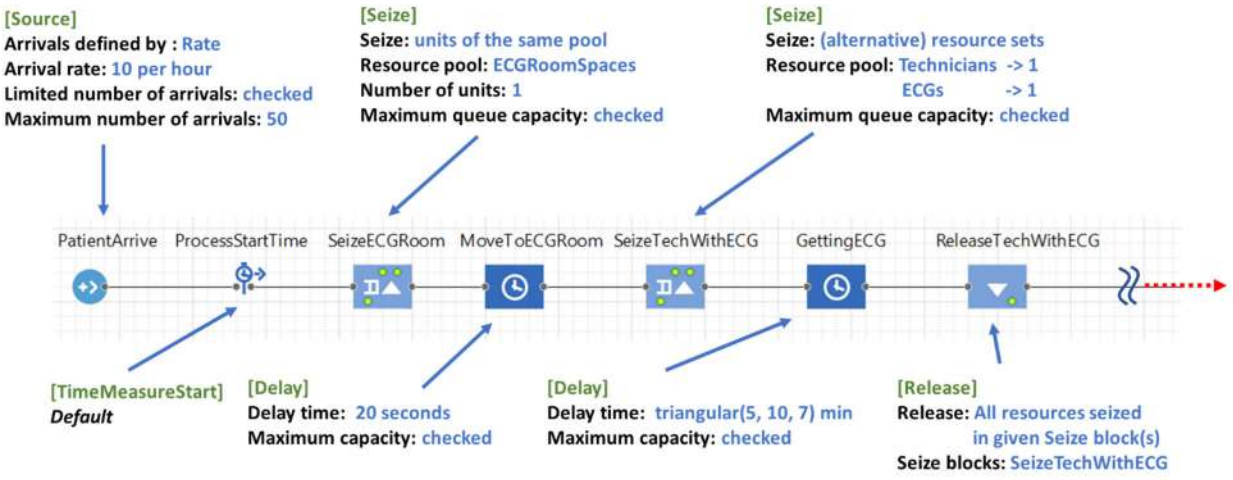


Figure 3-115: Getting ECG flowchart

In the "Getting ECG" flowchart, we first add a **TimeMeasureStart** block to start recording the time a patient spends in the clinic. Afterward, the patient will wait for a free ECG room. When the ECG spots are occupied, the patients will wait in the embedded queue of the **SeizeECGRoom** (unlimited capacity). The **SeizeECGRoom** is associated with the **ECGRoomspaces** resource pool; as setup in step 2, there are three source units in the **ECGRoomspaces** resource pool. A patient who seized one of these resource units will walk toward the ECG room.

Bear in mind, this version of the model doesn't explicitly model movement. This means a simple **Delay** will approximately represent the time the patient needs to walk to the ECG spot. Since the entity's speed (the patient's speed) is 1 m/s and the path is 20 meters long, it takes 20 seconds to walk to the spot.

Next in the process is to seize the resources - the technician and ECG machine - that the ECG procedure needs. The **SeizeTechWithECG** block seizes a set of a technician from the **Technicians** resource pool and an ECG from the **ECGs** resource pools.

One important decision we made in this version of the model was to not model the preparation needed for the ECG procedure. This is a significant assumption since, before the ECG procedure starts, the technicians must move the ECG machines to the ECG room. After the procedure, the technicians must return the machine, return to their room, and rest for a five minute before they attend to other patients. This high abstraction version of the model factors out these tasks that are part of the preparation process.

Another important concept is the nested pairs of **Seize** and **Release**. As you can see in Figure 3-115 above, the patient seized an ECG spot (**SeizeECGRoom**) and without releasing it, seized a

technician and ECG machine (**SeizeTechWithECG**). This sequential resource seizing was necessary since the movement toward and waiting in the ECG room don't depend on the availability of the ECG machine and technician. In other words, the patient should move to the ECG room and wait there any time there's an empty spot. They should do this even if an ECG or a technician aren't immediately available.

The process also won't release the spot in the ECG room until the patient is called for the US procedure. This means that while the process will release the technician and ECG machines and make them available immediately after the ECG procedure, the spot in the ECG room won't be available! If there are three patients in the ECG room and some have completed the ECG procedure, but the US beds are full, they won't leave the ECG room. This means no other patients can enter the ECG room. In this design, the availability of US beds could affect (and therefore depends on) the ECG room's availability.

After finishing the ECG procedure, the entity releases the technician and ECG machine in the **ReleaseTechWithECG** block. As you see in the setup figure, we changed the default setting for releasing from **All seized resources (of any pool)** to **All resources seized in give Seize block(s)**. This was done because we just wanted the technician and ECG machine to be released and not the ECG room space, which will be released later.

4. Build the process for getting the ultrasound, as shown in Figure 3-116.

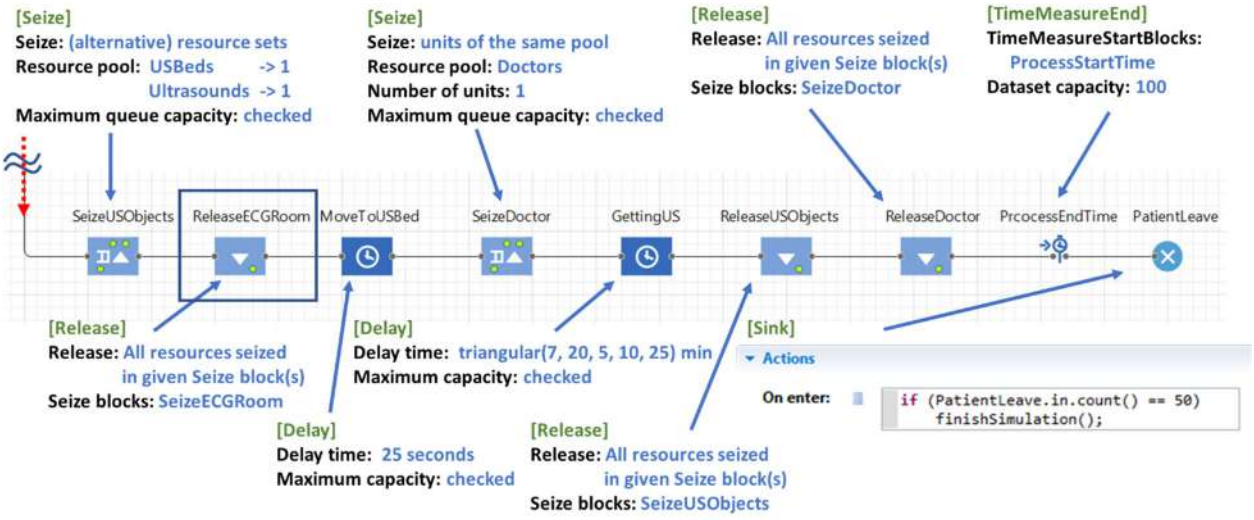


Figure 3-116: Getting Ultrasound flowchart

All the patients who completed the ECG procedure will enter the embedded queue of the **SeizeUSObjects** block and wait for a set of a US bed and US machine to be available. As shown in Figure 3-117 below, we didn't release the ECG room until after the US objects (US bed and Ultrasound) were seized. This results in patients continuing to occupy the space in the ECG room and only three patients able to be in any of the flowchart blocks between **SeizeECGRoom** and **ReleaseECGRoom**.

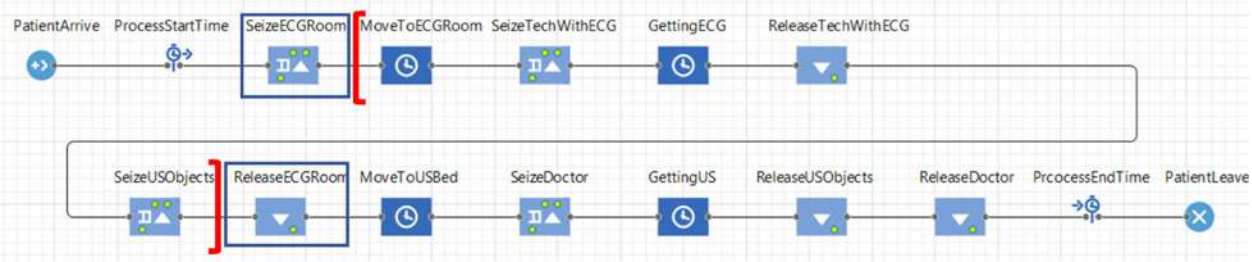


Figure 3-117: Only three number of patients allowed between the "SeizeECGRoom" and "ReleaseECGRoom"

Right after seizing the US bed and ultrasound in **SeizeUSObjects**, the **ReleaseECGRoom** block releases the ECG room space. The patient then moves to the bed with an estimated twenty-five second delay for this movement. After reaching the bed, the patient seizes a doctor with the **SeizeDoctor** block. It then receives the ultrasound procedure, draws its time from a truncated triangular distribution (with minimum of five, maximum twenty-five and mode of ten minutes, truncated at seven and twenty minutes, as shown in Figure 3-109).

Once the ultrasound procedure ends, the patient first releases the resources seized in **SeizeUSObjects** (the US bed and US machine), and then releases the resources seized in the **SeizeDoctor** block (doctor). We separated seizing the US bed and US machine from seizing the doctor because, for the patient to be called to the US room, we only need the bed and US machines (and not the doctor). On the other hand, we could have combined the releasing of the US bed and US machine into one **Release.** We separate them here knowing that in version 2 & 3 we'll need a more granular access.

We added a **TimeMeasureEnd** block at the end of our process and paired it with the **TimeMeasureStart** block at the beginning. Since the model's stop isn't preset and depends on the time the last patient reaches the end of the process, we'll add a simple condition to the **Sink** block's **On enter** field to check the number of departed patients. It finishes the simulation by the finishSimulation() function when it reaches 50 patients, the maximum number of daily admission.

5. To gather the required outputs, we add two **Output** objects from the **Analysis** palette (Figure 3-118). Both objects record a value in the moment the simulation ends (due to the default option for recording **On the simulation end**). The **LastPatientDepartureTime** object records the time when the **Sink** object's condition becomes true and the simulation ends. The **AverageProcessTime** object records the mean processing time for the patients who passed through the **ProcessEndTime** block (accessible since its embedded distribution saves their process duration).

Figure 3-118: Output objects added to record the time of last patient departure and average processing time of 50 daily patients

6. We're ready to run the simulation, and Figure 3-119 shows a single run's outputs. Just make sure the random seed is selected in your Simulation experiment randomness section.

As you may notice, this model stops running after 50 entities pass through it. This means it's essentially at a transient state (terminating). With a random seed setting enabled, the simulation experiment's outputs will differ if you run this model several times. In this version, our goal is to set a baseline we can build on in the following lower-abstraction versions. This means we aren't interested in running any other experiment other than the base Simulation.

As shown in Figure 3-119, the outputs of a single run of version 1 shows the daily admitted 50 patients are processed in a relatively short period of time (378 minutes / 60 minutes per hour < 7 hours) and the number of key resource units (technicians and doctors) is relatively high compared to what is needed (inferred from their low 32% utilization as shown in Figure 3-119).

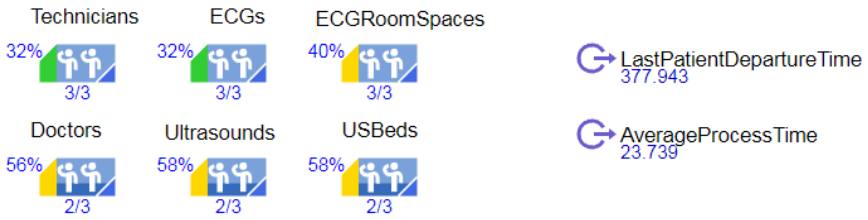


Figure 3-119: Outputs of one simulation run (version 1)

Version 2 (medium abstraction level): Physical space; 50 patients admitted daily with staff consisting of three physicians and three doctors

Building upon the version 1, we're going to add physical movements to the model. We'll do this with space markups - part of AnyLogic's model building blocks that allow us to draw a model's spatial elements. In a few cases, space markups are used only for the sake of a representative animation that does not contribute to the model's logic. Most often however, these spatial elements are part of the model's definition (its logic). This means these spatial elements will affect the model's quantitative outputs. In cases like this where space markups are part of the logic, we need to draw these elements in a properly-scaled environment.

In this clinic model, we want to explicitly incorporate the movements of patients, technicians, and doctors. We'll do this by importing an image that represents the facility's layout and then draw the space markups on top of it.

1. Open the first version of the model built in the previous section. From the **File** menu you can click **Save as…** and save the model with a different name to keep the original model intact.
2. From the Presentation palette, drag and drop the **Image** object to Main (Figure *3-120*). Browse your hard drive for a file named "**CLINIC_LAYOUT.JPG**" in the supporting materials. Select the image and move it outside and below the blue frame.

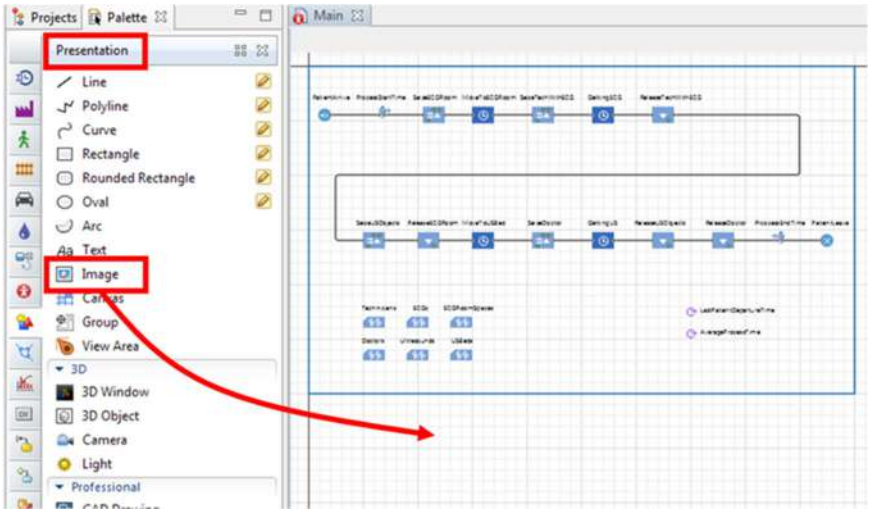


Figure 3-120: Drag and drop of Image object from Presentation palette

3. To lock the shape so it won't move when we start to draw our space markups on top of it, right-click the image and select **Locking** > **Lock shape**. At this point, you can't select the shape. The only way to unlock it is to right-click it and select **Unlock all shapes**.
4. To scale the environment, do the following:
   a. Zoom out to see the blue frame and the default **Scale** object above it (Figure 3-121).
   b. Click and drag the scale object close to the reference scale on the image (Figure 3-121). The imported image has a reference scale (a line with a known length of 5 meters) below the Doctors Resting room.
   c. Zoom into the area near the reference scale and move the **Scale** object so its left-end aligns with the reference scale's left-end.
   d. Click and hold the **Scale** object's right handle and drag it to match the reference scale's right-end.
   e. Click the **Scale** object, open its **Properties** tab and change the **Ruler length corresponds to** setting to five meters. This will scale everything in Main according to this reference scale. As you can see in Figure 3-121, this means fifteen pixels in the model's graphical environment corresponds to one meter in the simulated reality.

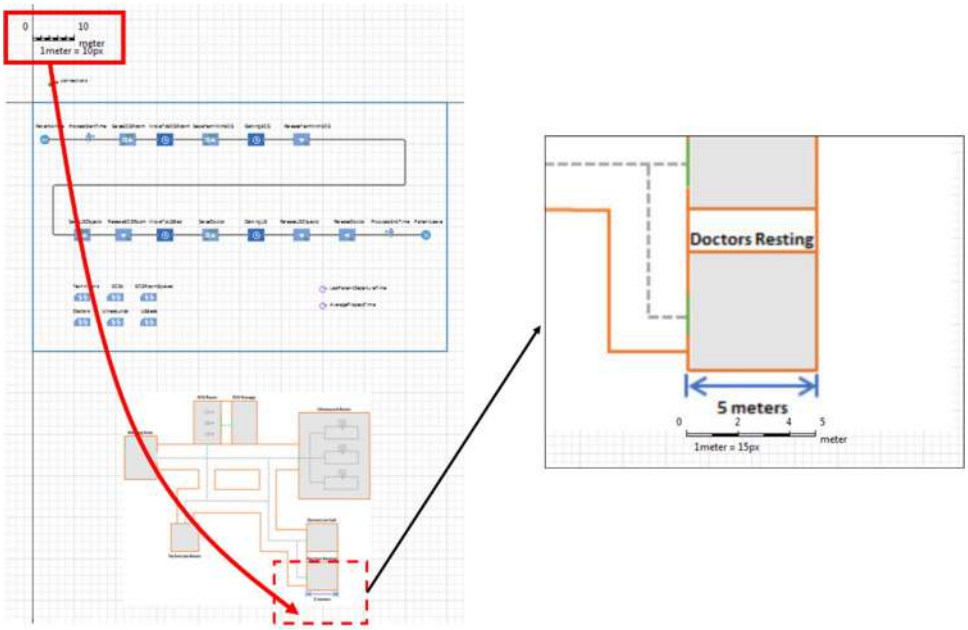


Figure 3-121: Scaling the environment with the help of "Scale" object

5. With the background image imported and the model's environment scaled, it's time to draw the space markups. We're going to draw a network which is a set of nodes interconnected with paths. In the network, a node defines the place where entities or resource units may stay. The PML library has several node types (Point, Rectangular, Polygonal), but we'll only use Rectangular Nodes in this model to represent the rooms and spaces.

   Draw the Rectangular Nodes as shown in Figure 3-123, then draw the paths between them. You can double-click the PML palette's Path Node or Rectangular Node elements to enable drawing mode (Figure 3-122) that allows you to easily draw the relevant objects.

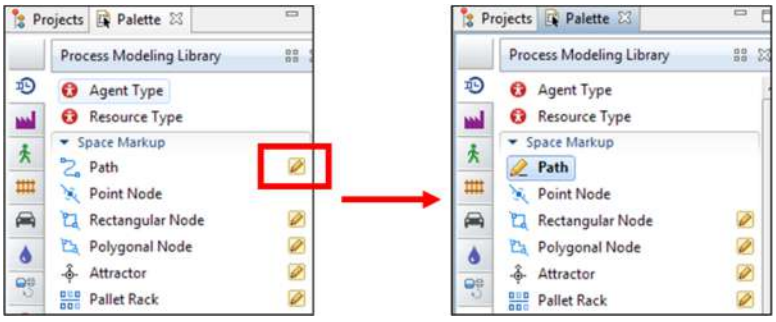


Figure 3-122: Double clicking the pencil icon takes the space markup element to drawing mode

Figure 3-123 and Figure 3-124 show the nodes and paths you need to draw. We're going to rename the Rectangular Nodes according to instructions you'll see in Figure 3-123 and Figure 3-124,but leave the automatically generated path names unchanged.

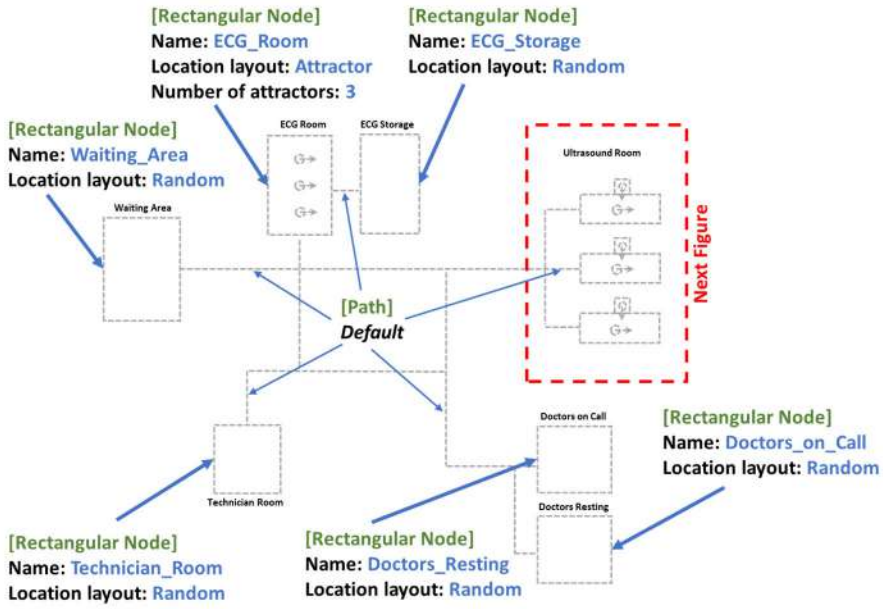


Figure 3-123: Space markups added on top of the background image (Paths and Rectangular Nodes)

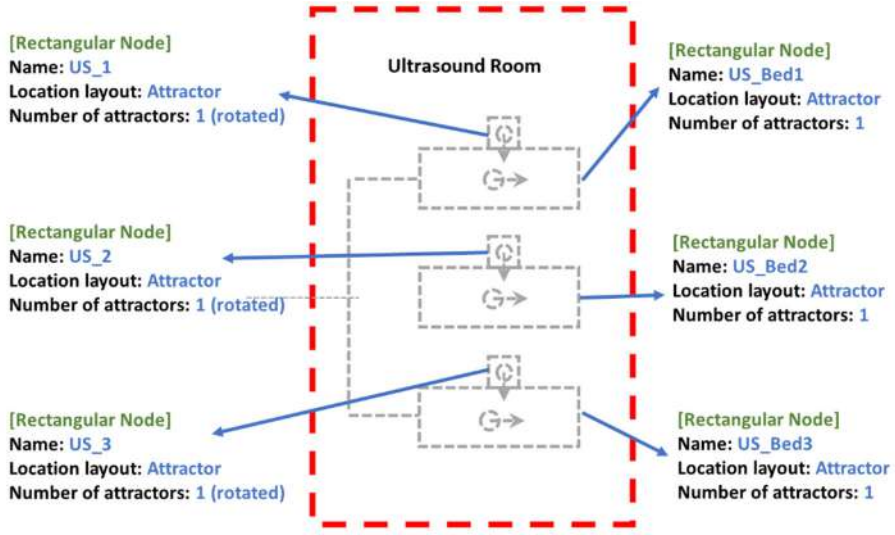


Figure 3-124: Detailed of space markups added in the zoomed area

One important setting we must add to the Rectangular Nodes is their inner layout. In each Rectangular Node's properties, you'll find the layout's default setting is **Random**. You should keep this default for all nodes except **ECG_Room, US_Bed<1,2,3>**, and **US_<1,2,3>**. In these nodes, you'll want to set the layout based on attractors: click the node, click the **attractors…** button in its properties and you'll have three options. For the **ECG_Room**, select the **Number of attractors** and enter 3 in the field. After you click **OK**, you'll see three attractors inside the node (Figure 3-125). You must do the same for the **US_Bed<1,2,3>** and **US_<1,2,3>** nodes, but you'll only have one attractor in each of them. It's important to note that each attractor has an arrow that defines the orientation of entities or resource units located in that attractor. The attractors inside the **US_<1,2,3>** nodes will need to be rotated 90 degrees so that they point toward the ultrasound

beds (Figure 3-125). You can read more about the attractor by searching for “Attractors in nodes” in AnyLogic help.

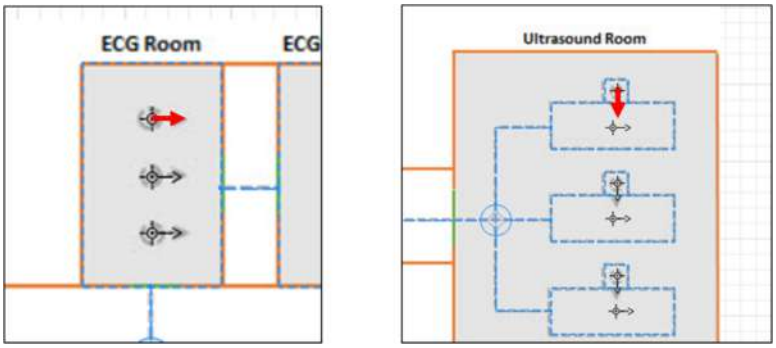


Figure 3-125: ECG_Room node with three attractors (left); US and US_Beds nodes with one attractor (right)

When you are connecting nodes to paths, AnyLogic will automatically build a network from the connected elements. In the **Projects** panel, if you click **Main** and expand the sections under **Presentation**, you’ll see the space markups you added to Main. AnyLogic shows markups that are connected under one network. You can expand each of these elements (the networks) by clicking the triangle to the left of the element’s name (Figure 3-126).

A collection of connected node and paths (that is, a network) defines the physical space for the movements of entities and resources. The start and end points of movements are always at nodes and movement happens along the shortest path between the origin and the destination nodes. One important caveat as you use the PML to move entities or resource units is that segments (or sections of each path) have unlimited capacity and entities that move along a segment aren’t aware of others. In other words, there’s no spatial awareness in the models built with PML and moving entities or resource units can pass through one another.

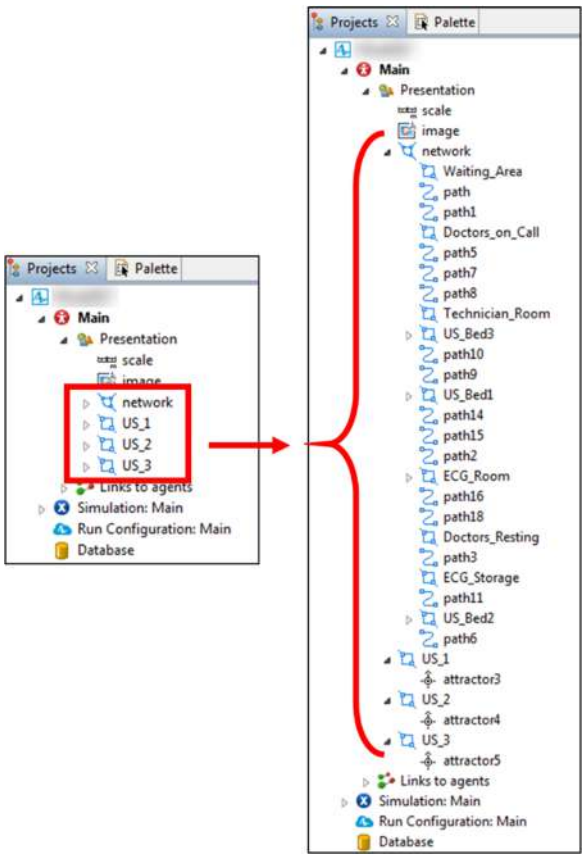


Figure 3-126: Space markups in the Presentation section of Main agent type (in Projects panel)

When you build models with physical movements, you should explicitly define the initial location of entities and resource units. In other words, when entities or resource units enter the virtual world you've created in your model, they should know where they are. And while it doesn't apply to this model, speed of moving entities can be changed dynamically.

6. Now we'll revisit the resource pools and modify them to incorporate Home location (their initial location) and speed (for moving resources). As shown in Figure 3-127, we modified the **Resource type** of **ECGs** to **Portable** to indicate these units can be moved, but not on their own. We set the Technicians and **Doctors** pools to **Moving** which means that they can move on their own or be moved by others. We also set the **Speed** of both resource pools to 1 m/s, which means they will move through the network at this speed. You should set the rest of the resource pools to **Static** to ensure they stay at their home location and prevent others from moving them. Their physical space representation (their space markup) defines the capacities of the following resource pools:

    - **ECGRoomSpaces**: capacity is defined by the home location (rectangular node named **ECG_Room**). We also checked the option that specifies the capacity **Based on attractor**. This means this resource pool's capacity will be equal to the number of attractors in the space markup of its home location. This pool will have three resource units.
    - **Ultrasounds**: Like **ECGRoomSpaces**, the ultrasounds' capacities are set by their home location, specified with a rectangular node. However, each of the pool's resource units has its own rectangular node (with one attractor inside). This means a set of rectangular

nodes (**US_<1,2,3>**) defines the home locations. We'll discuss why their order is important later. For now, make sure they're in alphabetic order.

You may have noticed we didn't check the **Based on attractors** option although each of the rectangular nodes had an attractor. Doing this specified we only want one resource unit per space markup (regardless of the number of internal attractors). Granted, since there's only one attractor inside each rectangular node in this example, checking the **Based on attractors** option would have resulted in the same number of units (three). You'll see how we use this attractor adjust the orientation of the ultrasound machines.

- **USBeds**: set them up like the **Ultrasounds** resource pool (discussed above); in addition, make sure the order of the space markups defined for **Home locations (nodes)** matches the order you see in Figure 3-127.

In the three resource pools described above, the **Show default animation** checkbox is clear to prevent the default animation for the units from showing. For **ECGRoomSpaces**, this is because they're purely logical units. For **Ultrasounds** and **USBeds**, we only want to know their physical location; we don't want an associated animation.

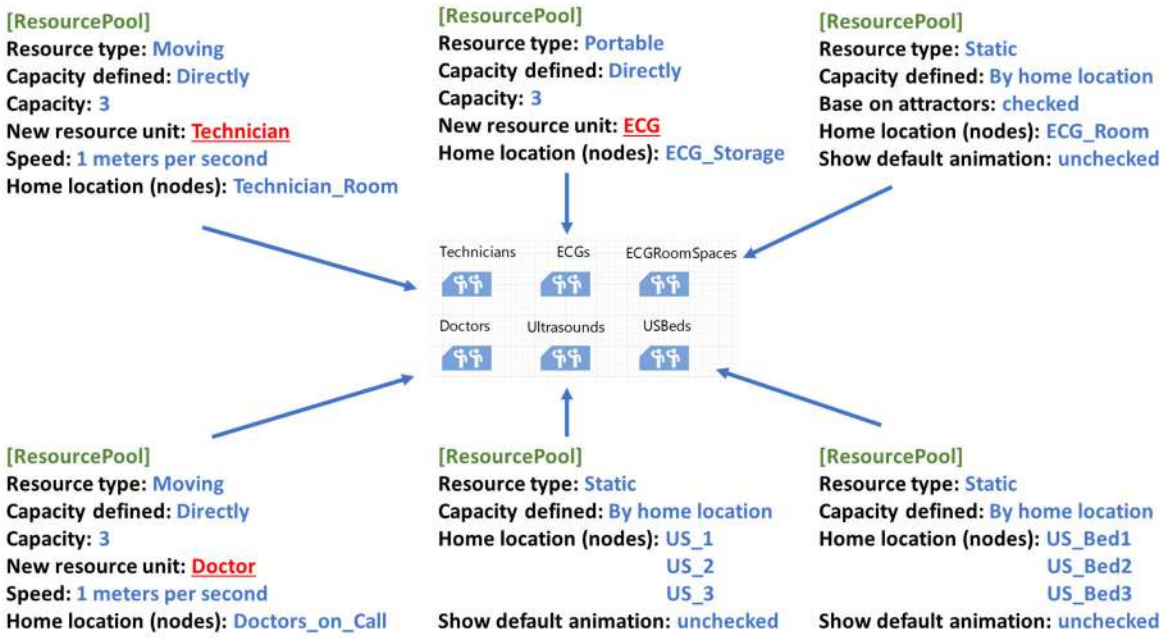


Figure 3-127: Modifications to the Resource Pools to incorporate movements and spatial aspects

7. In this step we're customizing the animation (icon) which will represent the resource units by building a template (blueprint) for each type of resource unit. For example, since all doctors have a similar 3D object for their animation, we can build a template for a single doctor with a specific 3D object and tell AnyLogic to use it to build all the doctors.

   In this chapter, we're substituting the AnyLogic's generic Agent type with a custom agent type – however at this point, it is NOT important for you to understand the importance or implications of this. Possibly unbeknownst to you, we have been using agents all along without mentioning it. All entities and resource units are agents in AnyLogic by default. At this point, all you need to know is

we're substituting the default template (generic blueprint) of resource units with custom ones that have 3D object animations (which are provided by AnyLogic).

We're going to build custom animations for **Doctors**, **Technicians**, and **ECGs** resource pools. Any generic resource pool has a setting in its properties window named **New resource unit** (Figure 3-128). If you click the blue **create a custom type** link below it, AnyLogic will display a wizard that allows you to build your custom blueprint step-by-step.

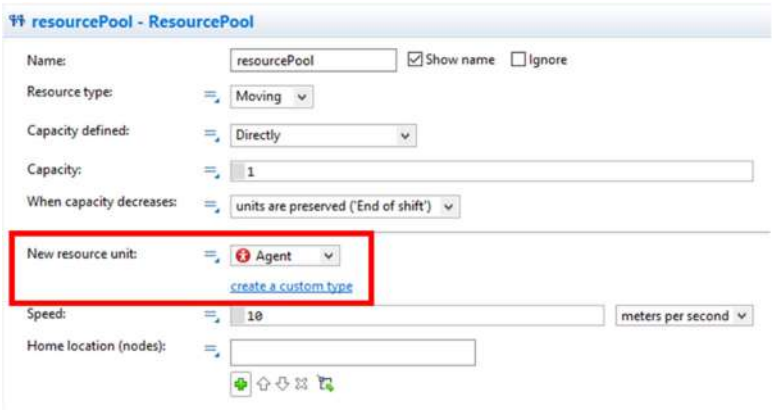


Figure 3-128: New resource unit and the link to "create a custom type" in resource pools

Start by building custom 3D animation for doctors, by first clicking the **Doctors** resource pool, then on the **create a custom type** link, and follow the wizard as shown in Figure 3-129. At the start of the wizard, you must specify this blueprint's name (Agent type). Here, the name should be **Doctor**, which meets the naming convention standard that requires an uppercase letter for the first letter of agent type's names. Click **Next**, select the Doctor 3D object under the **Health** category and click **Finish**.

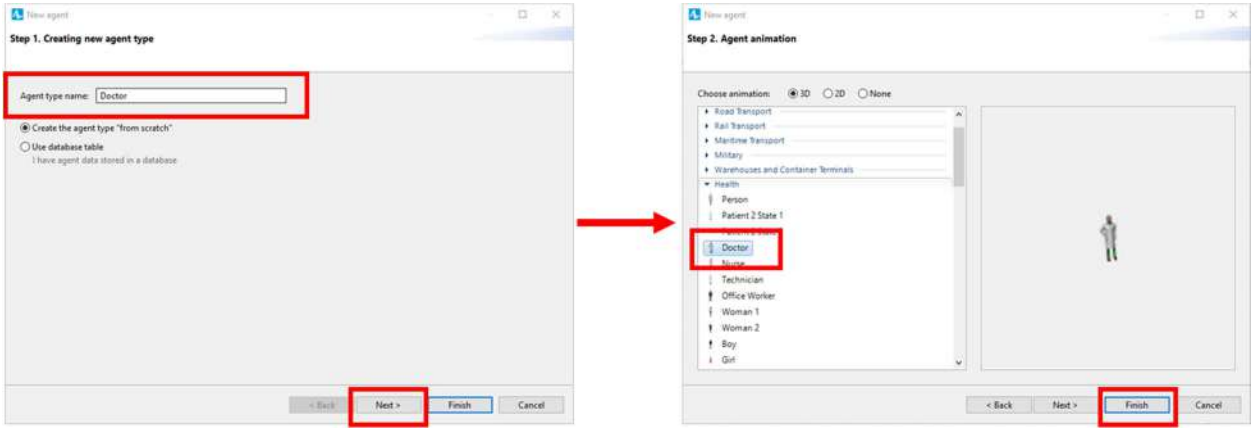


Figure 3-129: Building the Doctor agent type with the wizard

AnyLogic now automatically opens the new agent in a new tab named **Doctor**, and shows you the internals of this new agent type. If you close this tab, it can be reopened by looking in the **Projects** panel for the **Doctor** entry and double-clicking it.

In the graphical editor for the **Doctor** agent type, the 3D object used to represent each resource unit is in the upper left corner of the blue frame and is the origin for the X and Y axis in AnyLogic 2D space. Click the icon and you'll see its properties, which includes its current orientation and the colors of its different materials. To change the color of doctors' scrubs, change the color for **Material_1_surf** to green (Figure 3-130).

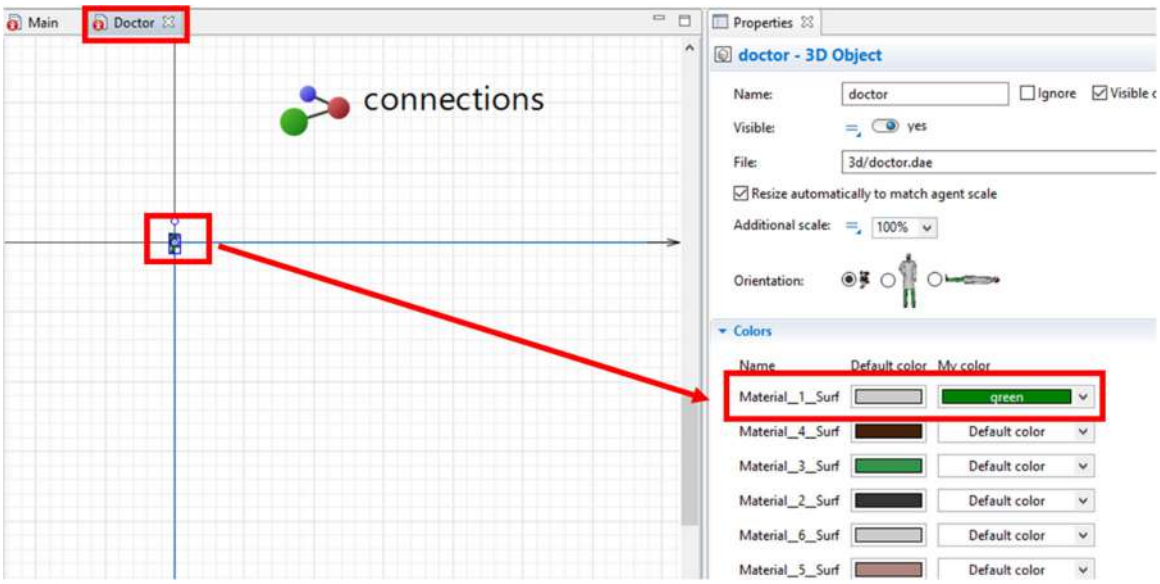


Figure 3-130: Changing the color of doctors' scrubs to green

Return to **Main** and go through similar steps to build the custom animations for two other resource pools:

- Technicians: First click the Technicians resource pool, then click the option for **create a custom type** in its properties. Name the agent type "Technician", click next, select the **Technician** entry under the **Health** category, and click the finish button. From inside the newly created **Technician** agent type (which should open automatically), click the 3D object and change the technicians' scrubs to blue by setting the material **MA_material_0** to blue.

- ECGs: Repeat the same steps as for the Doctors and Technician pools, but for the **ECGs** pool. Name the agent type "ECG" and for the 3D object, use the entry for **ECG** under the **Health** section. Keep all materials in this agent's 3D object set to their default colors.

If you now look in the Projects panel, you should see the three new agent types (**Doctor**, **ECG**, and **Technician**) alongside Main (Figure 3-131).

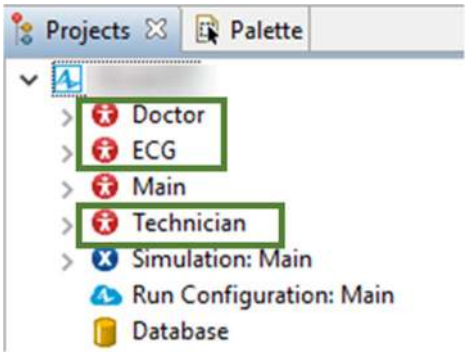


Figure 3-131: Custom agent types added (Doctor, ECG, and Technician)

8. Now return to the original flowchart and incorporate the physical space and movements in it. You need to make changes in three areas: the Source block and in the two main processes in our model (getting ECG and getting US).

   Source: As we mentioned, in a model with physical space, new entities should know their initial location when they enter the model. In most cases, we use the **Source** block to set their **Location of arrival** and **Speed** values (**PatientArrival** in Figure 3-132). We also want to customize the animation that represents incoming entities (patients). To do this, we must build a blueprint like the blueprint we built in the previous step for resource units.

   a. Click the **PatientArrive** block
   b. In the **Properties** panel, click **create a custom type**.
   c. In the wizard, name the agent type **Patient** and click **Next**.
   d. Under the **Health** section, select **Patient 2 State 1** icon.
   e. Click **Finish**.

   AnyLogic should add a Patient agent type in the Project view alongside the custom types and Main. It will also be automatically selected within your **Source** block.

   Getting ECG: In this process, we're substituting the **Delay** (named **MoveToECGRoom**) with a **MoveTo** block. **MoveTo** blocks move the entity from its current location in the network to the destination we set in the properties of the **MoveTo** block. Delete the delay block, add a **MoveTo** block, naming it **MoveToECGRoom** (same name as before). In its properties, set the **Destination** to **Seized resource unit** and the **Resource** as **ECGRoomSpaces**. We want the patient to move toward the seized unit, a space in ECG room (unit from the **ECGRoomSpaces** resource pool) which we seized in the **SeizeECGRoom** block. In the **SeizeTechWithECG** block we seized a set of an **ECG** and a **Technician**. By selecting the option for **Send seized resources** to the **Destination** that is set to **Agent**, we're telling the seized technician to move toward the patient (Agent). As you may noticed, we seized the ECG machine; but since these units' type are set to **Portable** (Figure 3-132), they can't be sent (or move on their own). If ECG units were of **Moving** type, they too would have moved toward the patient.

   Getting US: Like the "Getting ECG" process, we substitute the **Delay** block, which represents the movement of the patient to the US bed, with a **MoveTo** block. In this new **MoveTo** block (named

MoveToUSBed) we set the **Destination** to **Seized resource unit** and the **Resource** to **USBeds**. By doing this, we tell the entity to move toward the US bed unit seized in the **SeizeUSObjects** block. In the **SeizeDoctor** block, check the **Send seized resources** checkbox, set the **Destination is** field to **Other seized resource unit**, and the **Resource** as **Ultrasounds**. These settings tell the seized doctor to move toward another, already-seized unit (an ultrasound). In doing this, it becomes clear why it was important for the home location nodes inside Ultrasounds (**US_Bed1,2,3**) and **USBeds** (**US_1,2,3**) to be ordered correctly – it's to make sure the **SeizeUSObjects** block seizes the correct pair of bed and ultrasound; otherwise, the doctor would go to the wrong US which the patient isn't in the bed for.

To clarify, when a doctor moves toward the patient, he goes to the location of one of ultrasound machines (one of rectangular nodes named **US_1**, **US_2**, or **US_3**). To simplify the model, our design ensures when the doctor needs to go to the ultrasound room, he goes to the ultrasound machine's location. In reality, the doctor will be near the machine rather than at the same location.

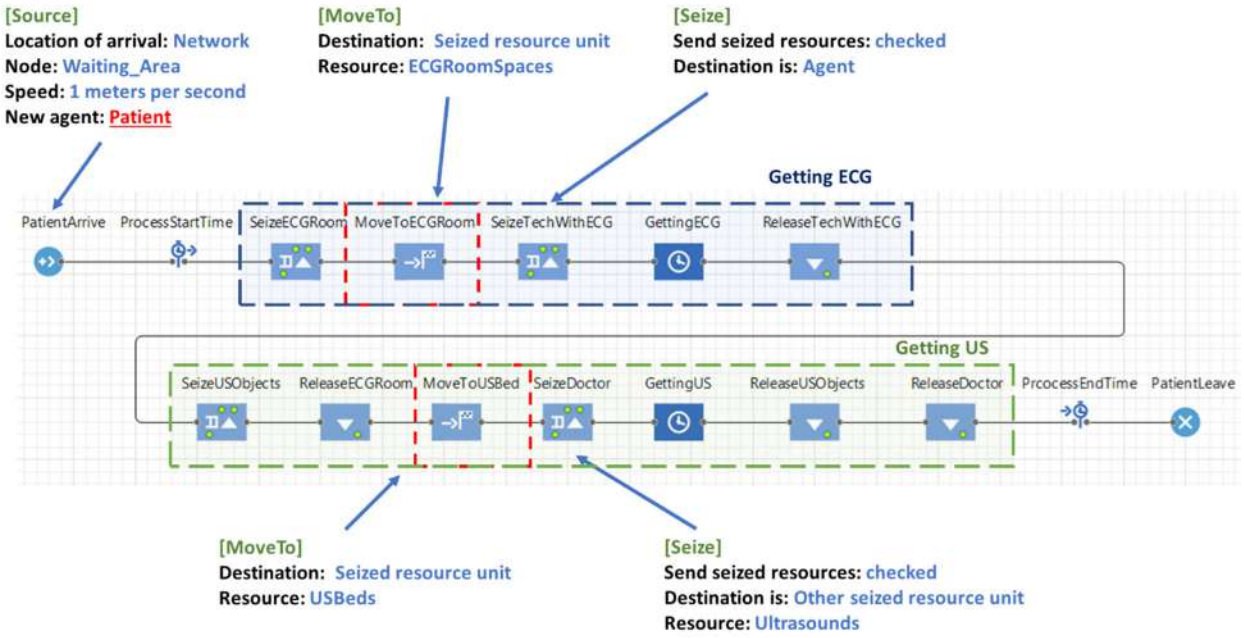


Figure 3-132: Modifications to the model logic to incorporate movements in the physical space

9. To better organize the model's logic and animations, we can use the Presentation palette to add three **ViewArea** objects. To do so, drag-and-drop three of them and set their properties based on Figure 3-133. By default, these view area objects will have the same size as the default blue frame (1000 pixels wide, 600 pixels tall). Make sure the process logic and 2D animation are inside the two view areas on the left.

   To have a 3D projection of the animation, drag-and-drop a **3D Window** object from the Presentation palette. Use its properties to set its width to 1000 and its height to 600 (the same as default blue frame). Move the **3D Window** in the third view area object. As you can see in Figure 3-133, you can assign labels to view areas. During runtime, these labels will display instead of the view areas' name.

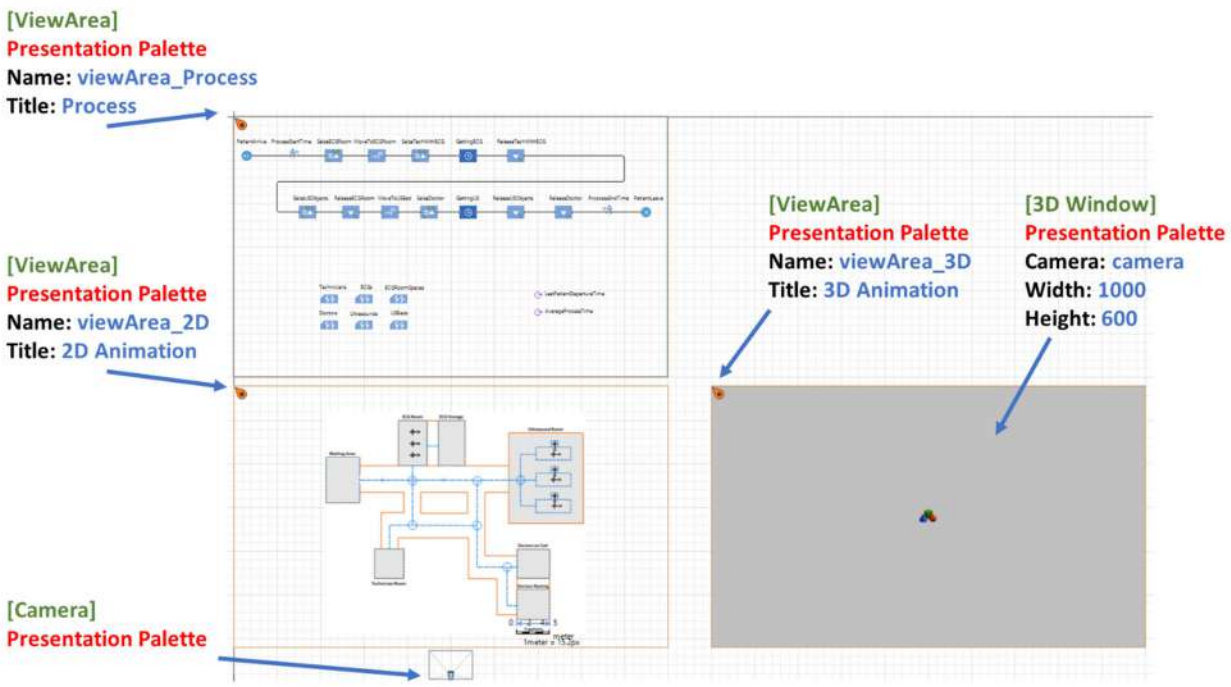


Figure 3-133: View areas, 3D Window and the Camera

Finally, we'll setup a preset location and angle for the 3D view by assigning a **Camera** to our 3D window. From the Presentation palette, drag-and-drop a **Camera** object, positioning it below the 2D animation. Then, with the camera selected, click-and-drag the triangle near the circle's radius (indicating its point direction) and rotate it toward the 2D animation area (Figure 3-134). Lastly, we need to connect this camera to the 3D Window. Click the **3D window** and select **camera** in the **Camera** drop down menu.

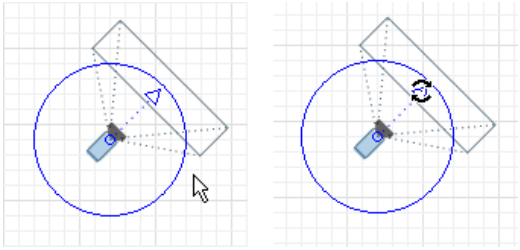


Figure 3-134: How to rotate the Camera object

10. Run the model and look at the different view areas. To switch between them while the model is running, open the developer panel by clicking the bottom-right button to toggle the **Developer** panel. After you open the **Developer** panel, you can click the view area dropdown to select the area you want to focus on (Figure 3-135).

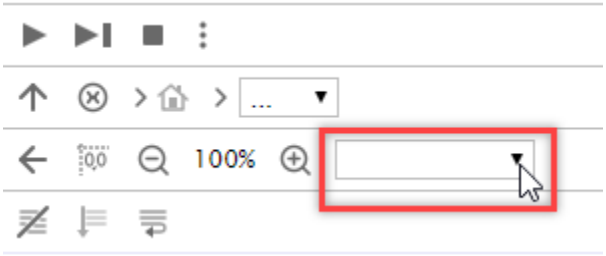


Figure 3-135: Dropdown to select desired view area

Switch to the 3D Animation view (**viewArea_3D**) and in the 3D window move the camera around to find a good angle and position:

- To move the camera left-click and drag the mouse in the desired direction.
- To zoom in and out of the 3D scene rotate the mouse's wheel.
- To rotate the scene, hold ALT key (Mac OS: Option key) then click-and-drag the mouse and left-right (rotation around Z axis) or forward and backward (rotation around X axis).

When you find a desirable location and camera angle, right-click the **3D window** (in the running model) and select **Copy camera location**. Return to the development environment (not in the running model), click the camera object and within its properties, click the **Paste the coordinated from clipboard** button (Figure 3-136). The next time you run the model, the 3D window will remember the angle you've set and show the 3D presentation from that viewpoint.

Figure 3-136: Customizing the camera's start location

11. Run the model till the end and look at the operation in both 2D and 3D (Figure 3-137). Take note of the changes from (partially) incorporating the physical movements of doctors and technicians.

From the standpoint of the model's statistical outputs, we didn't make major changes in this version. By substituting the logical **Delay** with a physical **MoveTo** block, movements went from taking a prespecified time (a random variable) to relying on AnyLogic to calculate the time based on the moving entities' speed and the distance in paths between the start and destination nodes.

The only added time to the operation in this version (compared to the first) is the time doctors and technicians need to return to their home location before attending to the next patient. These added movements will not significantly affect the operation; on average, we expect staff utilization, average

processing time, and last patient departure time to be slightly higher. You should know this model ends in a state that the system is very well within transient state. This means each sample path (result of each run) will be very different from the others. The results of these single runs aren't suitable for meaningful comparison between different scenarios.

Our objective of adding physical space in version 2 wasn't to compare it with the first version, but rather to show different abstraction levels (specifically, with and without physical space as part of the model logic). In version 3, we'll complete version 2 by incorporating the preparation and wrap up tasks the physician and doctors need to complete before and after seeing a patient.

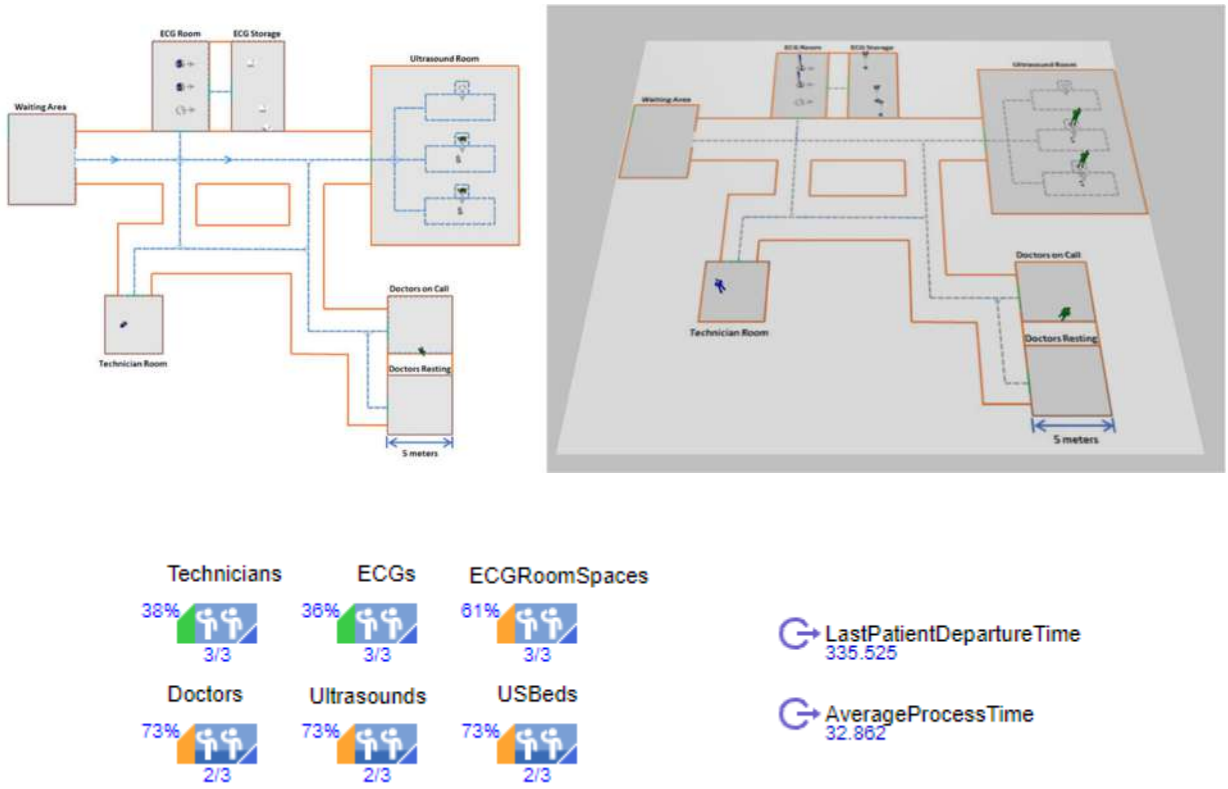


Figure 3-137: Outputs of the model performance metrics, showing 2D and 3D animation

Version 3 (low abstraction level): Physical space; 50 patients admitted daily with staff consisting of three physicians and three doctors

In this version, we'll complete the model we started in version 2. We'll incorporate the preparation of the technicians (picking up an ECG machine and bringing it to the patient) and their wrap-up task (returning the ECG machine and going back for a rest). We'll also incorporate the wrap-up task of doctors (going to the rest area before attending another patient). We'll do all these tasks using *preparation* and *wrap-up* branches that will occur in parallel with the primary flowchart (getting ECG and US) as shown in Figure 3-138.

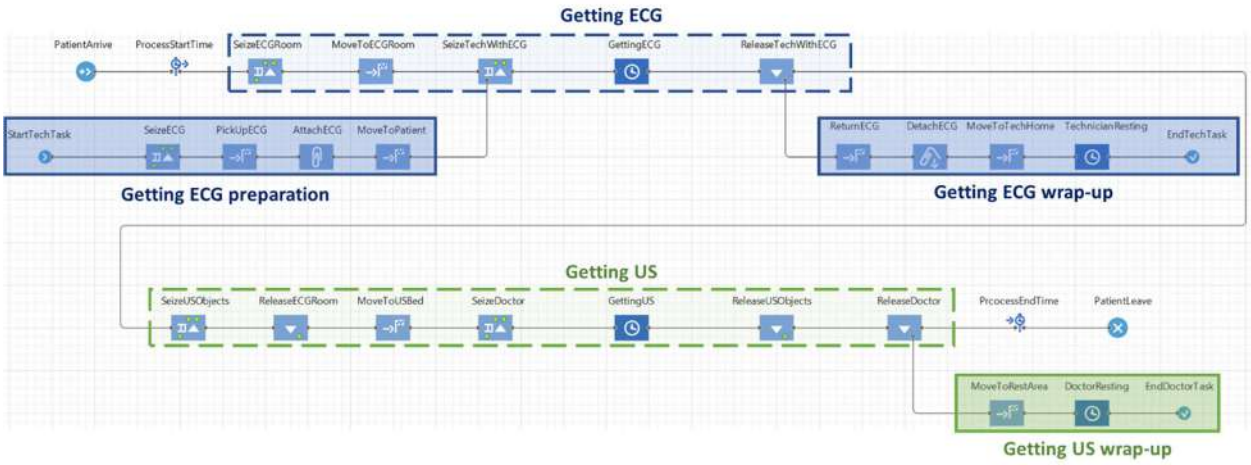


Figure 3-138: Primary flowchart with its preparation and wrap-up flowchart branches

1. Open the version 2 of the clinic model built in the previous section. From the **File** menu, select **Save as…** and save the model with a different name to keep the original model intact.

2. We'll add a preparation process branch and connect it with the "*preparedUnit*" port (bottom) of the **SeizeTechWithECG Seize** block (Figure 3-139). Due to the upcoming changes, we'll no longer seize ECGs alongside the technicians. This means we need to change its seize setting to **units if the same pool** and one technician. Since we'll model the technician movement in the preparation and wrap-up branches, we must make sure the **Send seized resources** option is clear.

   The preparation branch starts with a **ResourceTaskStart** block. One important caveat here is the technician (resource unit) seized by the patient in the **SeizeTechWithECG** block passes through the preparation process branch. If we change our perspective to the viewpoint of this resource unit, it behaves like an entity in the context of the preparation branch.

   > *In the preparation and wrap-up branches, the resource units are the main things that are passing through. This means we could imagine these resource units as entities while they're in a preparation or wrap-up branch. Although, from the perspective of the entities in the primary branch, they're still resource units.*

   After entering the preparation branch via the **StartTechTask** block, the technician seizes an ECG machine. We then attach the seized unit (the ECG) to the entity (technician) with the help of a **ResourceAttach** block. Because of this attachment, the technician and the ECG are move together in the subsequent **MoveTo** block, **MoveToPatient**. The destination setting is chosen to be the **Agent which possesses me** – this means the technician and attached ECG will move toward the patient in **SeizeTechWithECG** block, which is the entity that seized it. The settings also sets an offset so the technician moves to a location to the patient's right. The assigned offset is -15 pixels, which is negative due to the direction of the x-axis and approximately 1 meter to the left, based on our scale object properties.

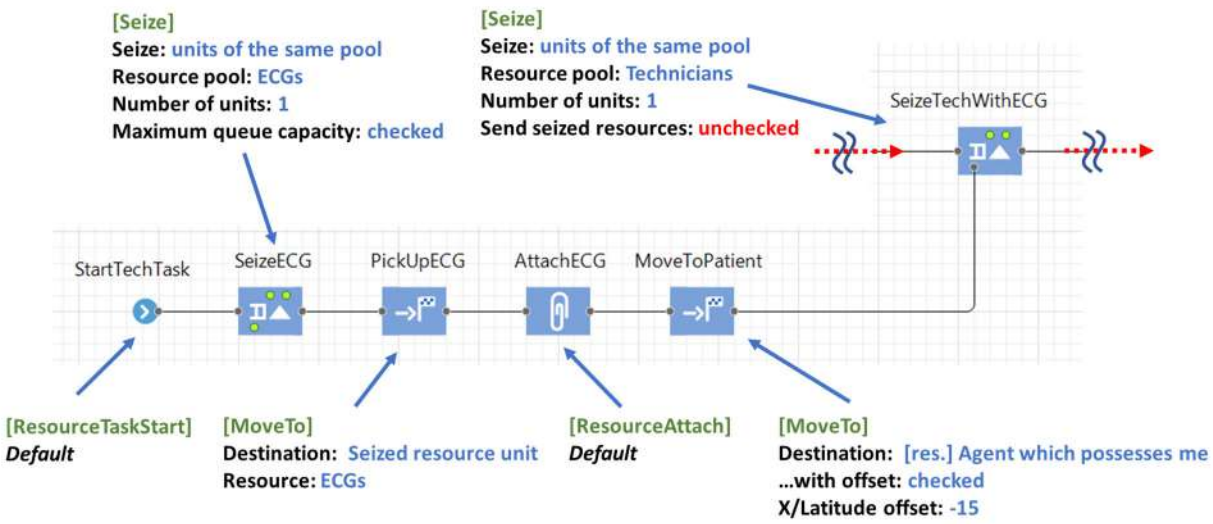


Figure 3-139: Preparation process branch for technicians

3. Now, we shift to the wrap-up branch for the technicians (Figure 3-140). In the **ReleaseTechWithECG**, we ask the released resource (the technician) to stay at its location after the release instead of returning to home location.

   That's because we're going to explicitly model the returning of the technicians in the wrap-up branch. As you know in **ReleaseTechWithECG** we released the resource we seized in **SeizeTechWithECG** which is the technician. The released technician then passes through the wrap-up process branch connected to the "*wrapUp*" port of **ReleaseTechWithECG**. Like the preparation branch, the resource unit that passes in the wrap-up branch behaves like an entity in the context of the wrap-up process. The **ReturnECG** block moves the technician with its attached resource (ECG) back to the home location of the resource. Then we detach the technician from the ECG in **DetachECG** block. The technician alone will move to the **Technician_Room** which is the rectangular node in the network representing the home location of technicians. A returned technician then rests for 5 minutes in the **TechnicianResting Delay** block.

   One important caveat: the entity (patient) leaves the Release block ("ReleaseTechWithECG") immediately. However, the released resource unit (the technician and the ECG machine attached to it) will continue to perform the assigned tasks in the wrap-up branch.

   After the delay, the technician reaches the **ResourceTaskStart** block (named **EndTechTask**) and will become an idle resource unit (in the **Technicians** resource pool) which will be available to be seized by the other patients in line. We cleared the **Move resource to its home location** on the properties of the **EndTechTask** because we returned the technician to the home location in the warp-up branch (via **MoveToTechHome**).

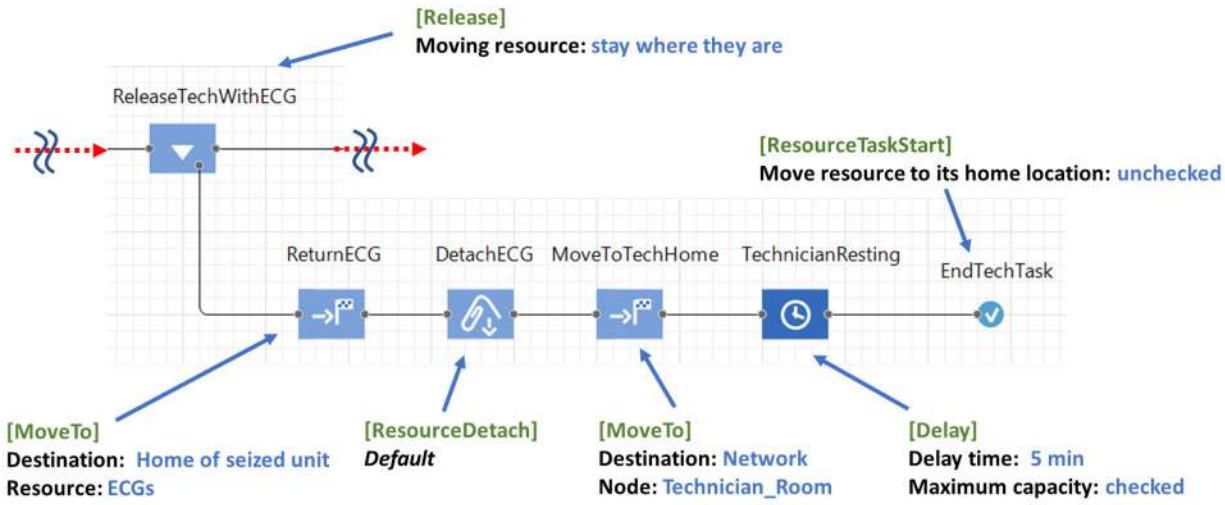


Figure 3-140: Wrap-up process branch for technicians

4. To build the wrap-up process of doctors after they're done with ultrasound procedure, reference Figure 3-141 below. Start by going into the properties of the **ReleaseDoctor** block and for the **Moving resources** option, select **Stay where they are** – this is done since we want to take control of where the released doctors go during the wrap-up branch. In the **MoveToRestArea** block, move the released doctor to the **Doctors_Resting** node (this node isn't the home location of **Doctors** resource pool but rather the location of doctor's resting room). With the **Delay** block, the doctors spend some time (between 10 to 15 minutes) in the resting area before returning to their home location. Returning to the home location isn't included in the flowchart because it's the default setting in the properties of the **ResourceTaskEnd** block, **EndDoctorTask**.

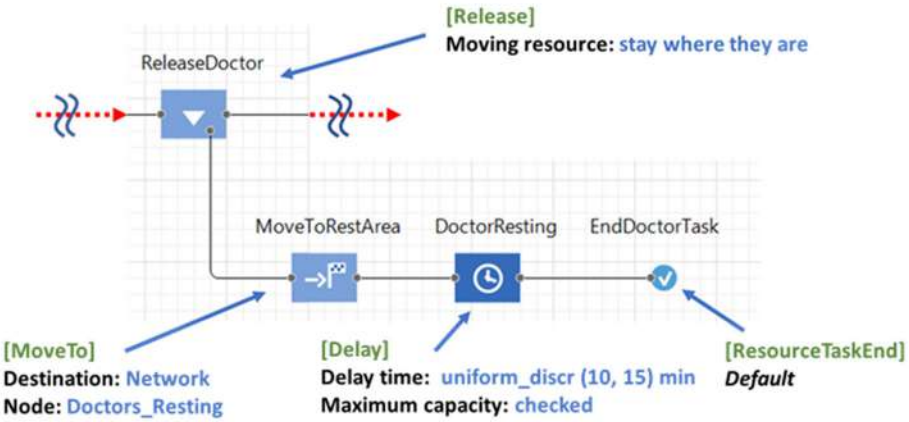


Figure 3-141: Wrap-up process branch for doctors

5. At this step, the model logic is complete, but we'll add some small aesthetic features to enhance the animation and operation the visualization. The first thing we'll do is adjust the distance between the technician and the attached ECG. As we've discussed in step 3, the **AttachECG** block will attach the ECG machine to the moving technician. However, as shown in Figure 3-142,

AnyLogic by default puts the attached resource unit behind the entity that seized it. This works in most cases since the resource unit is usually is the moving thing that moves the entity (for example, a forklift that moves a pallet). However, we want the technician to be behind the ECG machine. To change the default offset, you must add a **PMLSettings** object from the PML library; in its properties, change the setting for **Offset for attached units** to -5. This value is in pixels and, based on our scale, is approximately 30 centimeters or one foot.

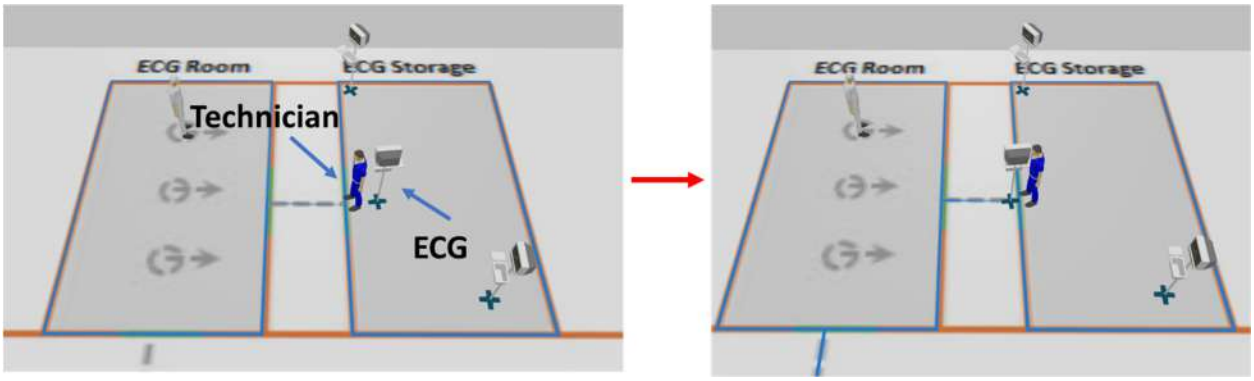


Figure 3-142: Before (left) and after (right) of adjusting the offset in PMLSettings for the ECG attached to the Technician

Next is to add 3D objects to the 2D plan to have a visual presentation for beds and ultrasound machines. Drag and drop three **Beds** from **3D Object** palette's **Health** section and rotate them to match the shapes in Figure 3-143.

Similarly, for the **Ultrasound Scanner** objects, place them to the right of nodes representing ultrasounds (**US_1**, **US_2**, and **US_3**). The default orientation of US objects doesn't need change. Take note the 3D objects added in this step are passive object and for aesthetic reasons.

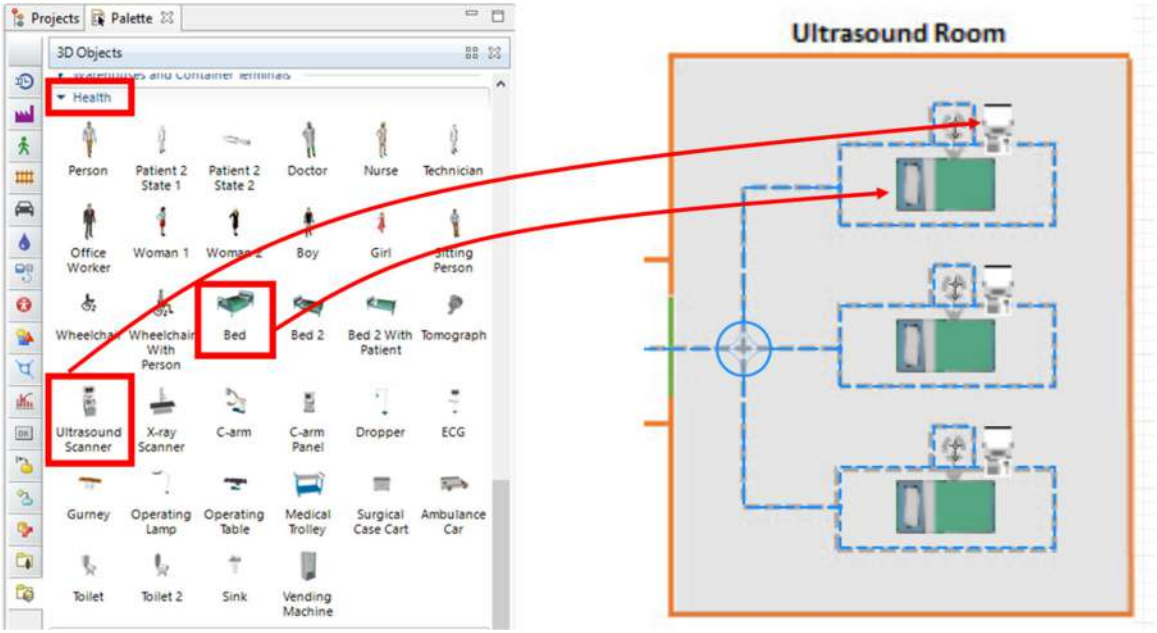


Figure 3-143: Placement of the bed and US 3D objects

Finally, we need to fix the orientation of patients when they reach the US bed (Figure 3-144). We'll use a trick to assign two 3D objects to the patient and change the visibility of either one based on the patient's location. The goal is to have the patient be in the upright position everywhere except the moment he or she reaches the US bed to receive the US procedure.

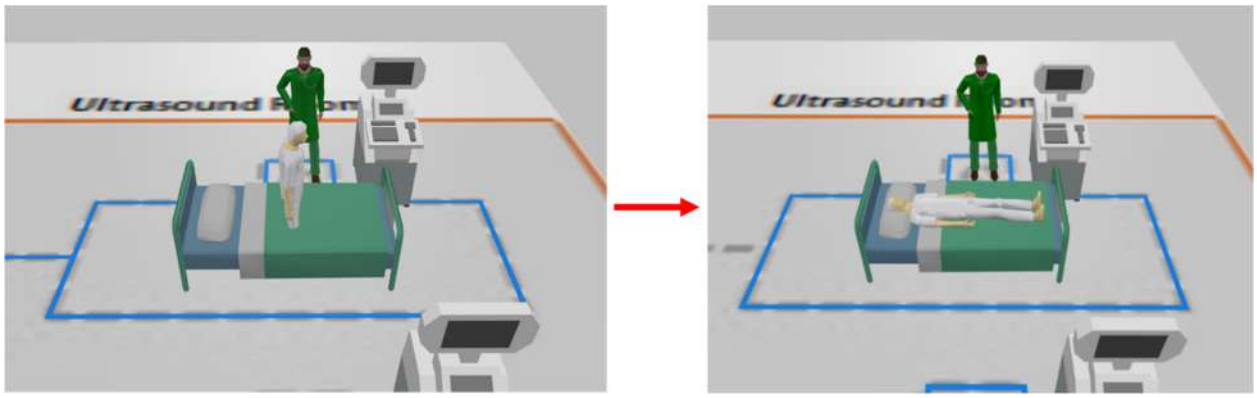


Figure 3-144: Patient is in upright position in bed (left); fixed animation with a patient laying on the bed (right)

To add a second icon (3D object) for the patient, you need to first open the graphical editor of the Patient agent type by double clicking the entry for it in the Projects view (Figure 3-145).

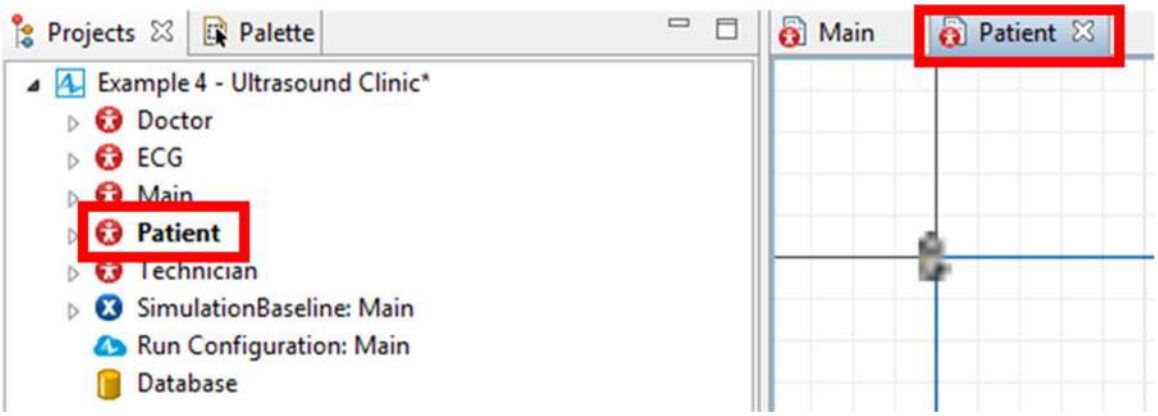


Figure 3-145: Opening the Patient agent type

From the 3D Objects palette, go to the **Health** section and drag-and-drop the **Patient 2 State 2** 3D object to the origin (on top of the previous icon). In the object's properties, set the **Visible** option to **no** (Figure 3-146). Our patient will have two icons representing their animation: **Patient 2 State 1** and **Patient 2 State 2**. By default, we set the **Patient 2 State 2** to be invisible (not visible), but we'll make it visible when the patient reaches the bed. To do so, we need to add two lines of code in the **On exit** field of the **MoveTo** box, **MoveToUSBed** (Figure 3-147):

```
agent.patient_2_state_1.setVisible(false);
agent.patient_2_state_2.setVisible(true);
```

When the patient reaches the bed, this code tells the patient (agent) to make its upright icon (**patient_2_state_1**) invisible and make the alternative icon (**patient_2_state_1**) visible. The

"agent" keyword in this code snippet points to the patient passing through the block in the primary flowchart. We'll get back to the meaning of this keyword and discuss it in more depth in the following chapter. Since the patient will not move again (upon completion, the process removes them from the flowchart), the icon doesn't change.

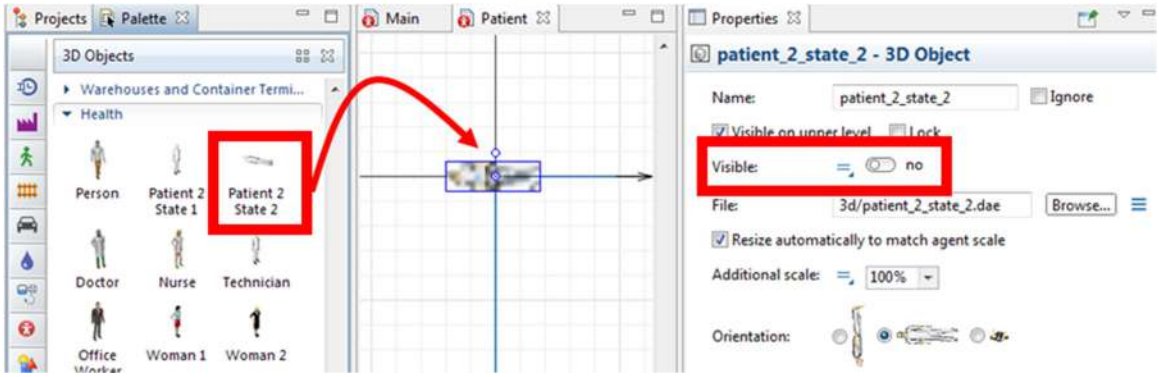


Figure 3-146: Adding the second icon to the Patient agent type and make it invisible by default

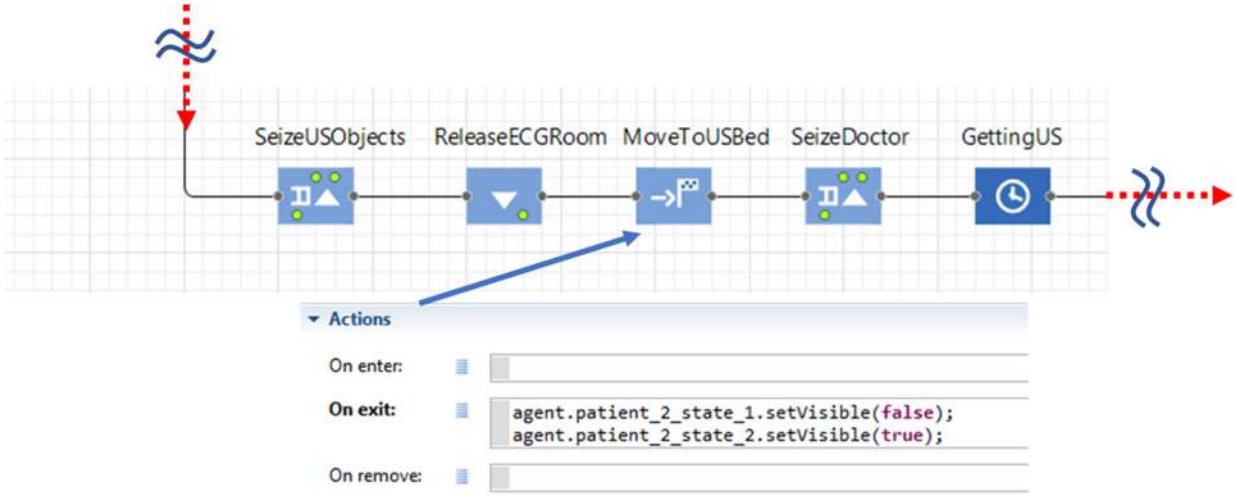


Figure 3-147: Code snippet added to the "On exit" field of "MoveToUSBed" block to change the patient icon

6. As this model grows, it's hard to keep the entire flowchart with the preparation and wrap-up branches in one view area. As shown in Figure 3-148, add a **ViewArea** object (from the Presentation palette) and place the process into two areas.

   You can select multiple blocks at one time by clicking-and-dragging from an empty spot in one corner of the graphical editor to another corner. You can then move the entire selection to the desired position.

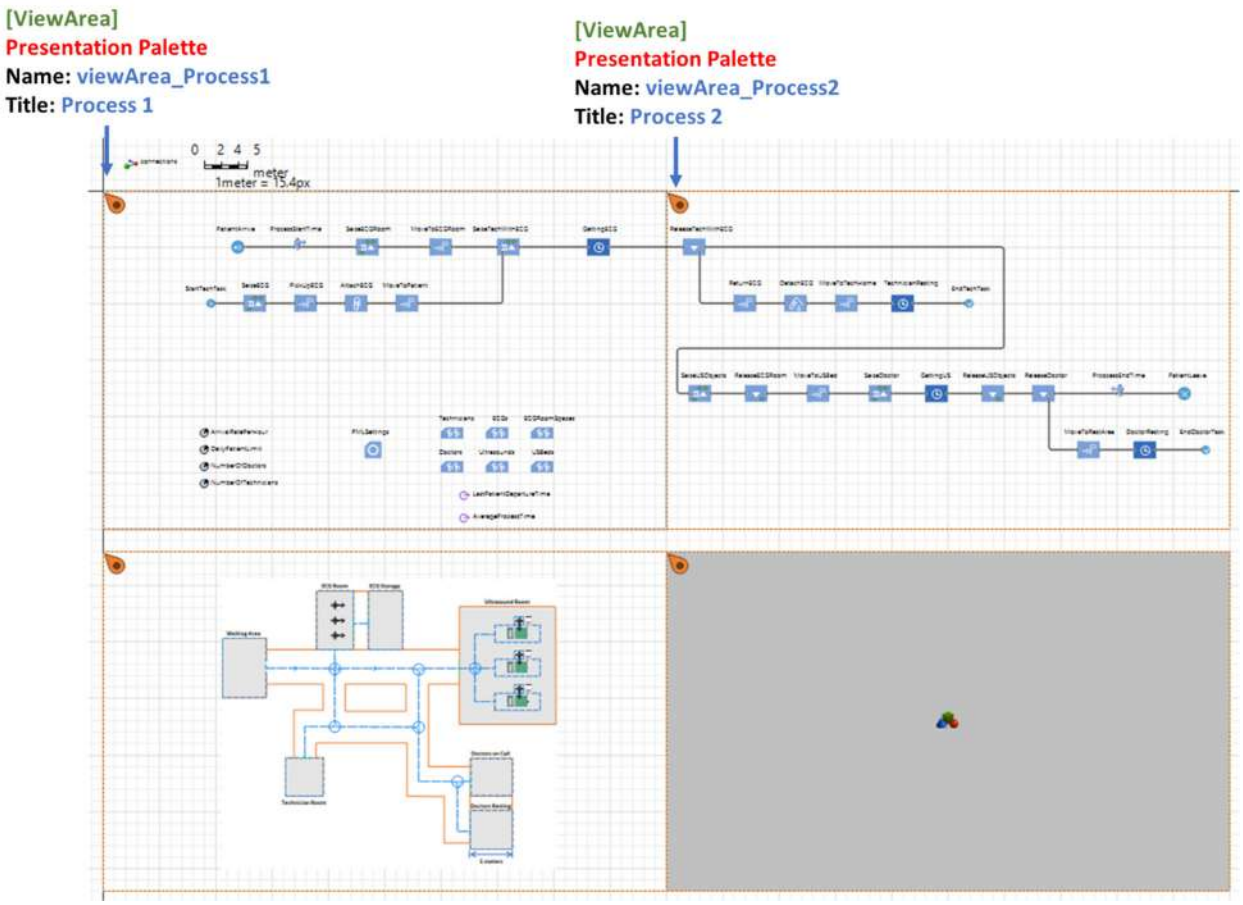


Figure 3-148: Separating the process flowchart into two view areas

7. With the model's logic and animation complete, start by adding a fifth **ViewArea** to the editor. This area will have two graphs that make the model's outputs easier to understand.

   The first graph should be **Bar Chart** from the **Analysis** palette, which we'll use to show the utilization of five resource pools (**Doctors**, **ECGs**, **Technicians**, **USBeds**, and **Ultrasounds**). In the chart's properties, in the **Data** section, click the plus sign five times to add five new bars (one for each resource pool). For each bar, set the **Value** option to call the resource pool's utilization function (for example, `Doctors.utilization()`) and the **Title** option to be an appropriately named description (see Figure 3-149's left side). The `utilization()` method of each resource pool will return the average use of the units in the specified pool.

   Then add a **Histogram** object, which shows the distribution of processing times for the 50 patients the clinic processed during their working hours. For this graph, use the settings as shown on the right side of Figure 3-149.

   Lastly, move the two **Output** objects (**LastPatientDepartureTime** and **AverageProcessTime**) from the process view area to the bottom of the newly added **Stats** view area (bottom of Figure 3-149).

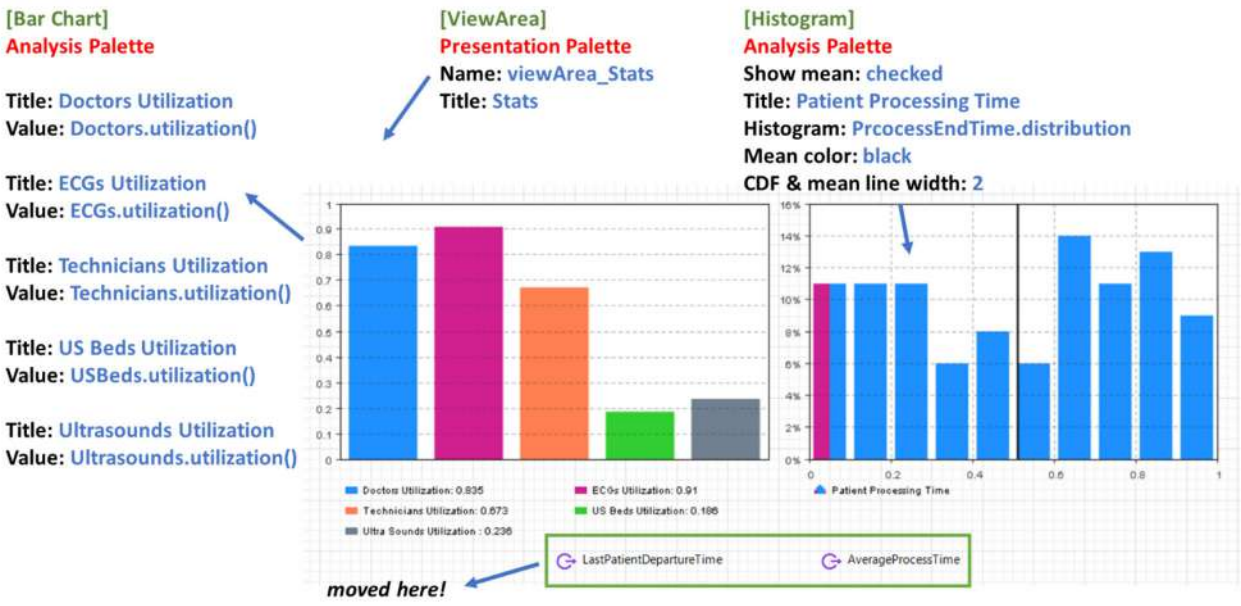


Figure 3-149: A new ViewArea surrounding a bar chart and a histogram to show model outputs

8. Run the model and analyze the outputs in the **Stats** view area; one run is shown in Figure 3-150.

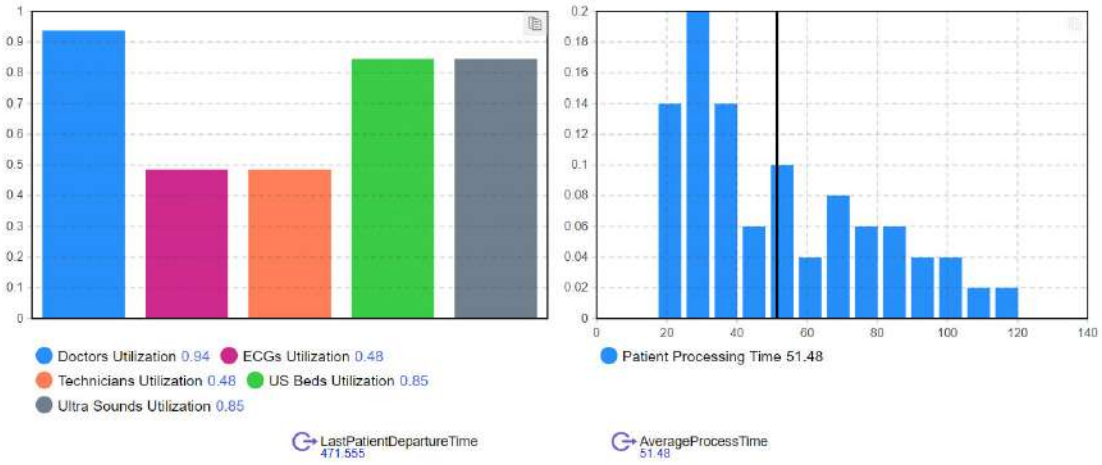


Figure 3-150: Sample run outputs of version 3

Since this model is in the transient state, in your runs with random seeds, you may get very different outputs (for example, **AverageProcessTime** > 100 min). The output of version 3's single run shows us all times and utilizations have increased compared to version 2, as shown in Table 3-19.

Table 3-19: Version comparison of statistics

Name	**Version 2**	**Version 3**
Last patient departure time	335.525	471.555
Average process time	32.862	51.48
Utilization of Technicians	38%	48%
Utilization of ECGs	36%	48%

Utilization of Doctors	73%	94%
Utilization of USs	73%	85%
Utilization of US beds	73%	85%

We expected this increase in the processing times and resource utilizations. This is because Version 3 includes preparation and wrap-up branches ignored in version 2. Keep in mind this is only one run - especially in a terminating simulation, the found results aren't accurate enough to be the basis for a decision. It simply shows us one possible outcome of the operation with specific inputs and configurations.

[Optional] Version 3 alternative (identical outputs to version 3, explicit definition of ultrasound units, explicit preparation process for doctors)

In this section, we'll discuss a variation of version 3 that shows how we can design the US resources and the doctors' preparation process in a more explicit way. We do this to show another way to build the same behavior in your model. However, since this modified version's quantitative outputs are almost identical those in the original version 3, you can ignore this section and jump to the next section.

In the original version 3, the units inside the Ultrasound pool were built by their three home location nodes (**US_1**, **US_2**, and **US_3** nodes) and the **New resource unit** field unchanged from the default agent. The **Show default animation** was unchecked and instead we manually added three **Ultrasound Scanner** 3D objects to the right of the nodes. In our original design, we also used the location of ultrasounds (nodes **US_1**, **US_2**, and **US_3**) as the location of doctors during the US procedure. We essentially added three fake 3D objects to represent the ultrasounds and used the actual location of the ultrasounds to place the doctors near the beds. To better understand why we needed to do this, refer to step 8 of version 2 and how we simplified our model by using this trick.

In this version, we first need to move the ultrasound animation to their actual physical location:

1. Open version 3 of the clinic model built in the previous section. From the **File** menu, select **Save as…** and save the model with a different name to keep the original model intact.
2. Select the three **Ultrasound Scanner** 3D objects and delete them – either by holding Control (Mac OS: Command) and clicking each or by going through them one-by-one (Figure 3-151).

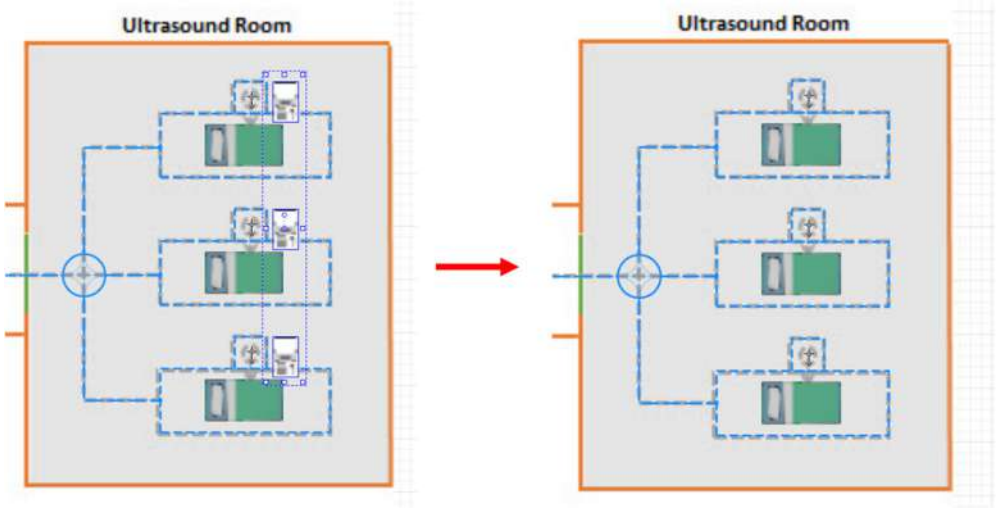


Figure 3-151: Three "Ultrasound Scanner" 3D objects are selected (left) and then deleted (right)

3. Click the **Ultrasounds** resource pool, then on **create a custom type** in its properties. In the opened wizard type "Ultrasound" as the agent type's name, press next, select the **Ultrasound Scanner** under the **Health section**, and click **Finish**.
   Select the ultrasounds nodes (**US_1**, **US_2**, and **US_3**) and move them a little bit to the right. Then click them one by one and rotate their attractors toward right (Figure 3-152). Doing this places the US nodes in the actual location of the US machines and ensures their animation will be shown at their home location node.

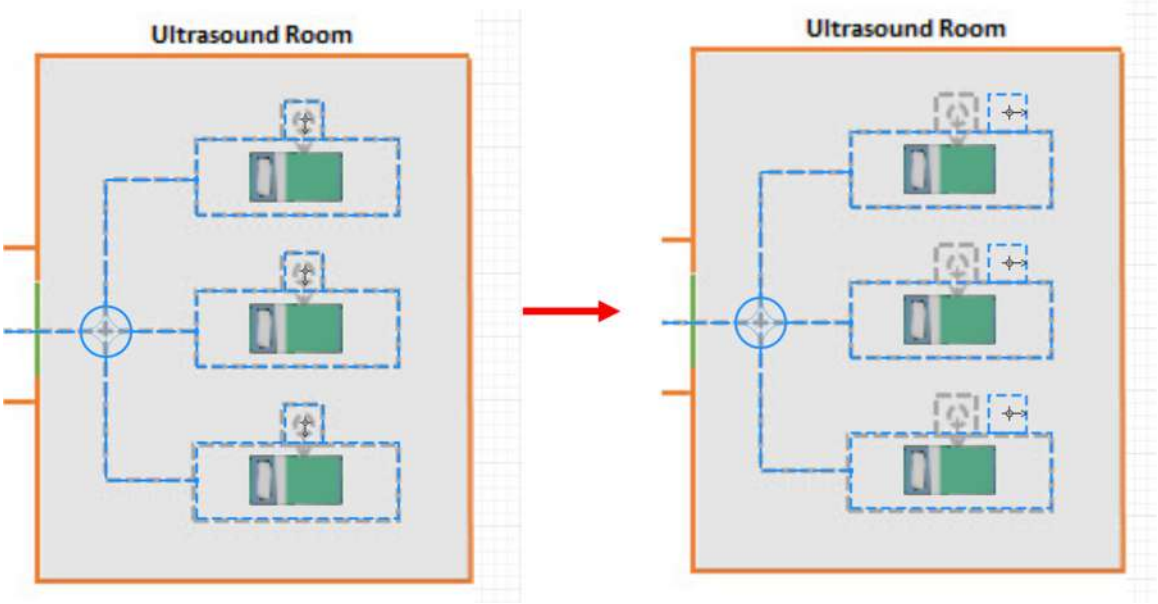


Figure 3-152: Moving the US nodes to the right and rotating their attractors

4. Next, we'll define an explicit preparation flowchart for the doctors. Start by clicking the **SeizeDoctor** block and clearing the **Send seized resources** checkbox. This allows us to define the tasks in the preparation branch (the blocks and their appropriate settings are shown in Figure 3-153 below).

   The doctors' preparation consists of two blocks: **ResourceTaskStart** (**StartDoctorTask**) and **MoveTo** (**MoveToPatientOnBed**). This process moves the doctors to a location 20 pixels (more than a meter) to the right of the seized ultrasound.

   To rotate the doctors toward the laying patient for the duration of the US procedure, we must add the following line of code to the **On enter** field of **GettingUS**:

   ```
   agent.resourceUnitOfPool(Doctors).setRotation(0.5*PI);
   ```

   Since this line of code is beyond what we've discussed, all you need know is it adjusts the animation. If you're curious, the following is a breakdown:

   - "`agent`" is the patient passing through the primary process.
   - "`resourceUnitOfPool(Doctors)`" singles out the doctor (seized from the **SeizeDoctor** block) from the other resource units currently seized by patient (the US bed and US machine are also seized in the **SeizeUSObjects** block).
   - "`setRotation(0.5*PI)`" rotates the animation of the singled-out resource unit (doctor) by 90 degrees.

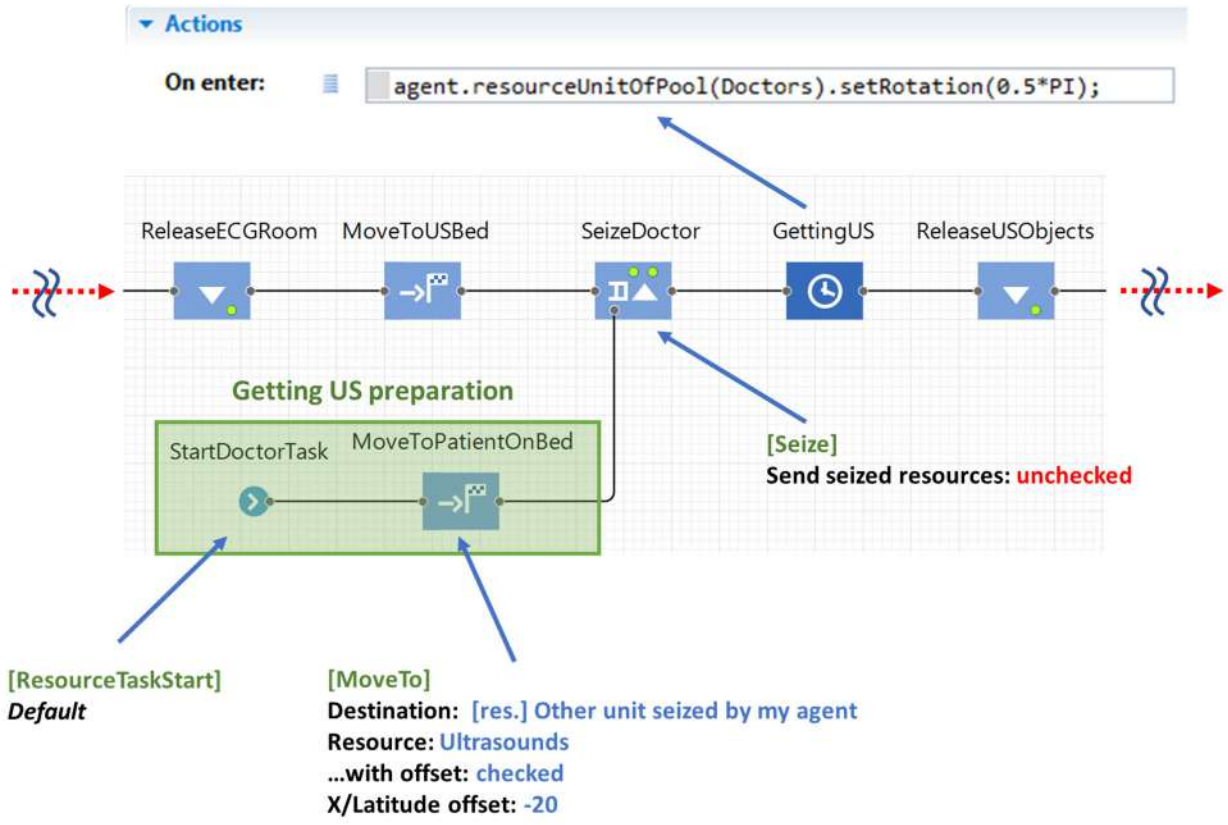


Figure 3-153: Adding the preparation process for the Getting US branch and other changes to the primary process

As mentioned, we created this alternate version to show there are other ways to build the model. You can find this alternate design in the supplemental material.

Optimization experiments

With version 3 of the model, we incorporated physical space into the model definition. At this point, we can evaluate and suggest improvements based the management's objectives. The management wants to see if they can improve (decrease) the average processing time of patients if they change the number of staff (doctors and technicians).

To use our simulation model as a decision-making tool, we need to run experiments. Running the simulation shows us just one realization (sample path) of the operation in a specific setting. In our current setup with three doctors and three technicians, the average processing time shows us what could happen in one hypothetical day of operation. Besides being only one possible outcome for that day's operation, the simulation is limited to showing us a possible output for that specific staffing scheme. To compare staffing schemes, we can run the simulation with different numbers of doctors and technicians. Solving this manually by setting up a scheme, running it enough times to obtain a statistically valid metric, repeating the same procedure for all alternative schemes, and then comparing the results to find the best would be tedious and time-consuming. AnyLogic has optimization capabilities that can automate this process.

Before we build our optimization experiment, we must parameterize our model based on the input values we have control over (which we'll use as decision-making levers). In our model, these parameters will be the number of doctors and number of technicians. To run a more comprehensive set of scenarios, we also parameterized arrival rate per hour and daily patient limit. However, these won't vary in our experiment since they aren't decision variables. For now, we'll keep their original values fixed (10 patients arrive per hour and 50 patients must to be processed in a day).

1. Open the version 3 of the clinic model (original version not the alternative). From the **File** menu, click **Save as…** to save the model with a different name and keep the original model intact.
2. Use the Agent palette to add four **Parameters** to Main. Their type should be int (integer) with initial values and labels shown in Figure 3-154. It isn't important where we place these parameters in **Main**, but placing it inside **viewArea_Process1** would be a good option.

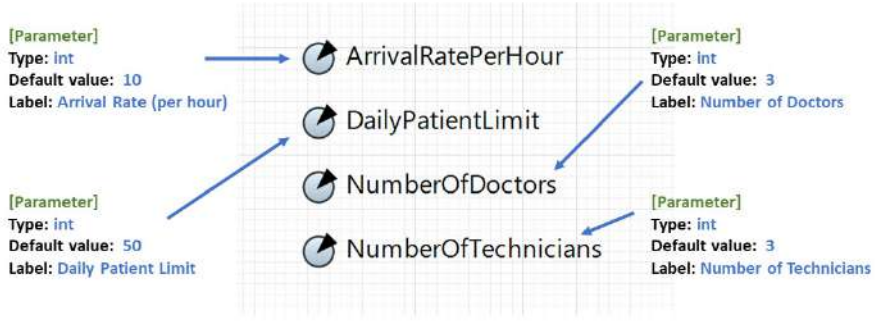


Figure 3-154: Adding four parameters to the Main

3. Substitute these new parameters with the hardcoded values inside the model as shown in Figure 3-155.

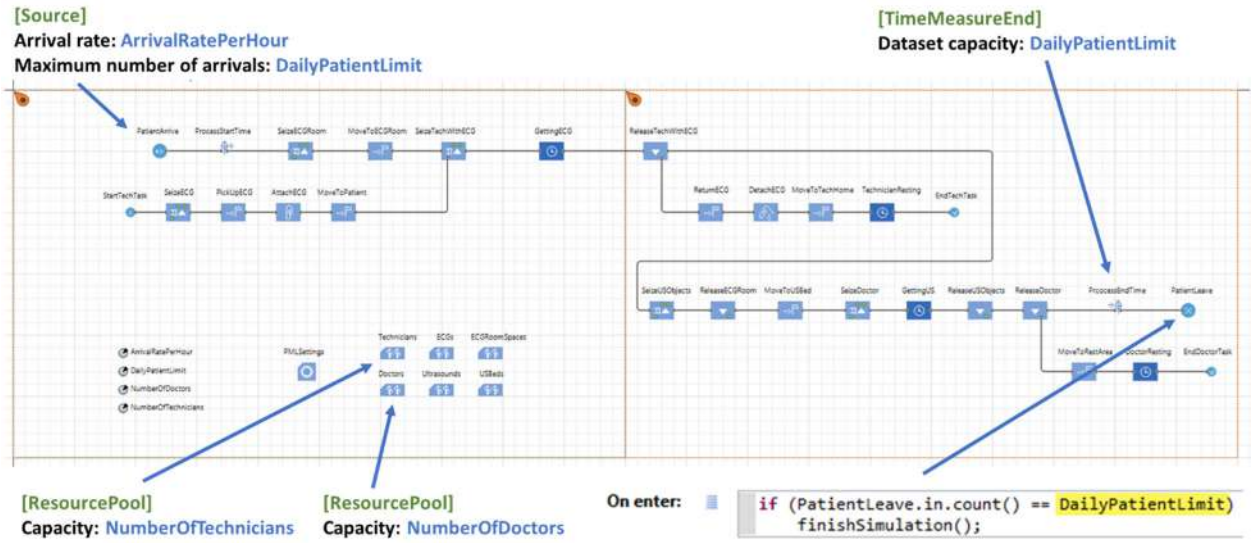


Figure 3-155: Incorporating the parameters into the process

4. With a parameterized model, we'll make one minor change: from the **Project** panel, right-click the **Simulation: Main** experiment and click **Rename…**. In the dialog box that appears, rename the experiment to "SimulationBaseline". This isn't necessary, but it's an easy way to quickly differentiate the original experiment from future experiments.

5. To build the optimization experiment, start by right-clicking the model name (in the **Projects** panel), then click **New** > **Experiment** (Figure 3-156).

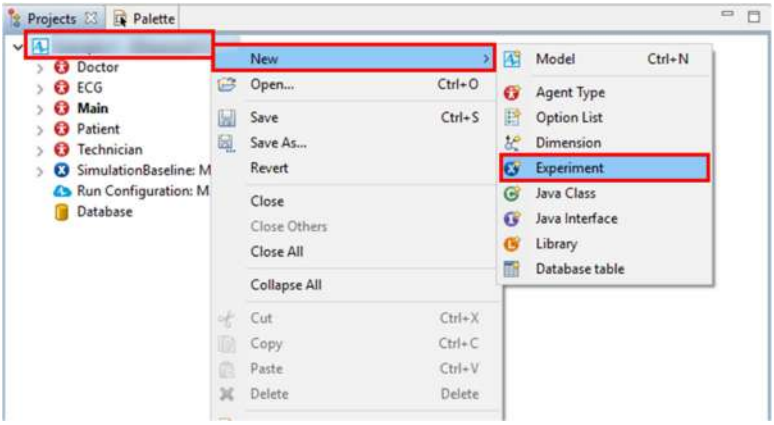


Figure 3-156: Getting to the Experiment wizard from the projects window

6. In the wizard, under the **Experiment Type** section, select **Optimization**. Name the experiment "OptimizationS1". As shown in with the default-checked box at the bottom of Figure 3-157, the Optimization experiment will copy its time settings from the simulation experiment.

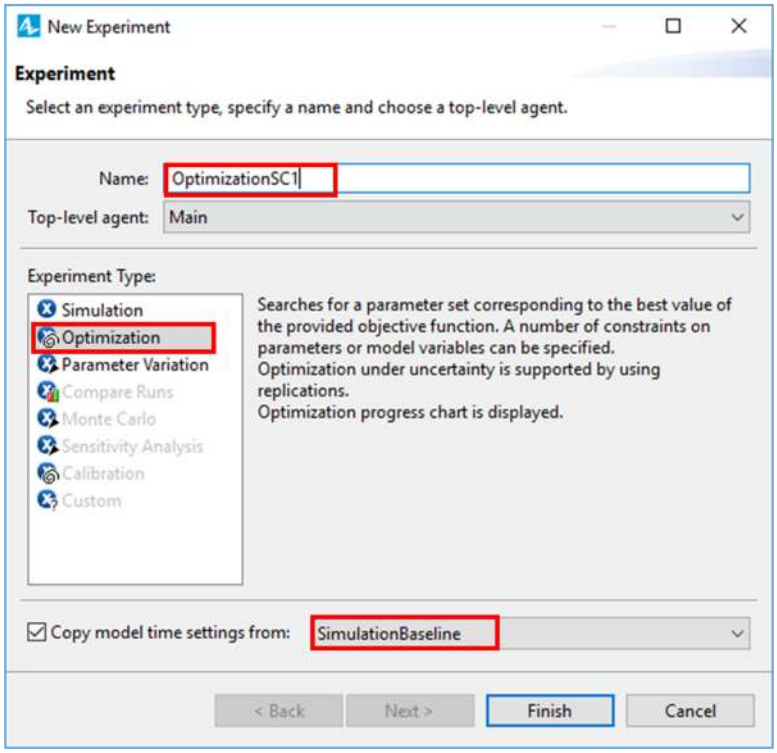


Figure 3-157: Setting up the new experiment

7. When you use the wizard to build a new experiment, AnyLogic automatically opens the experiment in the graphical editor. You can check the tab on top of the graphical editor to make sure the **Optimization** experiment is open. If you closed this tab, go to the **Projects** panel and double-click the experiment's name to open the experiment tab (as shown in Figure 3-158).

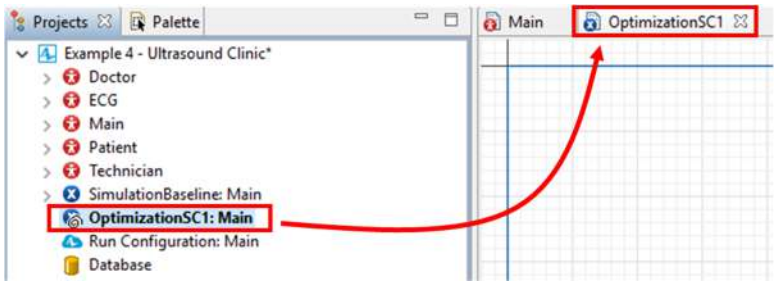


Figure 3-158: Opening the optimization experiment by double-clicking it in the Project panel

8. With the Optimization experiment as the active tab, do the following to modify its properties:

   a. The first change is to set the objective function, which is the target we want to achieve. This field's value determines whether the current scheme is considered "optimal". In our case, we want to minimize the average processing time (defined by the **AverageProcessTime** **Output** object in Main). The simulation engine will run the simulation and read the value of **AverageProcessTime** after the run. To set the objective function, select the radio button for **minimize** and set the field below it to: `root.AverageProcessTime`

Since the **AverageProcessTime** object is inside Main (our top-level agent), the prefix "root" is used as a keyword that is available in this field that points to **Main**.

b. To find the best set of parameters that minimizes the objective function, we must specify which parameters should be considered decision input parameters (that is, the levers to modify). AnyLogic automatically imports all the parameters from the top-level agent (**Main**) to the experiment. These parameters are modifiable from in the **Parameters** section of the experiment **Properties** window. In this example, **DailyPatientLimit** and **ArrivalRatePerHour** will have fixed values (50 and 10, respectively); only the **NumberOfDoctors** and **NumberOfTechnicians** parameters will be in consideration: set their type to int and a boundary for each parameter to be between 1 and 10 with a step of 1 (Figure 3-159).

c. After setting the input parameters and the objective function, click the **Create default UI** button. AnyLogic will automatically add informative representative elements to the experiment's UI.

d. Keep the **Number of iterations** at its default 500 iterations. This option specifies the number of unique combinations of input parameters that will be tested during the experiment. Since we only have two parameters, each with ten possible values each (1 to 10), the maximum possible number of iterations is 100. AnyLogic will only need to run 100 iterations to cover all possibilities; the 500 value should be thought as the upper limit of number of iterations rather than a mandatory number.

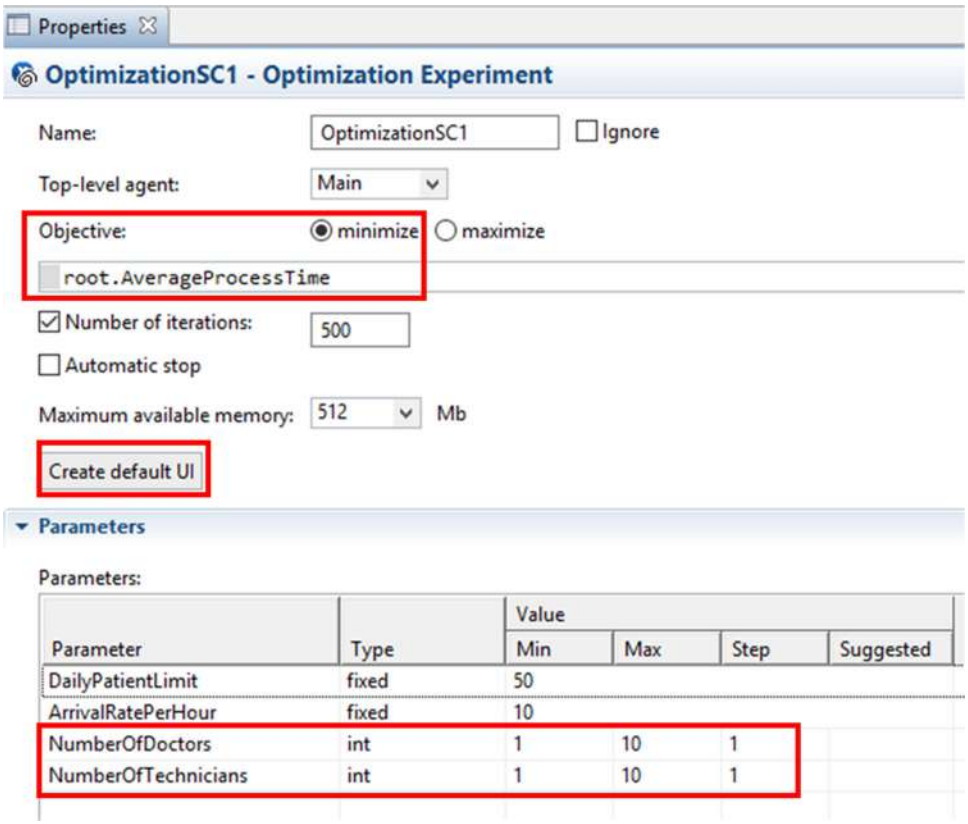


Figure 3-159: Setting the optimization objective function, input parameters, and creating the user interface

9. Two other settings to be changed from the **Requirements** and **Randomness** sections:

    1) On top of the objective function, you can further specify the desired configuration with constraints and requirements. Constraints in AnyLogic are similar constraint to ones found in in Linear Programming (LP) or Mixed Integer Linear Programming (MILP). Each time the optimization engine selects a new set of values for the optimization parameters, it checks these constraints. If the conditions aren't met, it moves to the next set of values. These constraints help shrink the search space and perform the optimization more quickly. While this optimization has no constraints, one such example would be if we didn't want more than 10 staff. Here, the constraint would be "`root.NumberOfDoctors + root.NumberOfTechnicians <= 10`".

       On the other hand, requirements add restrictions that AnyLogic checks after each simulation run. Based on the simulation outputs, the run will be considered infeasible (invalid) if the stated expressions aren't met. In this experiment, we must make sure the last patient departs before the eight-hour day ends. Since the given set of inputs don't tell us when the last patient will leave the clinic, we must complete the simulation before we can check this criterion.

       As shown in Figure 3-160, add a new requirement by clicking the plus-button under the **Requirements** setting and set the line to read:

       ```
       root.LastPatientDepartureTime <= 480.0
       ```

       As in previous steps, the keyword `root` is used to reference **Main** (our top-level agent). We set the value requirement to be less than or equal to 480; this value is in minutes, which we know since that's the time unit used in our model (accessible from the model's properties by clicking its name in the **Projects** panel) and is equivalent to 8 hours.

    2) It's also important to have a random seed, so each simulation run results in a unique and new outputs. This is done under the **Randomness** section by clicking the radio button for **Random seed**.

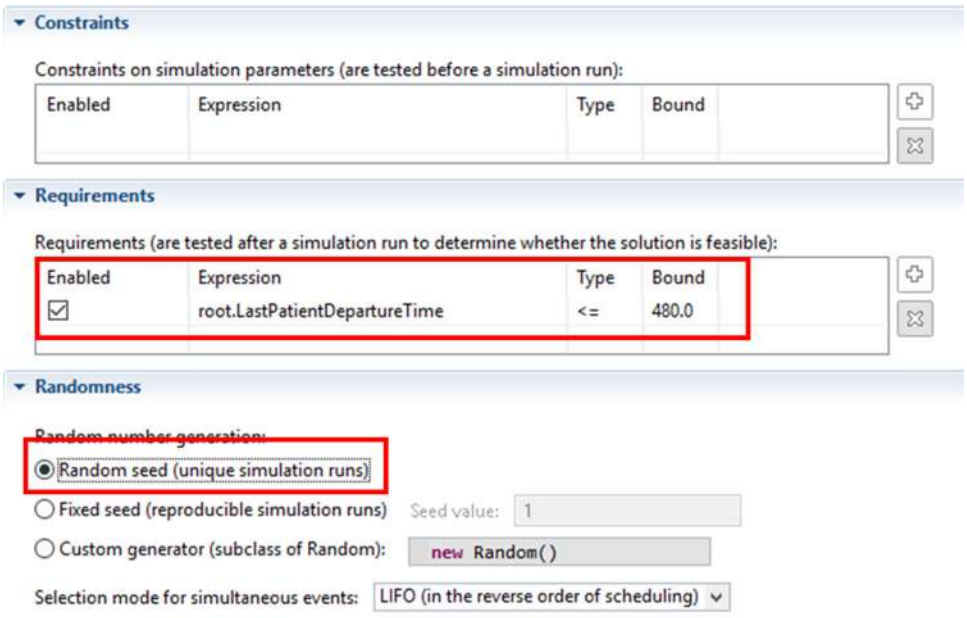


Figure 3-160: Adding a requirement and setting a random seed to the Optimization experiment

10. Run the **Optimization** experiment, the results of which you can see in Figure 3-161. You can also see several plots and outputs in the optimization experiment's UI, but the important ones are:

    a) The best value of the objective function. In our case, this is the lowest average processing time for the best combination of number of doctors and technicians.

    b) The values of the decision parameters that resulted in the best objective function. In our case, the 22.689 average process time is the result of having six doctors and ten technicians.

Figure 3-161: Output of first run of optimization experiment

While it may seem as though the optimum values for the parameter have been found, you can see running the same optimization experiments several times will result in different (supposedly) optimal values – the reported optimal values from six runs are shown in Table 3-20. It may not be clear how the optimum

processing time and number of staff is so different in each of run of the exact same optimization experiment. One of the main reasons behind the fluctuating outputs is the inherent randomness in the model. We knew that this model is a terminating simulation and the output of each run could be significantly be different (as we've mentioned in the step 9 of version 3). Even with the same exact inputs, the average processing time at the end of each working day might be very different. Therefore, there's a chance that for any given set of input parameters, the simulation's outputs are an outlier to the "true" values. Comparing a single sample path for each set of input parameters isn't the best way to differentiate sets of input parameters.

Table 3-20: Outputs of several optimization experiments with the same exact setting (without replication)

Optimization Experiment	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
Avg. processing time	23.11	23.08	22.52	23.29	22.63	22.70
# Doctors	7	7	9	7	9	10
# Technicians	6	7	7	7	10	9

11. As a remedy for the inconsistencies between single runs, we can run each iteration (a unique set of input parameters) several times and take the average outputs. These simulation runs that have the same set of input parameters are called "replications". For example, if we repeat the simulation with three doctors and three technicians for 10 times, you would refer to this as one iteration (because they have identical inputs) with 10 replications.

    To add replications to our experiment, go to the optimization experiment's **Properties** panel. Under the **Replications** section, check the **Use replications** checkbox. After that, a few new settings will appear below the checkbox; set the number of replications to 100, as shown in Figure 3-162. By doing this, we're assigning 100 replications to each iteration. Since we're going to run 100 iterations (all possibilities of number of doctors and technicians) and each iteration will run 100 times (100 replications), each experiment will have 10000 (100 * 100) simulation runs.

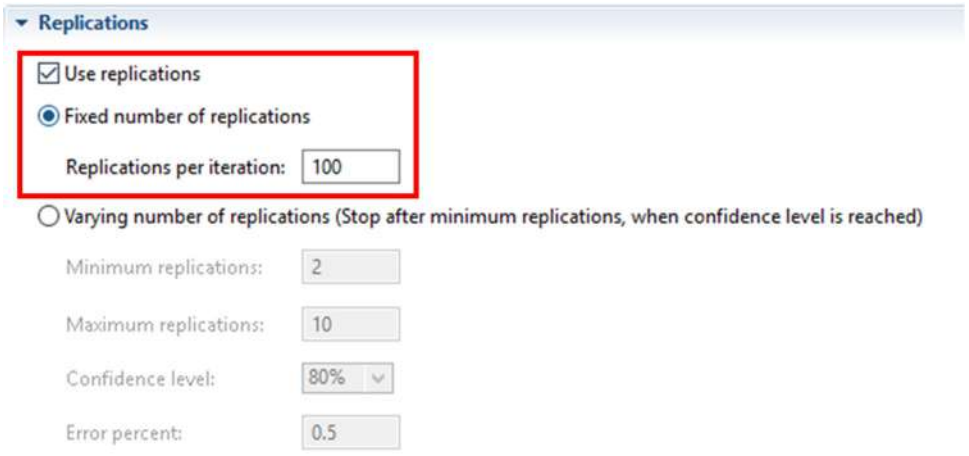


Figure 3-162: Setting the number of replications for the optimization experiment

12. Run the modified optimization experiment (now with the replications). The results of another six sample runs are shown in Table 3-21 below. From this, you'll see the average processing times in

these experiments is higher than optimization experiments without replications (Table 3-20) and the optimum number of doctors and technicians continue to fluctuate between runs.

The average processing time is should be considered roughly the same, since the difference is due to the inherent randomness of the model. Since the average processing times in the experiment are from several replications, they're closer to the expected values and are less likely to be from an outlier sample. Therefore, we have more confidence in these results compared to the ones from just one sample in Table 3-20.

For a terminating model that is still in a transient state, if we try to find the optimum set of parameters by running each set of input parameters run just once, we may end up in choosing the first outlier that gives us the minimum average processing time. However, when we run each iteration several times (each set of input parameters are run 100 times) and we compare their averages, we're working with more stable and representative values for the objective function. Therefore, the expected value of the minimum average processing time is more likely to be close to approximately 27 minutes (values from Table 3-21) rather than approximately 23 (values from Table 3-20).

It appears that counter to the improved accuracy in the average processing time, the reported optimum number of staff *appears* to have not benefitted in accuracy from the replications, which isn't true. In reality, this shows we have many different viable sets of input parameters, and all of these alternative sets will minimize the average processing time. For example, having eight doctors and nine technicians leads to an average processing time of 27.152 minutes (Run 2 in Table 3-21). This is much like having nine doctors and eight technicians with an average processing time of 27.05 minutes (Run 6 in Table 3-21). Since both input parameters sets result in a similar objective function value, it's up to us (or the clinic's management) to choose a set based on other preferences.

Table 3-21: Outputs of several optimization experiments with the same exact setting (with 100 replications)

Optimization Experiment	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
Avg. processing time	27.15	27.152	26.86	27.06	26.84	27.05
# Doctors	9	8	7	10	7	9
# Technicians	9	9	4	4	7	8

One interesting result in Table 0-4: Run 3 has the lowest number of staff compared to the other runs, but it still performed similarly to other optimization experiment runs. It may not be clear why the run with the lowest number of staff isn't consistently the optimum result. If you review the objective function and requirement we set for the staffs is desirable. Our objective was only to minimize the average processing time by selecting however many doctors and technicians (between 1 and 10) achieved this (while meeting our requirement that the last patient leave before the end of eight hours).

The optimization engine determined nine doctors and nine technicians that process patients in an average 27.15 minutes (Run 1) are as good as seven doctors and four technicians that process patients in an average 26.86 minutes (Run 3) because these input parameter sets minimize the objective function and satisfy the requirement. Now, we'll see how we can refine these requirements to produce a better set of values.

13. To define a more meaningful set of requirements for our optimization experiment, we first want to build a second experiment with similar settings. To do start, rename the current experiment to **OptimizationSC1**. Build a new Optimization experiment by going back and following steps 7 and 8 (making sure to name this experiment **OptimizationSC2** and ensure it uses a random seed with the same requirement as the first).

14. For this new optimization experiment, we'll add two more requirements (as shown in Figure 3-163):

    ```
    root.Doctors.utilization() >= 0.7
    root.Technicians.utilization() >= 0.7
    ```

    These two requirements read the utilization of the Doctors and Technicians resource pools from the Main (root points to the top-level agent which is set to Main) and only accepts the result as feasible if both utilizations are above 0.7 (or 70%). With these requirements, we force the optimization engine to try to reduce the number of doctors and technicians. Overstaffing will result in a lower utilization across each staff member.

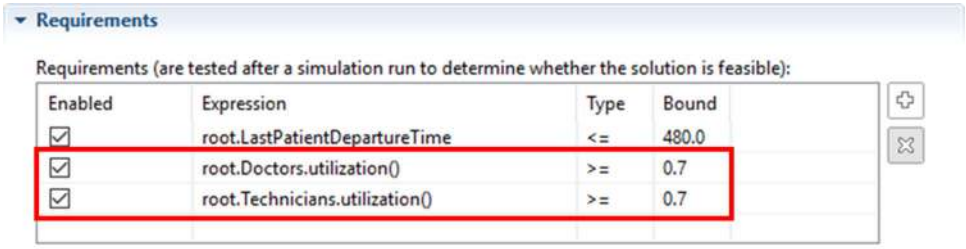


Figure 3-163: Adding two new requirements to the second Optimization experiment

15. Run the second optimization experiment with the extra requirement. The results of another six sample runs are shown in (Table 3-22).

Table 3-22: Outputs of the second optimization experiments (with 100 replications)

Optimization Experiment	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
Avg. processing time	51.816	51.535	56.331	52.445	47.701	48.197
# Doctors	5	5	4	5	5	5
# Technicians	2	2	2	2	2	2

Since we added two extra requirements, this placed extra conditions on what was considered to be an acceptable set of parameter sets. This resulted in the (minimized) average processing time to be increased in this scenario. Compared to the results of the first optimization experiment (Table 3-21), the set of parameter values are much more consistent between runs. By adding requirements, we reduced the range of acceptable input parameters.

16. To test the one of the optimum values found by the second optimization experiment (Run 1 with five doctors and two technicians), we'll make a copy of the original simulation experiment (**SimulationBaseline**) to separate it from the first. In the **Projects** panel, right-click the original

simulation experiment, click **Copy**, right-click the experiment name, and click **Paste** (Figure 3-164). Rename this experiment to "ScimulationSC2" and make sure it's set to run with a random seed.

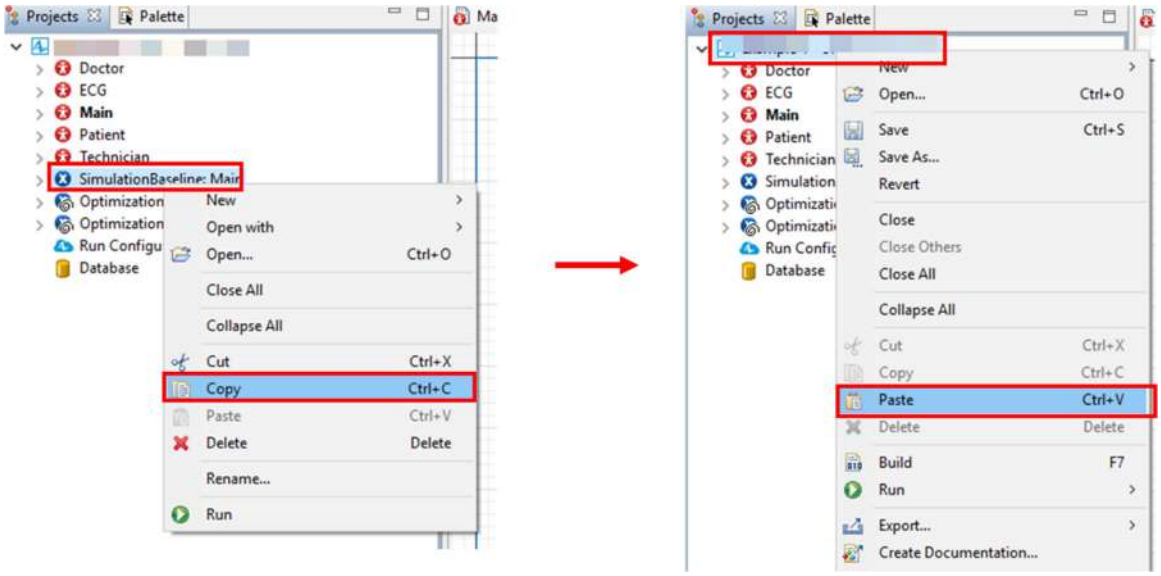


Figure 3-164: Make a copy of the original simulation experiment to run it with the optimum values

17. Open the new simulation experiment and set its number of doctors to 5 and technicians to 2. Then run the experiment to find the results.

As you can see from the sample runs shown in Table 3-23, each run has the optimized input parameters. However, it won't fully match the expected values from the optimization experiment. It also won't fully satisfy the requirements set in optimization experiment. However, we expected this: the clinic's daily operation has many random elements, which together cause our results to vary. A review of the results show they aren't far from the requirements and improved the average processing time significantly.

Table 3-23: Result of five simulation runs with the optimized input parameters (5 doctors, 2 technicians)

Simulation Experiment	Run 1	Run 2	Run 3	Run 4	Run 5
Avg. processing time	43.6	36.6	48.1	37.42	46.43
Doctors utilization	0.72	0.69	0.74	0.66	0.69
Technicians utilization	0.91	0.86	0.93	0.88	0.9
Last patient departure time	371.63	387.83	365.97	390.92	377.91
**The values shown here are outputs of one simulation run compared, values in Table 3-20, Table 3-21, and Table 3-22 were outputs of optimization experiments.*					

It's important to mention that the simulation experiment shown in this example was only based on the crude outputs from the OptQuest optimization engine. There are more advanced methods for selecting an optimal subset from a finite set of simulated scenarios that are outside the scope of this book.

Further questions and experiments:

It would be a good learning exercise to try to answer the following questions.

1. In a clinic, it's highly likely the arrival rate (for example, 10 per hour) may fluctuate when patients cancel their appointments. How could you incorporate this behavior into your model?

    - You could add a **SelectOutput** block after the source to remove a percentage of patients (the percentage of patients who won't show up) from the main process.

2. Assuming the clinic's management wants to use the clinic to its fullest capacity, how could you find the maximum number of people that can be processed in the eight hours of operation?

    - One solution would be creating an optimization experiment to maximize the number of patients that are processed (objective function set to maximize the value: `PatientLeave.count()`), making sure to have a requirement that the last patient departure time is no later than 480 minutes (8 hours).

3. Try creating a new optimization experiment with the number of doctors and technicians to be fixed (at the values found in one of the optimization experiments), and the hourly arrival rate and daily limit as variable parameters. Do the "optimized" values match up with what you saw in the other optimization experiment?

Appendix A: AnyLogic Development Environment

In this section, all the core and necessary elements of AnyLogic development environment are introduced. This introduction is enough to get you started on using the software. Obviously, there are more detailed and advance attributes that we'll cover in Level 2 and 3.

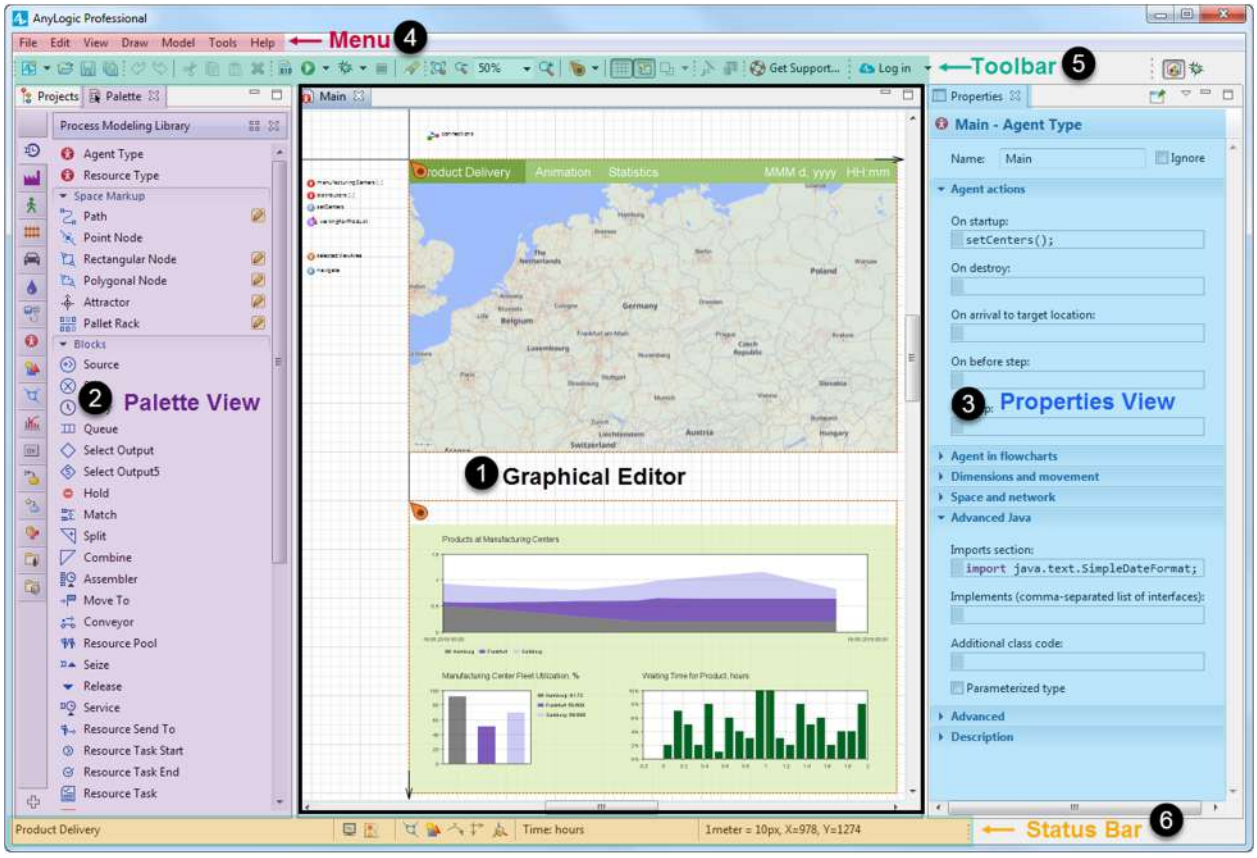


Figure A-1: AnyLogic user interface

Each of the following parts of the AnyLogic user interface are labeled in Figure A-1 above and Figure A-2 below:

1. **Graphical Editor**: This is the empty graphical editor that you build your models in. It is called *graphical* editor because everything that you add to this initially empty graphical editor is by dragging model elements from the *Palette View* and dropping them here. If you have not started a new model yet, this area might be empty.

2. **Palette View**: This is where all types of model building blocks are categorized into meaningful groups we call **palettes**. Your models are built from different pieces, and all those elements are taken from the Palette view. You just need to know what you want and in which palette they are.

3. **Properties View**: This area is contextual, meaning that it only shows the properties of whatever element you've currently selected in the graphical editor. Based on the currently selected element, you'll see different properties available to you for customization.

4. **Menu**: This area contains several menus, that let you access a comprehensive set of commands. "File", "View", and "Help" are the more frequently used menus that we'll explain in this section.

5. **Toolbar**: This area gives you quick access to frequently used commands. All these commands are also accessible from the menus.

6. **Status Bar**: Showcases some useful information about the model you are working on, like the name of the model and model time unit. The status bar also has two buttons that give you access to the *Console* and *Problems* views.

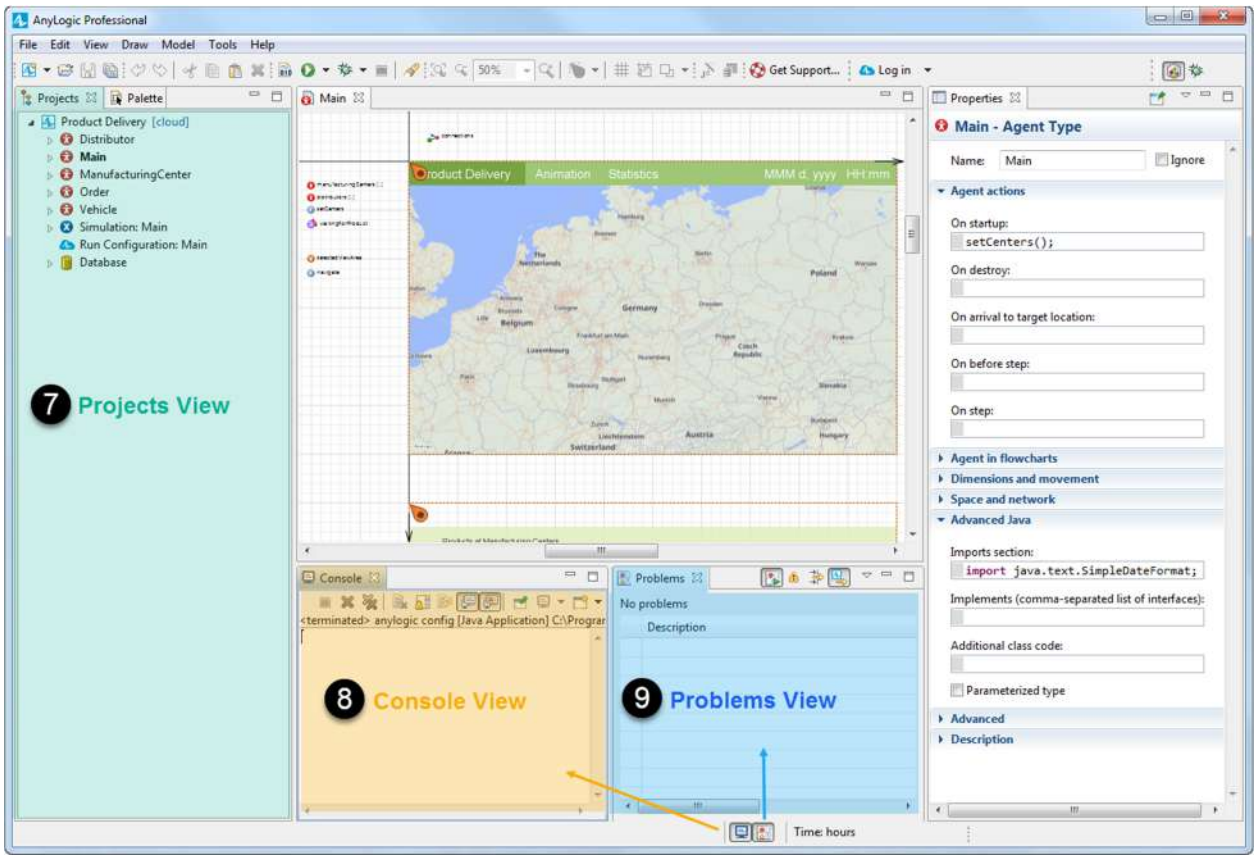


Figure A-2: AnyLogic User Interface (continued)

7. **Project View**: This area provides access to the currently opened models and their elements. You can see elements of a model in tree format and easily access any element in the tree. The projects view also shows all the experiments you've built for a model. Configuration settings for uploading model to AnyLogic Cloud and built-in database tables are also accessible from this view.

8. **Console View**: This is an interface to a running model. One of its uses is to output some useful information about what is going on during the model execution. If your model encounters some errors during the model execution (runtime errors), AnyLogic prints some useful information about the problem along with links to their location in this view.

9. **Problems View**: This window shows compile time errors. Compile time errors are errors that can be detected by the compiler before running the model. The most common type of compile errors are syntax errors related to not following the mandatory Java syntaxes. To evaluate and see if the models has any compile time errors, you must *build* the model first.

Appendix B: Basics Model Building Techniques

In this appendix, we'll review some of the basic model building techniques that will help you comfortably build models in AnyLogic.

Zooming and panning in the development

To pan (move around) inside the graphical editor, you first need to make sure the desired environment is opened. Taking Figure B-1 below for example, to open **Main**, you first need to find the entry for it under the current model's tree in the **Projects** view. By double clicking on it, its graphical editor will open in a tab with in the graphical editor. The active opened tab will be in a bold font in the **Projects** view.

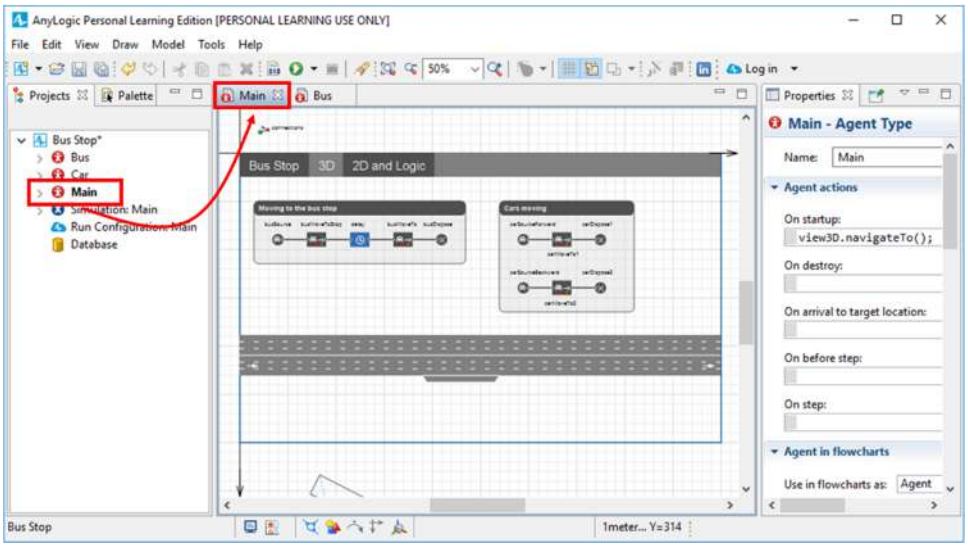


Figure B-1: Opening a graphical editor from Projects view

Panning lets you move around in the graphical editor. Since the graphical editor is technically an infinite space, there's no difference between different parts of this space and you can put your model objects anywhere you want. The only special area in this infinite graphical editor is a default blue frame inside it that specifies the area that will be initially visible when you run your model. Beside the fact that you cannot move or delete this frame (though you can resize it), it has no other special attributes. Parts of your model outside the blue frame are still included in the model but are only not initially visible at run time.

To pan around in the selected graphical editor, you need to right click and drag around. To zoom in and out in a graphical editor, you need to hold Ctrl (Windows) or Cmd (Mac) and use your scroll wheel.

The combination of panning and zooming lets you to move around and focus on a specific region of your model throughout the model building process.

Zooming and panning in in model window (runtime)

While running a model, AnyLogic will only show what was in the blue frame (Figure B-2). You may not see all the desired elements of the model. Like the graphical editor, you may also need to pan and zoom around to see other elements with a better focus.

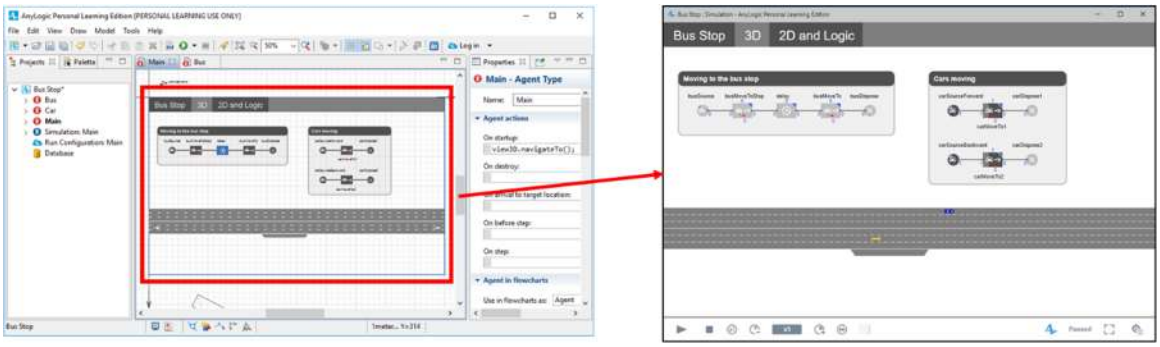


Figure B-2: AnyLogic shows all the elements that inside the blue frame at development environment (left) in the model window at run time (right)

To pan around in the model window during runtime, the action is very different based on your operating system. Windows users can simply drag the window (in contrast with the development environment, you don't need to right-click). Mac users must hold Cmd while dragging.

To zoom in and out of the model window during runtime, the behavior is similar to the development environment: Windows users hold Ctrl and use the scroll wheel, Mac users hold Cmd and use the scroll wheel.

The combination of panning and zooming lets you move around and focus on a specific region of a running model, which may be inside or outside the default view. If the default blue frame is not enough, or if you want to separate the model into several views, AnyLogic also lets you build multiple customized areas.

Adding blocks to the graphical interface

The building blocks of AnyLogic models are categorized under the **Palette** view. To add a new block to the graphical editor, you first need to select a palette from the **Palette** view. Then you can drag-and-drop any element from that palette to the graphical editor (Figure B-3).

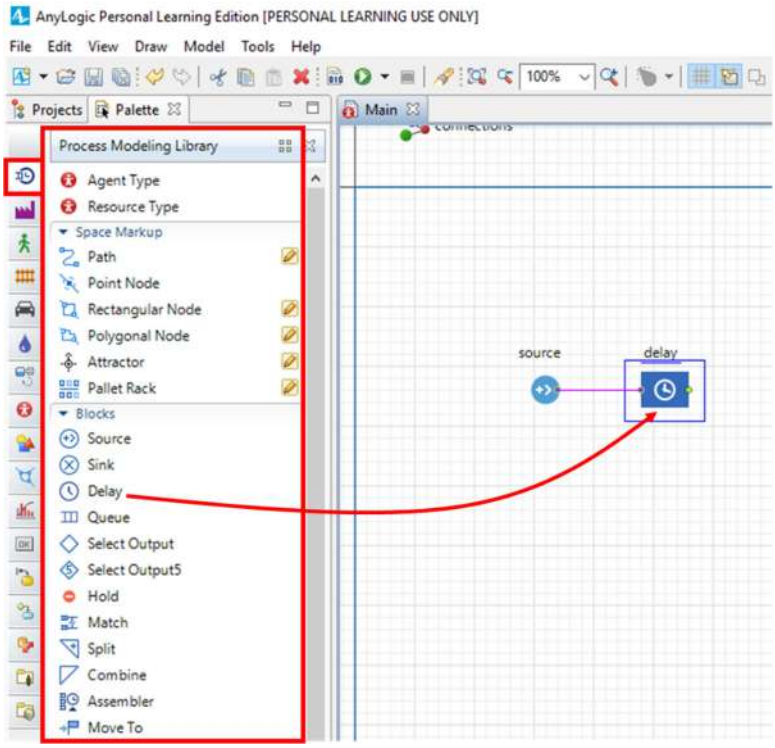


Figure B-3: Drag-and-dropping a block from Process Modeling Library (PML) into the graphical editor

Elements of some of the palettes are separated into several section. The Process Modeling Library (PML) has two main sections: **Space Markup** and **Blocks**. Space markup elements are used to draw a physical environment that will become part of your model definition. For more easy customizations, some of these elements have a secondary way of being added to the graphical editor called drawing mode. If you look closely, you'll see that some elements have a pencil icon to the right of their name. If you double-click on the elements with this icon, you'll be able to draw the element instead of just a drag-and-drop operation. Depending on the drawing element that you chose in the space markup section, you'll have appropriate options to easily draw that element. For example, if you are drawing a rectangular node, your first left-click specifies one corner of the rectangle; while you keep the left-click held down, you can drag the cursor to specify the other corner of the rectangle (Figure B-4).

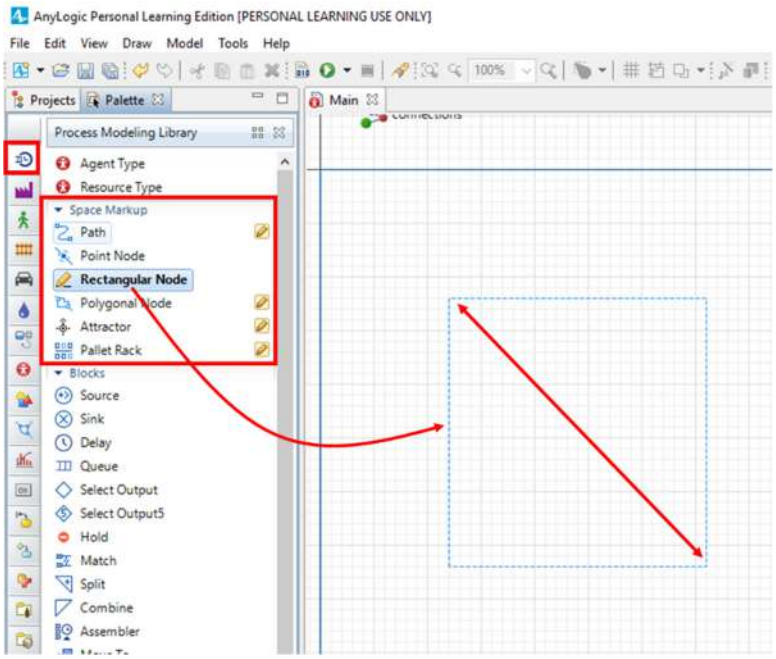


Figure B-4: Drawing a space markup element in the drawing mode

Selecting, moving, and deleting blocks to the graphical interface

To select a block or a space markup in the graphical interface you just need to click it. To select more than one element, you should drag the mouse around the area containing the elements you want to select (Figure B-5). You can press Ctrl + A (Mac: Cmd + A) to select all the elements inside the graphical editor.

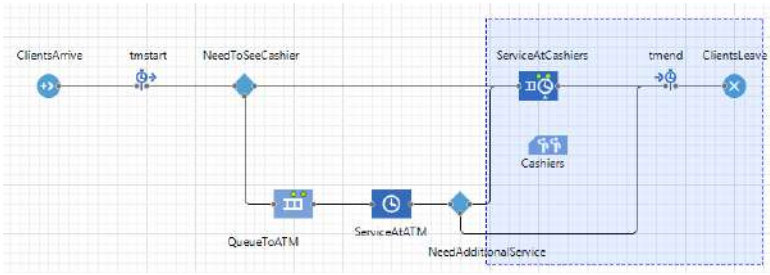


Figure B-5: Selecting several elements at once by dragging an area around them

To move one or more selected blocks, click one of the elements and drag them to the desired location (Figure B-6). As an alternative, you can move the selected elements using the arrow keys.

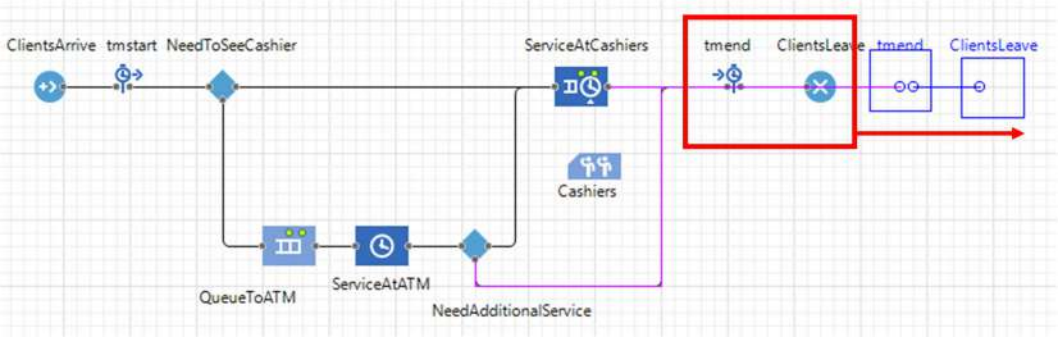


Figure B-6: Selecting a couple of blocks and moving them to the right by dragging the mouse

To delete one or more selected blocks, right-click one of the selected elements and select **Delete**. Alternatively, you can delete the selected element(s) by pressing the **Delete** key on your keyboard (Figure B-7)

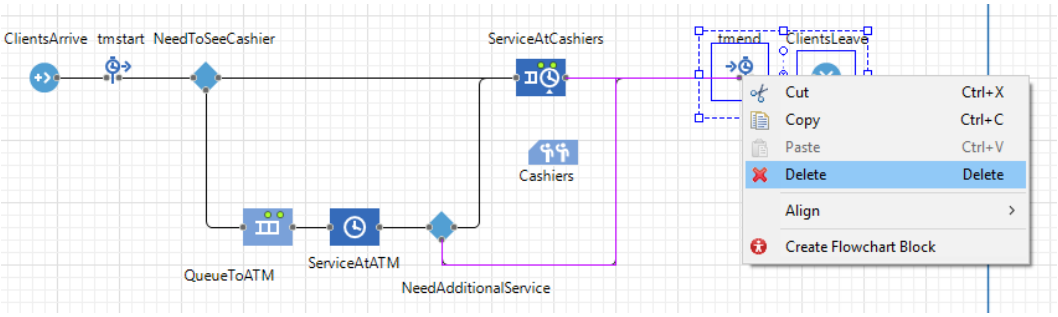


Figure B-7: deleting a selected element with right-click > Delete

Connecting ports with connections

Models built with a sequence of connected blocks (e.g., blocks from the PML) use a **Connector** element, which can be found in the **Agent** palette. The connector's function is to make a logical connection between two ports that lets entities move from one block to another. However, when you drag and drop PML blocks into the graphical editor, AnyLogic automatically connects the two closest ports. If you want to manually draw a connection, double-click the first port to start a connection, then click the other port. For this to properly work, you need to have zoomed-in enough for the ports are clearly visible, otherwise it may not properly connect the ports.

When two ports are correctly connected, you'll see a small green colored dot inside the port. Otherwise hanging (unconnected) connectors will be highlighted in red (Figure B-8).

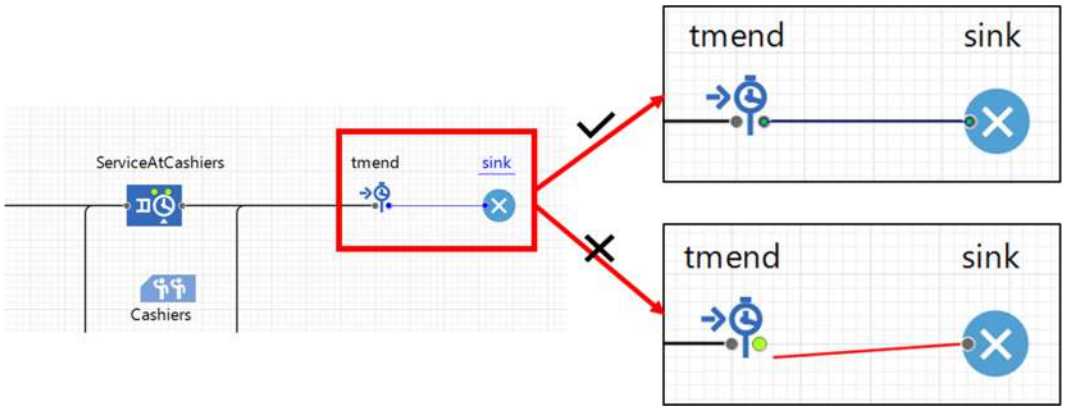


Figure B-8: Connecting the ports between two blocks (correctly connected vs hanging connections)

You can deactivate automated connections by disabling the option located at: **Tools** > **Preferences** > **Graphical Editor** > **Connect automatically, when ports are close enough** checkbox.

To delete a connection, select it (by clicking) and then right-click > **Delete** or pressing the **Delete** key on your keyboard.

In AnyLogic, whenever you have a linear shaped object (e.g., connections between ports, transition between states, paths between nodes) you can add extra sections to draw it with more flexibility. See Figure B-9 for an example, where the middle of a connection was double-clicked and then the newly added point was dragged to change the shape of the connection.

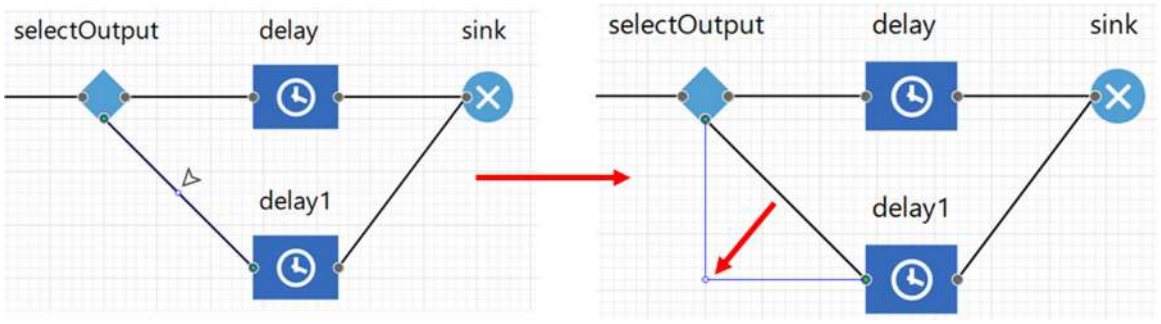


Figure B-9: Adding a new point to a connection and move it to change the shape of the connection

Adding sections and changing the shape or length of a connection does not have any effect on its functionality.

Running a model

To run a Simulation experiment (or any other type of experiment), you have several options:

1. In the **Projects** view, find the experiment that you want to run. Then right-click the experiment (e.g. **Simulation**) and choose **Run** from the popup menu (Figure B-10).

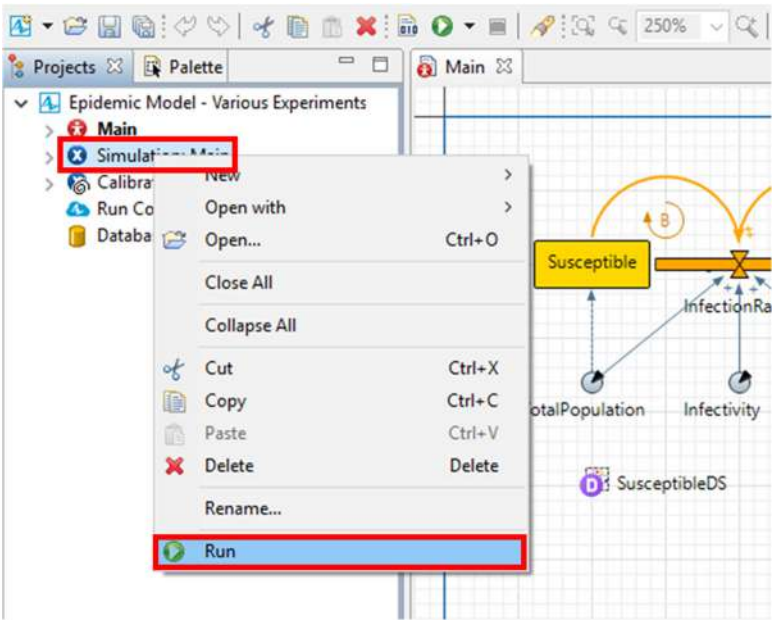


Figure B-10: Running the model from the Projects view

2. You can click the arrow to the right of the **Run** toolbar button and choose the experiment you want to run from the drop-down list (Figure B-11).

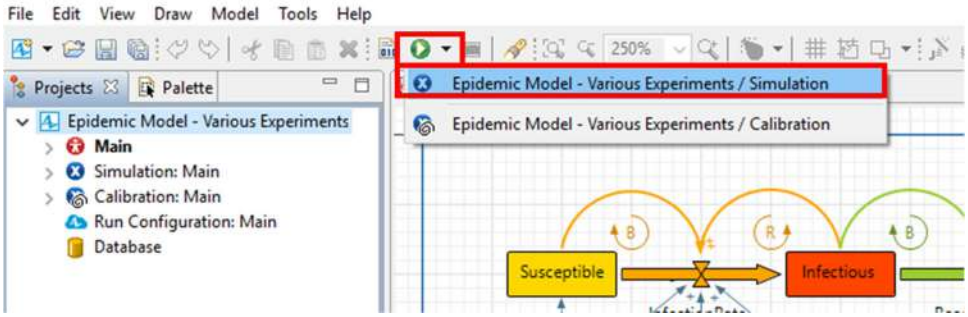


Figure B-11: Running the model from the toolbar button

If you only have one experiment, or you want to run the last run experiment, you can just click the **Run** button of the toolbar or press F5 on your keyboard.

Running an experiment does not start the simulation automatically. Instead, it takes you to the model's setup window that shows you the experiment's page. To start running the experiment you must either click the play button in the control panel of the model window or press space (Figure B-12).

Figure B-12: Run button in the control panel of the model window (Simulation experiment)

Model window, control panel and developer panel

When you run an experiment, the model is automatically displayed in a newly opened window. The control panel at the bottom of the model window lets you control the execution of the model, gives you dynamic information about the status of the model, and lets you get access to the developer panel.

Figure B-13: Different parts of the control panel

The different parts of the control panel (Figure B-13) include:

1) Play/Pause: Starts the model execution if not playing or pauses the model if it's currently running.

2) Stop: Terminates the execution and destroys the simulation model (does not close the window).

3) Slow down: Decreases the ratio of model time to real time.

4) Model time to real time ratio: Displays the current ratio of model time to real time (1 seconds). This scale is only active when the execution mode of a Simulation experiment is set to real time. This setting means that you want the simulation model to try to imitate the passage of time by mapping of model time (the artificial virtual time) to the real time. For example, scale of 5X means that the simulation tries to map 5 units of model time to 1 second of real time.

5) Speed up: Increases the ratio of model time to real time.

6) Virtual time mode: Run the model as fast as possible. You can switch between the real- and virtual-time mode by pressing this button.

7) Shows the simulation status:

- Loading: model data is being loaded.
- Running: model is running.

- Please wait: engine is executing a non-interruptible command (e.g. pausing the model).
- Paused: model is paused.
- Idle: model is either stopped or hasn't been started yet.
- Finished: model execution has been completed.
- Error!: runtime error occurred.

8) Opens the developer panel, which provides access to the additional model controls, simulation data, and console with experiment's output (Figure B-14).

Figure B-14: Different parts of the Develop panel

a) Model time indicators: Displays the current model time or date

b) Execution controls: Similar to the control panel, you can run, pause or stop the model. However, if you expand this section, by clicking on the setting show/hide options icon (three vertical dots), it gives you two more options (Figure B-15):

   - Run until: Sets the time or date at which model simulation will be paused (absolute time in future).
   - Run for: Sets the duration of time that the model should run starting from now.

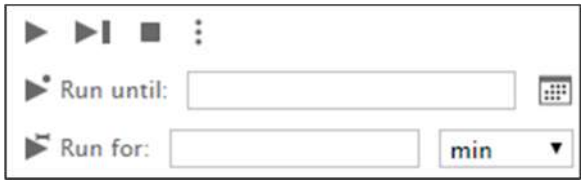


Figure B-15: Expanded execution control window

c) Model navigation: Lets you to navigate to the experiment start page, navigates to the top-level agent of the model (e.g. Main).
d) Agent navigation: Lets you navigate inside and between different agents within the model.
e) Console: shows the output of model that is written to the console with the help of `traceln()` or `trace()` functions.
f) Status bar: displays the low-level information about the model execution including the events-per-second (EPS), frames-per-second (FPS), steps that model has run, how long the model has run for, and the percent of memory currently used by the model (of the specified total amount).

Bringing back a view or the default perspective (development environment layout)

If you closed a specific view panel, you can bring it back by clicking on the **View** option in the top menu and select the chosen view from the available options (Figure B-16).

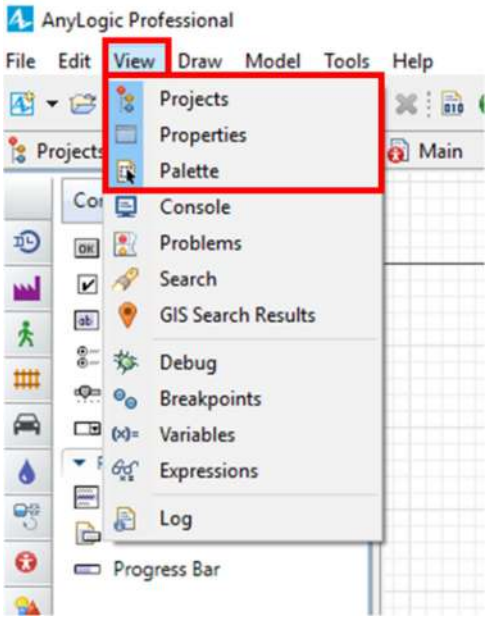


Figure B-16: Activating specific views from the top menu

AnyLogic development environment has perspectives. A perspective defines the set and layout of views in the AnyLogic workspace. If you moved several view panels (e.g., **Properties** from the right panel to the bottom) you may find it hard to put it back them all at their original location. To solve this problem, AnyLogic lets you to reset the perspective to its original layout. To do so, click **Tools** from the top menu and then select the **Reset Perspective**.

Appendix C: Model Building Blocks

Model Building Blocks - Level 1

Source

Source blocks are where the entities enter the process. The **Source** block's job is to build and inject new entities into the process. Therefore, the **Source** is like a magical machine that gives birth to new entities and then injects them into the process.

The most important attribute of the **Source** block is the way it handles arrivals of new entities into the model. Under the properties window you can access the arrivals setting, as shown in Figure C-1 and Figure C-2.

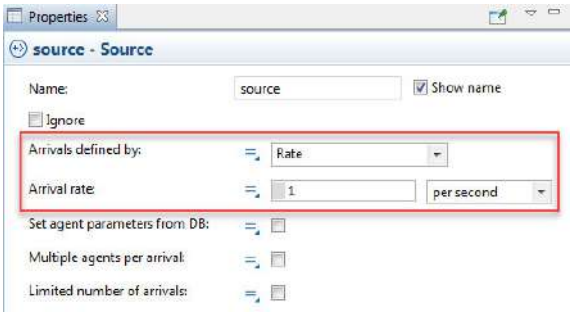


Figure C-1: Arrivals defined by Rate

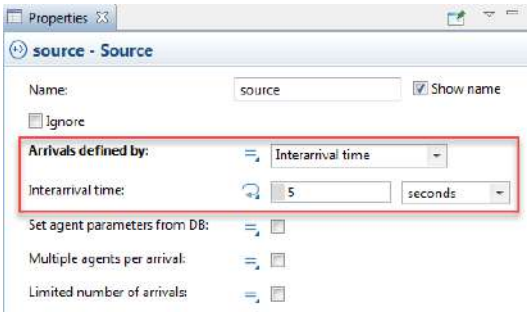


Figure C-2: Arrivals defined by Interarrival time

You can explore different options in **Arrivals defined by** dropdown menu. We'll go over the first two options, **Rate** and **Interarrival time**:

- **Rate**: entities are generated at an average of the specified value. This equates to arriving with an exponentially-distributed interarrival time whose mean value is $\frac{1}{rate}$). In essence, the rate is the expected number of entities that will enter the model over a specific period of time (e.g., 1 per second). You should keep in mind that the rate shows you the *expected* value. Because it follows a Poisson process, is not deterministic (refer to **POISSON PROCESS**).
- **Interarrival time**: the time between two subsequent entities is defined by a fixed value (e.g., 5 seconds) or a distribution (e.g., `uniform(1,5)` seconds). When fixed values are used, the injection of entities into the process is deterministic. If we use an exponential distribution (with the shape parameter x) for interarrival time, the arrivals will be equivalent to a rate of x. Using any distribution causes interarrival times to be stochastic.

Functions

`void inject(int n)` : Generates the specified number of agents (n) agents at the time of call.

Sink

The sink blocks dispose of entities. When an entity reaches the **Sink** block, we're no longer following the entity (that is, the end of the process). In the real world, the entity may still exist after the sink, but since it is outside the scope of our model, we discard and remove the entity from our model.

Functions

`long count()` : returns the number of entities that have exited from the **Sink** object so far.

Queue

A queue is a place where entities will wait if something stops them from moving forward in the process. If there's enough space in blocks after the queue, entities will pass through it in zero time. This means that if you look at the queue at any point in time, nothing will be inside it. The only reason for entities to accumulate in a queue is because something in the downstream blocks stop them from moving forward. You must develop the intuition that a queue on its own does not accumulate anything and only works as a transitional block to the next - it only starts to accumulate entities when other blocks in the downstream cannot move the entities forward fast enough.

Capacity of the Queue

The most important attribute of a queue is its **Capacity** that specifies the maximum number of entities that could be accumulated inside that queue at any point in time (Figure C-3).

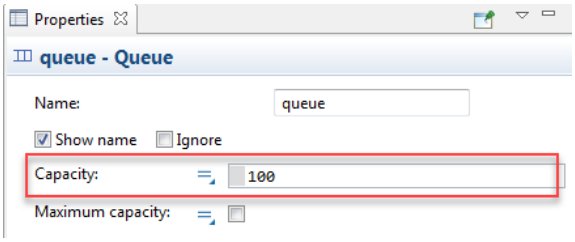


Figure C-3: Queue capacity limited to 100

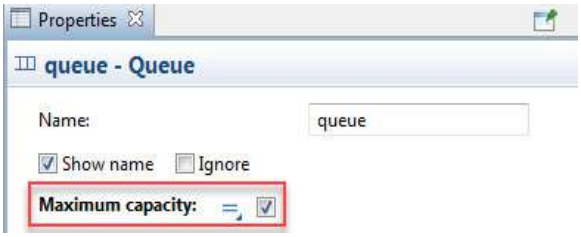


Figure C-4: Queue capacity set to infinite

In case the queue does not have any capacity limitation, you can check the **Maximum capacity** check box to effectively set its capacity to infinite (Figure C-4).

Queueing policy of a Queue block

By default, the queuing policy is set to **FIFO** (First-In-First-Out). This policy is the natural way things move in a queue: entities leave the queue in the same chronological order in which they entered (Figure C-5).

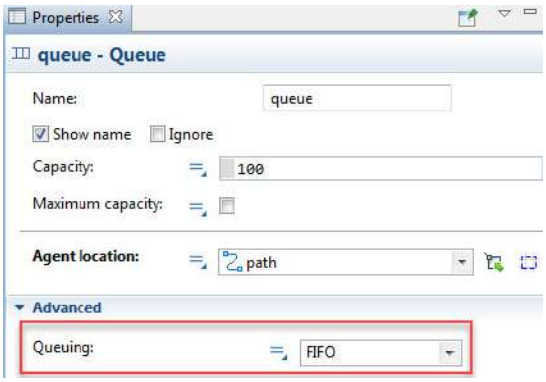


Figure C-5: Queueing policy of the Queue block

Delay

Delay is one of the most important building blocks of a process, probably the most fundamental block in the abstraction of reality to a process model. The **Delay** block impedes the progress of entities in the process for a given amount of time. The time delayed for represents the period the entities need to spend in a specific place before they can proceed (e.g., the time that a customer spends at an ATM machine). Two main attributes of a **Delay** block we'll to discuss first are **Delay time** and **Capacity**.

Assigning delay time

The value sets the amount of time the entity needs to stay at the delay block before it can leave it. It could be a constant (e.g., 5 seconds) or drawn from a distribution (e.g., `uniform (5, 10)` seconds). Having a constant value (Figure C-6) means that all the entities passing through the delay must spend the exact same time in the delay. In contrast, drawing from a distribution (Figure C-7) results in a different delay time for each entity that enters the delay. Since the delay time comes from a distribution, the difference between delay times will be limited to the boundaries and shape of that distribution (e.g., for a value set to `uniform(5, 10)` seconds, the delay time of an entity could be 5, 6.4, 7, or 9, but it cannot be 2 or 12 seconds). We'll discuss drawing values from distribution in more details in APPENDIX.

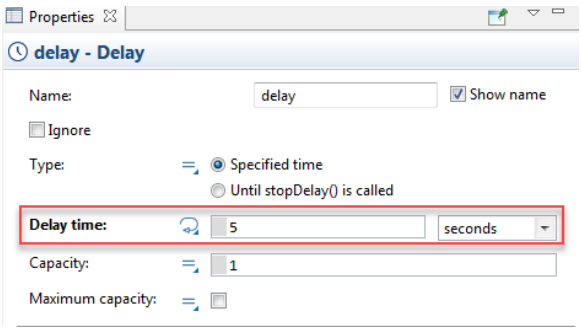


Figure C-6: Delay time - Deterministic

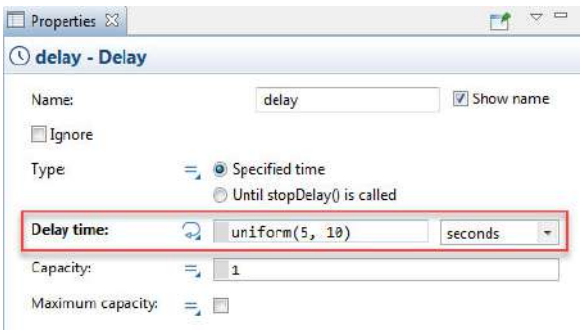


Figure C-7: Delay time - stochastic (from distribution)

Capacity of a Delay block

Capacity in a **Delay** block has a different meaning compared with the **Queue** block. Since the **Queue** and **Delay** blocks both have capacity as one of their main settings, it is useful to first clarify their differences before explaining the meaning of capacity in a **Delay** block specifically.

As previously stated, **Queue** is just a place to accumulate entities somehow blocked from moving forward in the process. The capacity of a **Queue** is the available space for accumulating entities at any moment in time. In contrast, entities that are staying in a delay are doing so because the **Delay** block itself stops them from moving forward. A **Delay** block's capacity relates to the number of agents that can be delayed concurrently and independently (Figure C-8).

For example, if you imagine a line of customers in front of a check-out counter at a shop, capacity of the queue limits the number of customers that can stand in the line. The capacity of the delay is the number of tellers, where more tellers result in more customers able to be helped simultaneously). In most cases we model the tellers more explicitly as resources which will be explained in later sections.

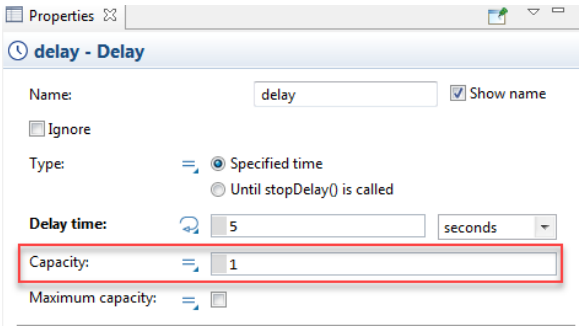


Figure C-8: Delay with only one entity able to be stopped at a time

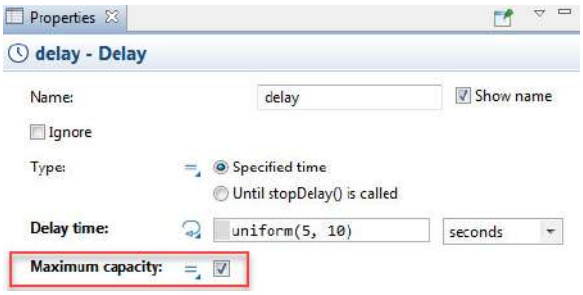


Figure C-9: Delay with an infinite number of entities able to be stopped at a time

In case the delay does not have any capacity limitation, you can check the **Maximum capacity** check box (Figure C-9). This means that the delay block can postpone an infinite number of entities simultaneously. Any entity encountering a delay block with maximum capacity will never have to wait to enter and may start the delay time immediately.

Select Output

The **SelectOutput** block routes incoming entities to one of the two output ports depending on a probabilistic or deterministic condition. This block is used when we want to separate entities into two groups that will continue moving along different flowchart branches. It is very important to keep in mind that entities spend zero time in a **SelectOutput** block. Contrary to other blocks (e.g., **Queue** or **Delay**), **SelectOutput** does not have any capacity, as entities can just pass through it in zero time. If the downstream blocks of a **SelectOutput** are blocked, the entities will not be able to pass through the **SelectOutput** and will wait in upstream blocks.

Selecting entities that exit from the True output based on probability

For now, we'll only focus on the probabilistic condition (and not the deterministic one). In the *Probability* field of a **SelectOutput** block's properties (Figure C-10), a value is specified indicating the chance the entity will exit from the outT (or True) port (Figure C-11). For example, a value of 0.7 means that there's a 70% chance for the True port to be chosen as the one to exit from. In the long run, when a lot of entities have passed through the Select Output block, the number of entities that exited from this port will be close to 70% of the total.

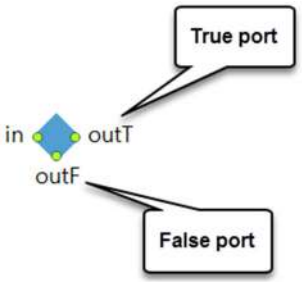


Figure C-10: Ports in a SelectOutput block

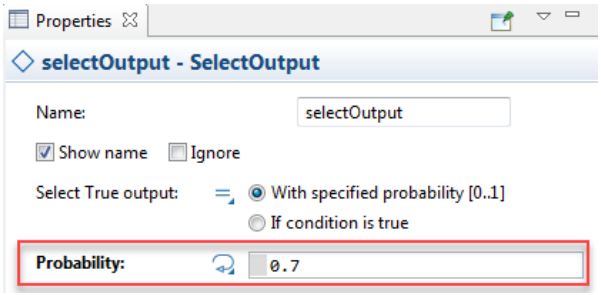


Figure C-11: Setting probability of exiting the True port

SelectOutput5

The **SelectOutput5** block is very similar to **SelectOutput** but with five out ports to route the incoming entities depending on a probabilistic or deterministic condition. As with before, we're only using the **Probability** option for now. With this enabled, five probabilities should be chosen for each of the exit ports and they all should add up to 1. If their sum does not equal 1, the values are first normalized (by having each value divided by the total; for example, probabilities of 4, 2, 4, 10, 5 will become 0.16, 0.08, 0.16, 0.4, 0.2).

Inner workings of the SelectOutput5 block [optional]

When you specify probabilities for different branches of the **SelectOutput5** block, it internally is assigned to conditions that use a discrete probability distribution which returns a boolean value (true or false) to each branch. The `randomTrue(p)` function internally uses a uniform distribution (between 0 to 1) to generate a random number; if the generated number is less than p, the function returns true. Therefore, the probability of returning true is p and the probability of false is $1 - p$. For example, `randomTrue(0.2)` means that true is returned 20% of the time and false is returned the other 80% of the time.

If you use the **Probabilities** setting in a **SelectOutput5** and assign equal chances to each port (20%), AnyLogic will internally set conditional choices that uses the `randomTrue` function as its conditions (Figure C-12).

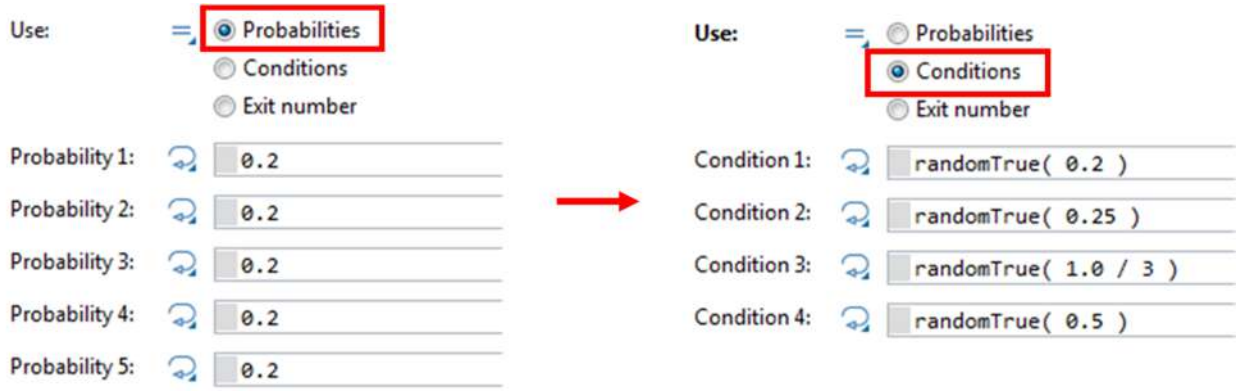


Figure C-12: SelectOutput5 probabilities are internally conditions with probability distributions

However, you'll notice that not all `randomTrue` distributions on the right side of Figure C-12 have the same 20% probability. This is because the probabilities of each subsequent condition are dependent on its

predecessors due to the domain of possibilities shrinking each time. If we want conditions 2 to 4 have the same probability of occurring as the first, we need to assign conditional probabilities to these fields that mathematically make them equivalent. Each condition is set to a percentage which is its (normalized) percent chance divided by the difference of one and the sum of the previous percentages.

For the example that is shown in Figure C-12, the following shows the value for each condition:

- Condition 1: $\frac{0.2}{(1-0)} = 0.2$
- Condition 2: $\frac{0.2}{(1-0.2)} = 0.25$
- Condition 3: $\frac{0.2}{(1-0.4)} = 0.\overline{3}$
- Condition 4: $\frac{0.2}{(1-0.6)} = 0.5$
- Condition 5: $\frac{0.2}{(1-0.8)} = 1.0$ (Not specified in AnyLogic due to being implied to be true)

Probability formula reminders:

$$P(A|\bar{B}) = \frac{P(A \cap \bar{B})}{P(\bar{B})} \quad , \qquad P(A|\bar{B},\bar{C}) = \frac{P(A \cap \bar{B} \cap \bar{C})}{P(\bar{B} \cap \bar{C})} \, , \qquad P(\bar{B} \cap \bar{C}) = P(\overline{B \cup C}) = 1 - P(B \cup C)$$

Assuming C1 = Condition 1, C2 = Condition 2, C3 = Condition 3:

$$P(C2 = true \mid C1 = false) = \frac{P(C2 = true \ \cap \ C1 = false)}{P(C1 = false)} = \frac{0.2 \ (defined \ target)}{1 - 0.2} = 0.25$$

$$P(C3 = true \mid C1 = false, \ \ C2 = false) = \frac{P(C3 = true \ \cap \ C2 = false \cap \ C1 = false)}{P(C2 = false \cap \ C1 = false)}$$
$$= \frac{0.2 \ (defined \ target)}{1 - 0.4} = \frac{1}{3}$$

Resource Pool

The **ResourcePool** object builds resource units and holds them. A **resource unit** is an individual resource, which is what the resource pool is made up of. The **ResourcePool** object itself is passive, meaning that it does not do anything on its own. It is just a storage for the resource units, until they're ready to serve the blocks that can use resources, like **Seize** and **Service**.

Functionalities of a ResourcePool object

The main functionalities of a **ResourcePool** can be summarized in three points:

1. It builds the resource units at the initialization of the model. **ResourcePool** is also capable of changing the number of resource units during the simulation if needed (it can build new units or remove existing ones).
2. Acts as a place to holds resource units. If an entity needs to use a resource unit, it can access the pool of available resource units.
3. Keeps track of the current status of each resource unit (if a specific unit is available or not) and changes the tasks of resource units. This point needs more explanation but for now just remember

that the resource pool knows the status of units and can tell us if a resource is available to the entities that request them.

Capacity of the ResourcePool

The ResourcePool's main property is its **Capacity** field which will specify how many resource units should be built at the model's initialization. We could define the capacity in many different ways, but we'll use the first option where units are **Directly** assigned (Figure C-13). For example, directly setting a pool of doctors to a capacity of 10 means the resource pool will build 10 doctors at the beginning of the simulation and make them available for use by entities in Seize or Service blocks.

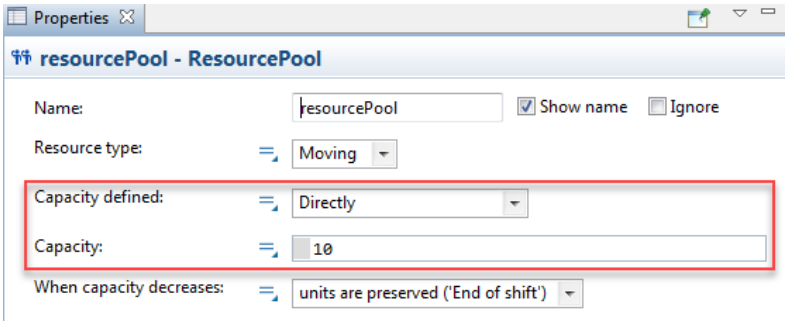


Figure C-13: Direct assignment of capacity to the resource pool

Different Types of ResourcePool

Resource units inside the pool are also have a type: Moving, Static, or Portable. You should consider these three types as a label that categorizes the resource as one of the three possible types:

- **Static** resources: bound to a particular location (that is node) and cannot move or be moved.
- **Moving** resources: can move on their own or move others.
- **Portable** resources: can be moved by others but not on their own.

By assigning a type to a resource, we make sure there's a clear understanding of its capabilities and a safeguard is in place against erroneous selection of a resource. If you select a wrong type, you should anticipate different behavior to be observed from the resource compared to what you've expected.

For example, let's say that you want a forklift resource to move toward and pick up an entity, drop it off at a destination, and then go back to its home location. However, you inadvertently specified the type of the forklift resource as static. As a result, the model will run, but the forklift won't move and only will be treated a just a logical constraint on the number of entities that can be moved concurrently. In most cases, leaving the type to default (**Moving**) is the least restrictive option and will work for all possible scenarios, knowing that the resource units' movements are modeled explicitly with the process blocks and a moving resource can also behave as either other type.

Seize

The Seize block grabs a given number of resource units from a given ResourcePool. If the entity needs a resource before it can progress through the process (e.g., a patient needs a nurse to be moved to the examination room), a Seize block will do the job of checking the availability of the needed resource units and grab them if available. However, the needed resource units are not always available and therefore

the entity that needs a resource has to wait for the next available resource units. Therefore, the **Seize** block has an internal queue to keep the entities until the needed resource becomes available. As soon as the resource becomes available, the waiting entity seizes it which allows the entity to move forward with the next step in the process. The internal queue should have a minimum capacity of one, representing the next agent waiting in line to seize a resource. If you don't want an entity to enter this queue when there are no available resources, you should stop the entities from progressing into the seize block with other mechanisms (e.g., with a **Hold** block).

As described in the **Queue** block description, the time that entities spend in a queue is not associated with the behavior of the **Queue** block itself but is rather the consequence of interaction with other blocks. For the internal queue of the **Seize** block, the time that entities spend in this queue is related either to the availability of resources or blockage of downstream process blocks. If the needed resource for an operation is not available (e.g., no available nurse to move the patient to the ER) then the entities will accumulate in the internal queue and wait for the next available resources. Therefore, you should consider the availability of resources as a constraint on the movement of entities in the flowchart (e.g., if there are 10 patients waiting to be moved to the ER and we have only two nurses, then we only can move two patients concurrently and the rest must wait for the nurses to return).

There are four main attributes of the Seize block: **Seize mode, Resource pool, Number of units,** and **Queue Capacity**:

Defining the Seize mode

The **Seize** field could either be either to **(alternative) resource sets** or **units of the same pool** (Figure C-14). The alternative resource set option is the more comprehensive option that lets you choose a combination of resources from different **ResourcePool** objects (e.g., one doctor and two nurses) or even set an alternative sets (e.g., if one doctor and two nurses are not available, two doctors could do the operation as well). A detailed explanation of **(alternative) resource sets** is provide in the **MODEL BUILDING BLOCKS – LEVEL 2**. For now, our focus is on the simpler option, **units of the same pool**. This option means all the needed resources are from the same **ResourcePool** (e.g., doctors). If all the resource units needed for a task are from one pool you can assign that pool to the **Seize** block with the two following settings (Figure C-14 and Figure C-15):

- **Resource pool:** if you select the **units of the same pool** option, you can select your desired **ResourcePool** object from this dropdown menu.
- **Number of units:** This option sets the number of units that are needed for each entity passing through the **Seize** block (e.g., 3 workers to move a pallet).

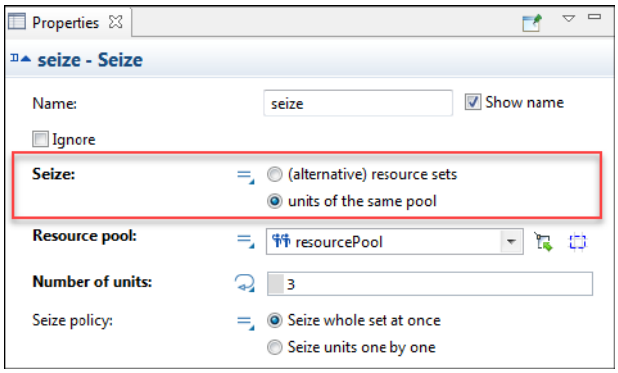


Figure C-14: Setting the seizing type option

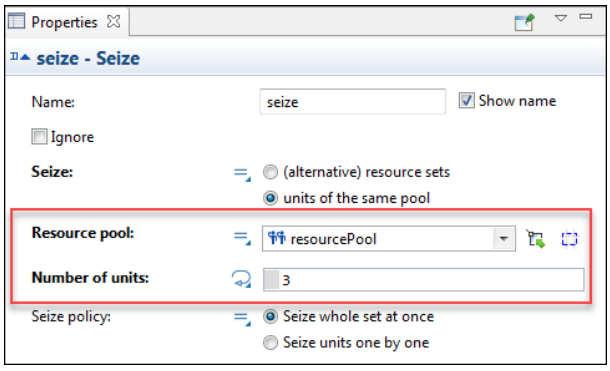


Figure C-15: Assigning a resource pool to a Seize block

Defining the capacity of embedded Queue

The **Queue capacity** field sets the capacity of its internal queue. This capacity sets the number of available spots in the queue that will accumulate the entities that cannot seize a resource unit immediately and pass through the Seize block (Figure C-16).

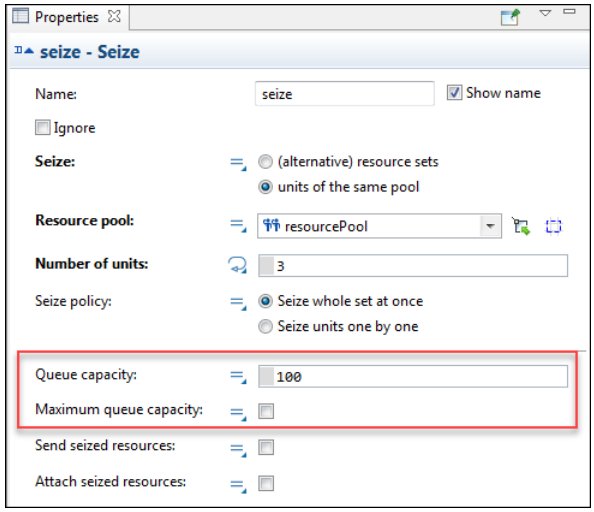


Figure C-16: Assigning a capacity to the embedded queue of the Seize block

If you select **Maximum queue capacity** the queue will have infinite capacity to accumulate incoming entities.

Release

The entity will eventually reach a point in the process that it does not need the seized resources anymore. At this point, we release the units and put them back in the resource pool with a Release block. AnyLogic gives you an error (*RunTimeException: Agent being disposed possesses [X] unreleased resource units*) if an agent that seized a resource leaves the model (e.g., goes to Sink) without releasing the resource first. Release has several modes but for now we're only using the default **All seized resources of any pool**, which releases all the resources that the entity had seized in its journey up to this Release block (Figure C-17)

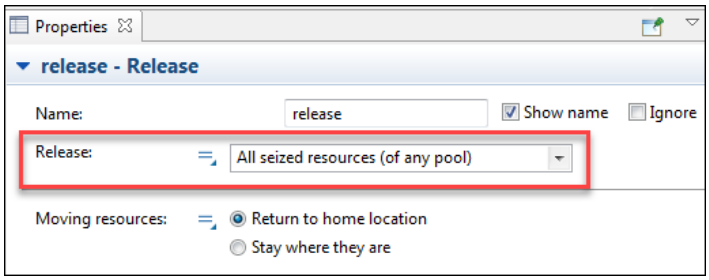


Figure C-17: Releasing all seized resources of any pool

Service

Service is a special block that is internally a combination of the **Seize**, **Delay**, and **Release** blocks (Figure C-18). As mentioned, the **Delay** is the most fundamental block in abstracting a process. A **Delay** block bundled with a **Seize** and **Release** will give us the most generic form of abstraction for an activity which takes some time and needs resources for its completion. The internal **Delay** in the **Service** block is set to have maximum capacity, meaning that the maximum number of entities that can enter the internal delay is controlled by the availability of resources and not the delay's capacity. **Service** is designed in a way that most attributes of its internal blocks (**Seize**, **Delay**, **Release**) are exposed for customization. However, there are a few advanced settings that are only accessible in the original building blocks.

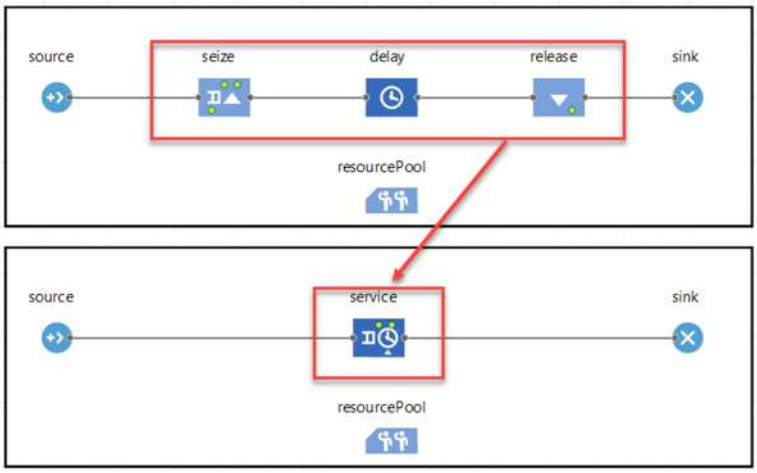


Figure C-18: Service is a combination of the Seize, Delay, and Release blocks

It should be mentioned that **Service** can only be used in cases that we only want *one* **Delay** in between **Seize** and **Release**, otherwise we should directly use **Seize** and **Release**.

TimeMeasureStart

You can pair **TimeMeasureStart** and **TimeMeasureEnd** blocks to measure the time the entities (or resource units) have spent between them. You can imagine that whenever an entity passes through this block a time stamp is added to it. When it reaches a **TimeMeasureEnd**, the time difference between that moment and the recorded time stamp would be the elapsed time between the two points for the entity. Having a **TimeMeasureStart** on its own or unpaired to a **TimeMeasureEnd** is possible; however, it is unadvised since it would functionally do nothing.

TimeMeasureEnd

This block is used to mark the end point for a paired **TimeMeasureStart**. Unlike its counterpart, a **TimeMesaureEnd** must be paired with a **TimeMeasureStart** - your model will build successfully but you will encounter a runtime error once an entity tries to pass through it. **TimeMeasureEnd** has internal tracking of the elapsed times for the entities that have passed through it. In addition to the option to pair a specific **TimeMeasureStart** block, the main property available is **Dataset Capacity**.

Capacity of TimeMeasureEnd

Each **TimeMeasureEnd** has two embedded datasets to collect the traverse times: a regular dataset and a histogram data (distribution). When an entity reaches the **TimeMeasureEnd**, the time it took from the paired **TimeMeasureStart** will be added to the embedded dataset and histogram data. The **Dataset capacity** field specifies the capacities of the dataset. When the number of entities that have passed through this block exceeds the specified capacity, the earliest values will be removed from the dataset. However, the counts stored in the histogram data will be preserved. For example, if the capacity is set to 100, and 1000 entities have passed through this block, the dataset will contain the specific times for entities 901 to 1000 and the histogram data will show the distribution for all 1000.

RestrictedAreaStart

Some blocks like **Queue**, **Delay**, **Seize**, etc. have a capacity which prevents no more entities than the specified capacity to be inside the block at the same time. However, there are cases that the limitation on the number of entities is not related to one block, but a group of connected blocks in the flowchart. In these cases, we can put **RestrictedAreaStart** and **RestrictedAreaEnd** blocks around the area to limit the maximum number of entities inside the subsection of the process that they encompass. If you do not pair this block with a **RestrictedAreaEnd**, an runtime error will be thrown when the capacity has been reached. The main property available to this block is **Capacity (max allowed)**:

Capacity (max allowed) of RestrictedAreaStart

The capacity specified here dictates the upper limit of entities allowed between the start and end of a given section. Once the number of entities that entered the **RestrictedAreaStart** minus the number of entities that exited the **RestrictedAreaEnd** reaches the limit, the **RestrictedAreaStart** object blocks the entry and would not allow a new entity in until one of the entities exits the area.

Functions

`boolean isBlocked()` : Returns whether the restricted area is full. If the capacity has not yet been reached, this method will return true, otherwise false.

RestrictedAreaEnd

This block is required be paired with a **RestrictedAreaStart** to specify the subsection of the process flowchart that has a limitation on the number entity that it can contain. Not pairing this block to any **RestrictedAreaStart** will result in a runtime error. You can pair several **RestrictedAreaEnd** blocks to one **RestrictedAreaStart** if needed.

Model Building Blocks - Level 2

In this section, we'll go over more sophisticated parts of different AnyLogic blocks – some from the Level 1 blocks section are expounded upon and others have not yet been discussed.

Source

In this section, we'll discuss how you build more sophisticated arrival mechanisms from **Source** blocks: arrival rates that are changing over time (nonstationary Poisson processes) via schedules, multiple entities per arrival, and limiting the number of arrivals. We also cover how to change the default animation of an entity with your own custom animation.

Defining the arrivals by rate schedule

We previously discussed two options that are available from the **Arrivals defined by** option: **Arrival rate** and **Interarrival time**. The arrival rate lets you assign constant arrival rates; however, in most realistic scenarios, arrival rates changes over time. Mathematically speaking, arrival rate is a function of time and thus deals with a nonstationary Poisson process. To calculate nonstationary Poisson process, we usually separate the time into subintervals (that could be equal or different) and then find the average arrival rate for each subinterval. In AnyLogic, the arrival rates that are calculated for these subintervals could be assigned to the source by the **Rate Schedule** option. To do this, you must define an **Schedule** object and specify the arrivals in each subinterval which you then assign to a **Source** block (Figure C-19). To learn more about the **Schedule** please refer to its description in the next section.

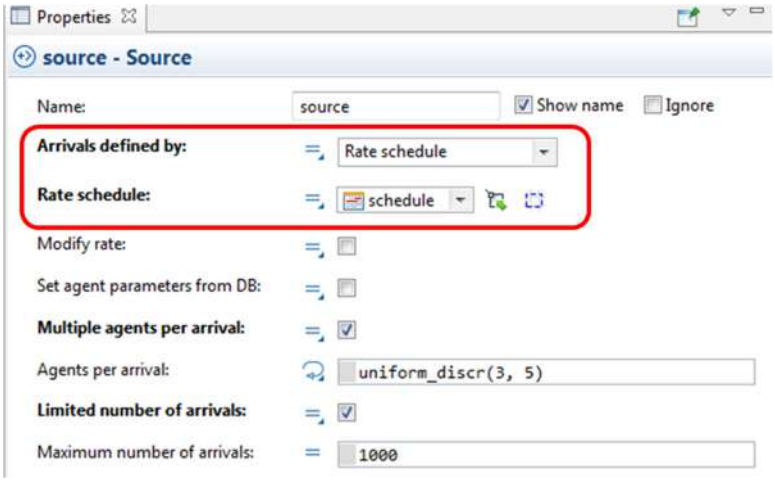


Figure C-19: Arrivals defined by Rate schedule

Multiple entities per arrival, and Limited number of arrivals

The **Multiple agents per arrival** option lets you inject several entities at each arrival. The number of entities per arrival could be a deterministic or stochastic value (from a distribution), as seen in the top rectangle of Figure C-20.

The **Limited number of arrivals** option forces a **Source** block to stop injecting entities into the process after a specified number of arrivals have been reached (Figure C-20). It is important to note that this setting limits the number of arrivals and not the total number of arrived entities. For example, if **Agents**

per arrivals is set to 3, and **Maximum number of arrivals** is set to 10, no more new entities will be injected after the 10th arrival (at which point there will be 30 entities in the process).

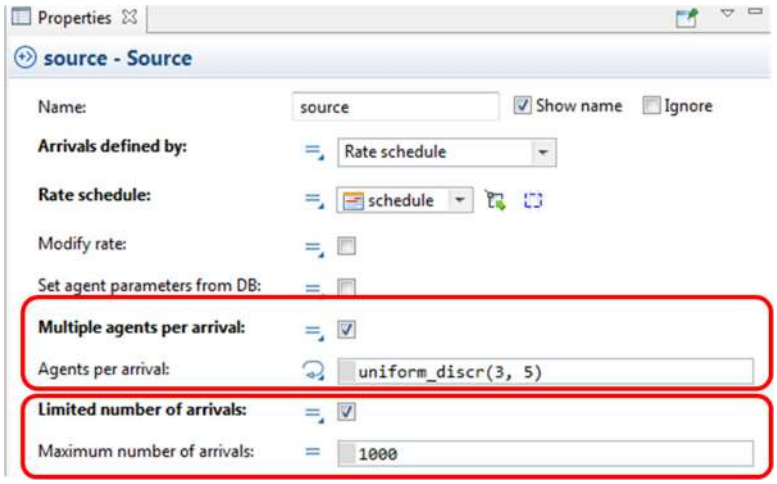


Figure C-20: "Agents per arrivals" and "Maximum number of arrivals" fields in Properties window of Source

Customizing the entity's animation representation

Source uses a default template/blueprint (Agent) to build new entities. The default animation is a circle with a randomly assigned fill color. We can substitute the default blueprint with our custom one. To do so, you need to click the **create a custom type** link below the **New agent** field under the **Agent** section (Figure C-21).

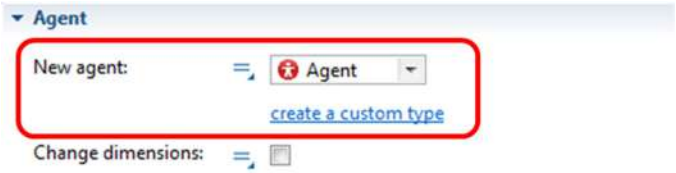


Figure C-21: Link to building a custom agent type (customizing the entity's animation)

By clicking on the link, a wizard will open that lets you first name your Agent type (the blueprint). It is a convention that agent type should start with an upper-case letter. After clicking the **Next** button, you will be taken to the next step of the wizard where you can select the agent animation (Figure C-22). From this, you can select from the list of available 3D objects, 2D shapes, or no animation at all. Afterwards, click **Finish** to close the wizard and be taken to the new Agent's environment. Your chosen animation will appear in the origin (intersection of X and Y-axis). If you selected **None**, you can draw your own shapes at the origin using the objects available in the **Presentation** palette.

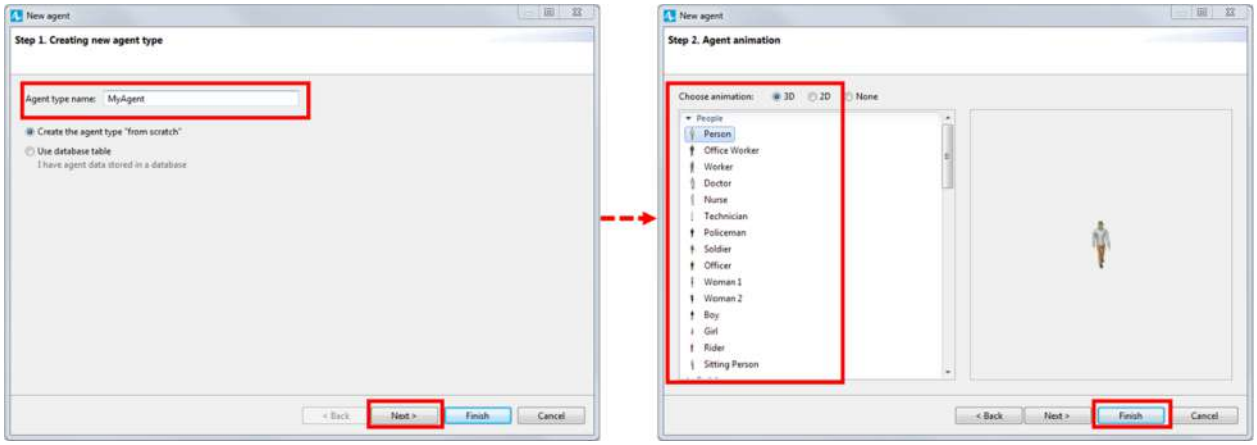


Figure C-22: Naming the agent type and choosing the animation

Location of arrival

Since **Source** is the block in which entities get generated and enter the model, there's an option to assign the location of newly arrived entities. There are six options, but the two main ones that we'll focus on are:

- **Not specified**: No specific location is assigned to a new entity. This is the default option and is used when our model does not have any physical component and is not a networked-based model. Whenever our model only represents the system at logical level (and not physical) this option is selected.
- **Network / GIS node**: This option is used to assign an initial location to entities when we're building a network-based model and parts of the process logic depend on physical space. With this option selected, you can to assign a node (Point, Rectangular, or Polygonal).

Schedule

The **Schedule** object can be imagined as a function that takes the current time as its argument (input) and returns a value (output). In other words, a schedule can map numerical values to moments of time or *time intervals*. This block has many different setting combinations, but only two particular applications will be focused on:

Interarrival times that change over time

If the interarrival time changes over time, these changes need to be mapped to time intervals (e.g., the interarrival time is set to one hour between 8:00 and 9:00 am and is set to three hours between 9:00 to 10:00 am). In this case, the **Type** field should be set to **real** (integers or decimal numbers) and **The schedule defines** field should be set to **Intervals (Start, End)**.

If your model operates based on a repeating period of time, you should select **Custom** for the **Duration type** field. Doing so will open the **Repeat every** field where you can set the period length. The period is the time required for the **Schedule** function to complete one full cycle. With these settings enabled, the **Start**, **End** and the **Value** columns are filed out for each interval (Figure C-23).

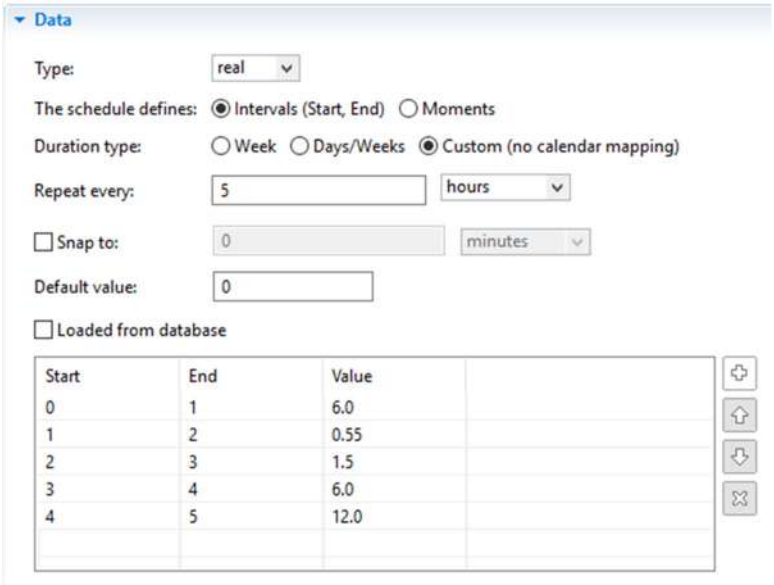


Figure C-23: Schedule for mapping changing interarrival times to time intervals (no calendar mapping)

Alternatively, if specific days of the week define your model time, you can select **Week** for the **Duration type** field and map the interarrival times to an actual calendar. In this case, the period of the schedule is one week. You then map days and hours of the week to real numbers (values). You also should set a default value for times outside the mapped time intervals (Figure C-24).

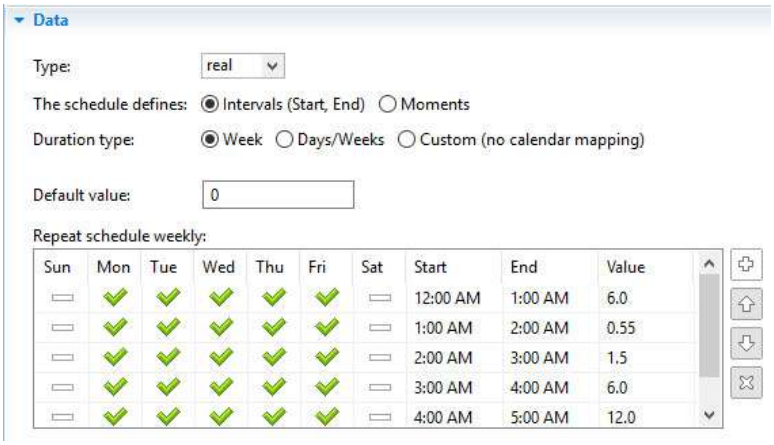


Figure C-24: Schedule for mapping changing interarrival times to time intervals (with calendar mapping)

Regardless of the way you've assigned the interarrivals times to the time intervals (with or without calendar mapping), the **Schedule** could be assigned to the **Source** by calling its name like a function (Figure C-25). Since the value that Schedule returns is unitless, the unit of time should be correctly selected in the block that uses the **Schedule** (e.g., **Source**).

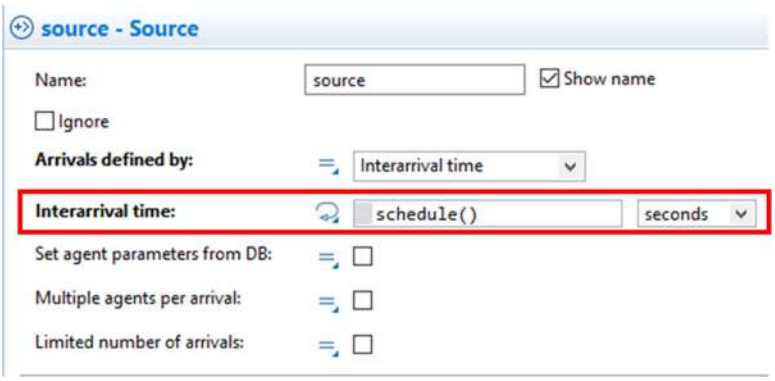


Figure C-25: Assigning the Schedule to the Source to specify the interarrival times

Arrival rates that are changing over time

The **Schedule** object could also be used for defining nonstationary Poisson processes (in which arrival rate is a function of time). In this particular use-case of **Schedule**, you should set the **Type** field to **Rate** and set the **Unit** field accordingly. Like the interarrival time, you can map the intervals without calendar dates. In this case, the value column shows the rates instead of interarrival time (Figure C-26).

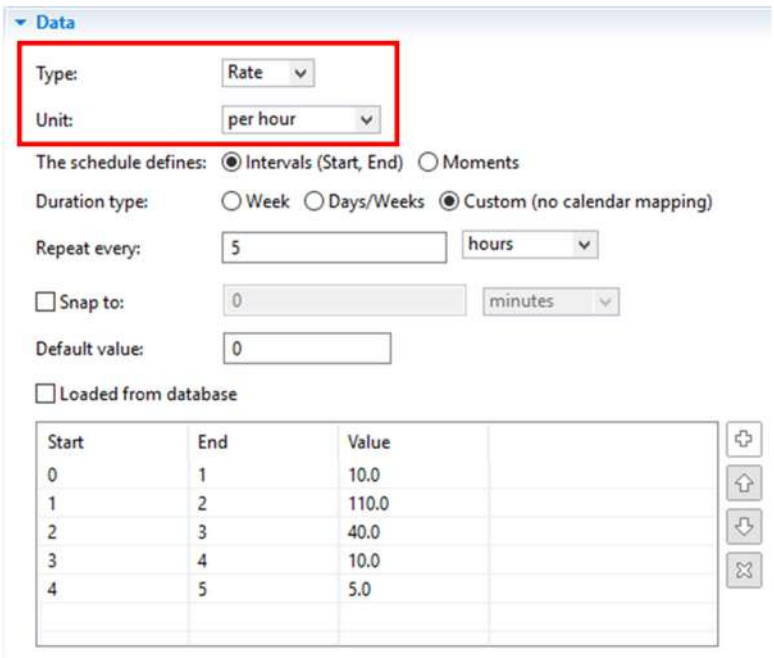


Figure C-26: Schedule for mapping changing arrival rates to time intervals (without calendar mapping)

If your model time is set based on calandar dates, you can map the intervals based on time and day of the week with an arrival rate (Figure C-27).

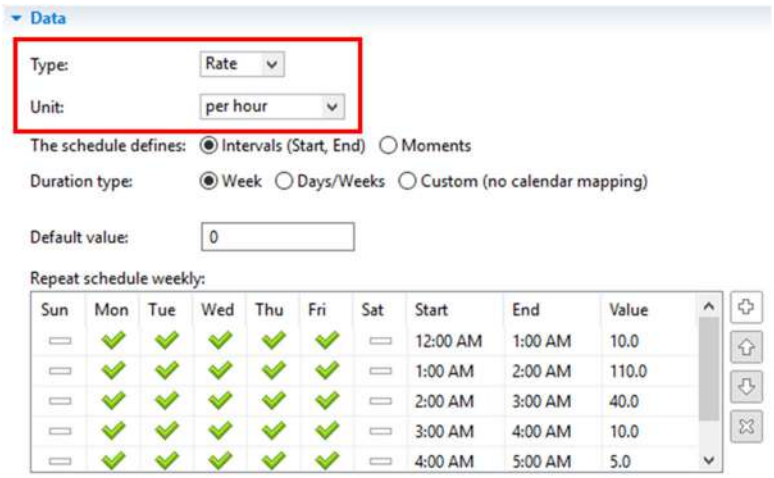


Figure C-27: Schedule for mapping changing arrival rates to time intervals (with calendar mapping)

To assign a nonstationary Poisson arrival process to the **Source**, you must select the option for **Rate Schedule** for the **Arrivals defined by** field and then select an appropriate schedule from the drop-down menu (Figure C-28 below). As you may have noticed, the **Rate schedule** field does not have a time unit option. This is because the unit is read from the unit set in the **Schedule** object (Figure C-27 above).

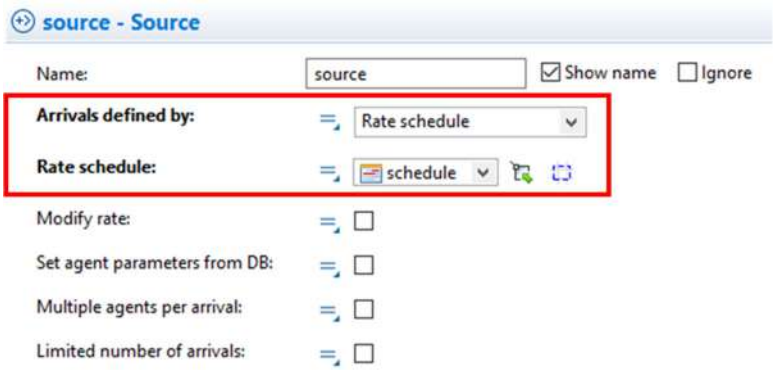


Figure C-28: Assigning the Schedule to the Source to specify the Rate (the rate that changes over time)

ResourcePool

In this section, we talk about additional properties of the **ResourcePool** object we mostly use in network-based models.

Home location of resource units

Resource units are introduced to the model in the **ResourcePool** object they're built in. In non-networked-based models where a resource pool does not have a physical representation, we leave this field blank. For network-based models, one or more nodes (Point, Rectangular, or Polygonal) are selected (Figure C-29). If we have more than one unit in the pool and the capacity of the pool is defined by the **Directly** option (with an integer provided), AnyLogic will arbitrarily assign the units to the nodes (and their internal attractors, if any exist). However, in most network-based models, it is a better option to set the capacity of the **ResourcePool** based on the **Home location**, as we'll discuss below.

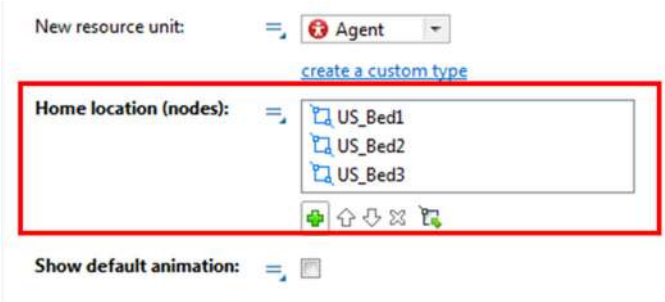


Figure C-29: Home location of resource units

Capacity of resource pool is defined "By home location"

We know we can use an integer value to directly set the ResourcePool capacity. In network-based models that the ResourcePool have a physical representation, the capacity could be defined **By home location**. When you select the capacity to be defined by home location the **…based on attractors** setting associated with it gets enabled and is selected by default (Figure C-30).

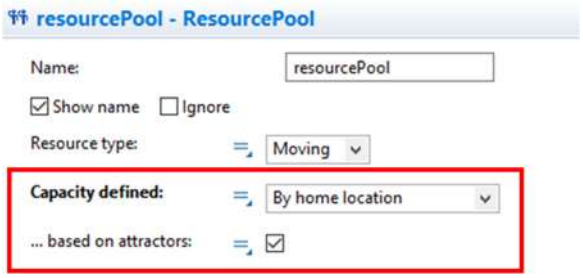


Figure C-30: Capacity of the ResourcePool is defined based on the home location node(s) and its (their) attractors

When **By home location** sets the resource pool capacity, AnyLogic builds resource unit based on assigned nodes (and their internal attractors if the **…based on attractor** setting is checked). In case the resource pool is set by home location, the home location could be set as:

- One node with multiple attractors inside – AL will build one unit per attractor
- Multiple nodes without attractors – AL will build one unit per node
- Multiple nodes with attractors inside - AL will build one unit per attractor

Customizing the units' animation

Similar to the Source block that uses a blueprint (Agent type) to build the entities, the ResourcePool block uses a blueprint to build the resource units. The default **Agent** type assigned to the ResourcePool has a circle with a random fill color as its animation. To customize the animation, we can click the **create a custom type** link in the ResourcePool properties window (Figure C-31).

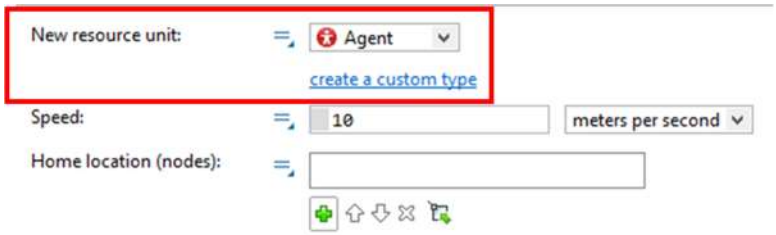


Figure C-31: Link to create a custom resource animation

The link opens up a wizard that lets you name your custom resource type (by convention the name should start with an upper-case letter). In the next step, you select a 2D image or a 3D object as the custom animation of your resource units (Figure C-32). You can also select no animation and draw your own using basic shapes provided in the Presentation palette.

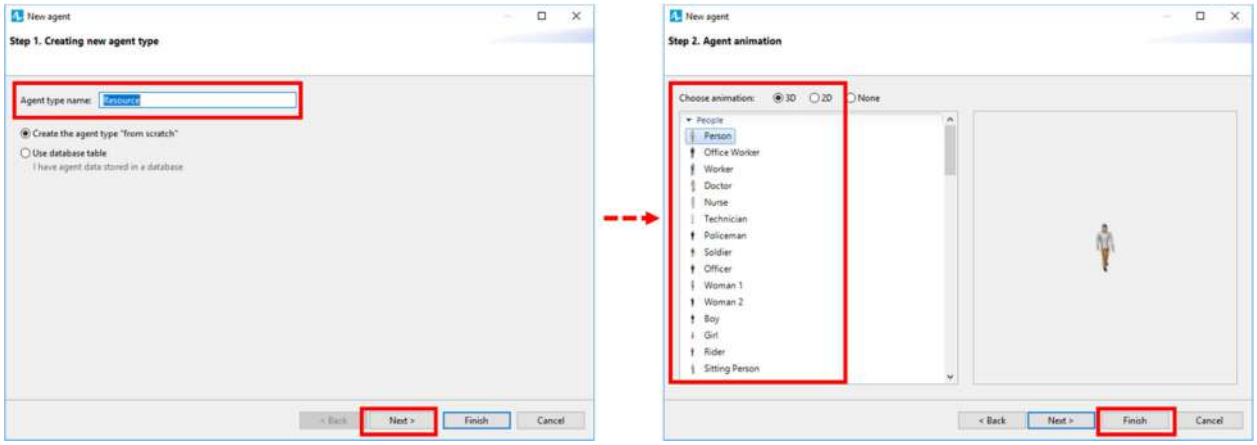


Figure C-32: Wizard to build the custom resource type and its animation

Speed of resource units

If your **ResourcePool** type is set to **Moving** (not **Portable** or **Static**), then you can assign a moving speed to your resource units (Figure C-33). However, if you attach a resource unit to an entity, the resource unit will move with the entity's speed, regardless of its speed defined here.

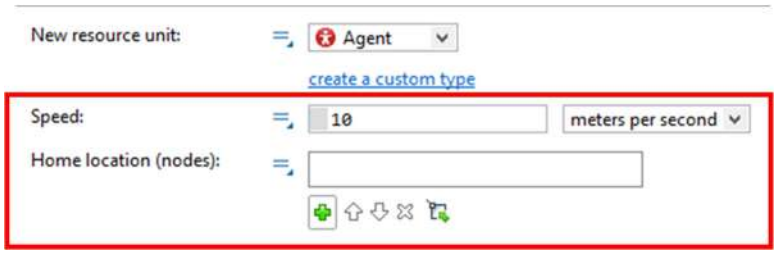


Figure C-33: Speed and Home location of a ResourcePool block

Seize

In this section, we'll expound upon the previous explanation of the seize block by going into the details of the second available mode, **alternative resource sets**, which we did not cover in **LEVEL-1** blocks section. We'll also look at the "*preparedUnits*" port of the **Seize** block and its application in assigning preparation tasks to seized resource units. We also look into how to assign priorities to the tasks and differentiate between the different attributes of the **Seize** block that let you control the movement of seized resource units. Finally, we'll learn how the priorities could be used to preempt other tasks that have already been started.

Alternative resource sets

If all the resource units needed for a task are from the same resource pool, we should select the **units of the same pool** option in the **Seize** field. However, there are cases where needed resource units are not all from a single resource pool. In these scenarios, we should select the **(alternative) resource sets** option. By selecting this option, we can define alternative sets of resources – an example of this can be seen in Figure C-34 where the resource sets consist of: one doctor and two nurses *or* one doctor, nurse, and technician *or* two doctors. These alternative sets are tested in the order they're listed. In this example, if one doctor and two nurses are not available, the model will next try to seize one doctor, nurse, and technician. If neither are available, it will finally try to seize two doctors. If none of these sets are available, the entity will wait in the embedded queue.

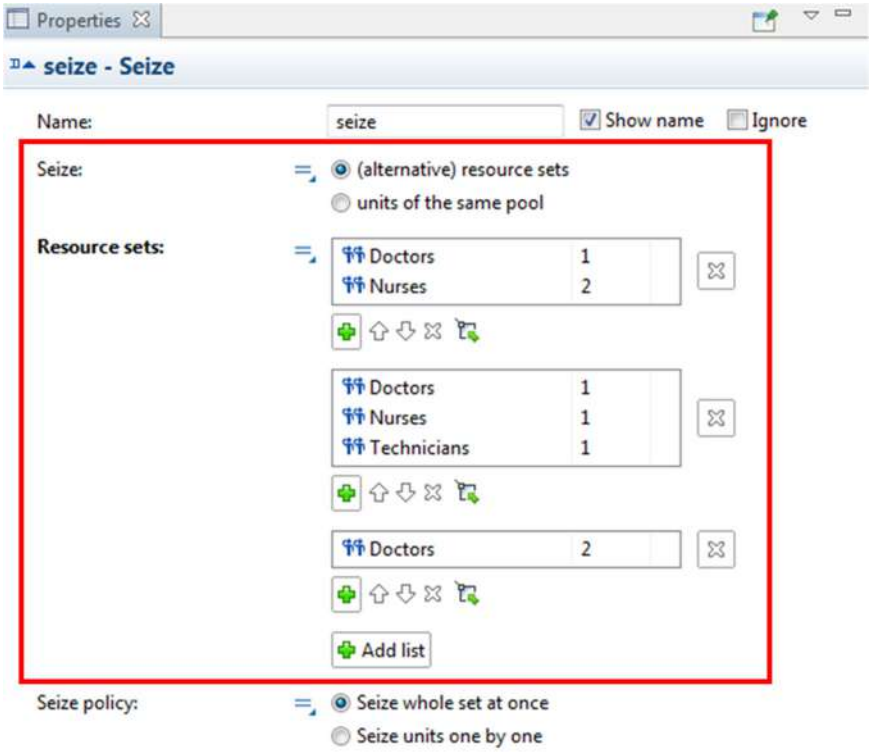


Figure C-34: Alternative resource sets in a Seize block

In general, if all your resource units are from the same resource pool you can also use the **(alternative) resource set** option. On important caveat in using the **(alternative) resource set** option is to be mindful

of adding empty sets by mistake. If you add a set (by clicking on the **Add list** button), but don't add resource pools to the set, there will be an empty set as one of the alternatives. Regardless of its position in the list, AnyLogic will assume you don't want *any* resource pools to be used. This means it won't use any of them.

Adding a preparation branch to the Seize block

The **Seize** block also has a "*preparedUnits*" port that lets you connect a preparation process branch (Figure C-35).

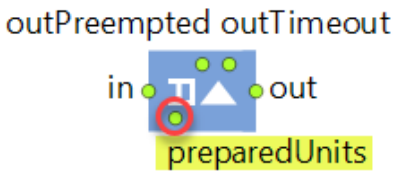


Figure C-35: "preparedUnits" port of the Seize block

If the seized unit has some tasks it needs to perform in preparation of helping the entity, a process can be started with a **ResourceTaskStart** block and connects to the "*preparedUnits*" port of the **Seize** block.

Send seized resource

If you are working on a network-based model, you can choose to automatically send a seized resource unit to a defined destination upon being seized (Figure C-36). This works similar to a hidden **MoveTo** block that moves the resource unit to a destination. There are several options for the seized unit's destination:

- **Agent** - resources units are sent to the entity (agent) that seized them
- **Network node** - resources units are sent to a specified network node (Point, Rectangular, or Polygonal). If the nodes have internal attractors, the units will go to those attractors.
- **Other seized resource unit** – resource units are sent to the current location of another seized resource that is also seized by the **Seize** block.
- **Home of seized resource unit** – resource units are sent to their home location.

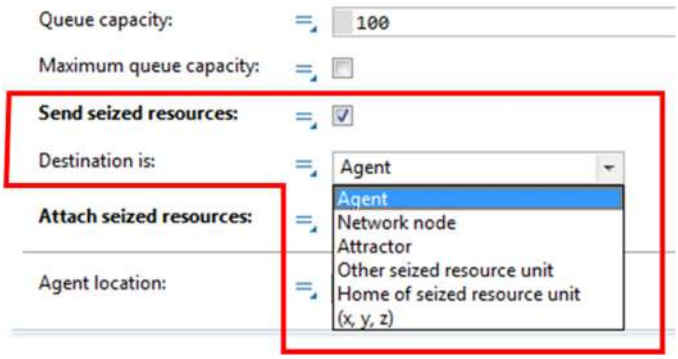


Figure C-36: Send the seized resource unit to a specified destination

Attach the seized resource unit to the entity (agent) that seized it

You can attach the seized resource unit to the entity that seized it by checking the box for **Attach seized resources** (Figure C-37). From that point on, the entity and the resource unit move together (the speed of the couple is the speed of the entity by default). This option is often used when you are sending the seized resource unit to the entity and then want them to move together from that point on.

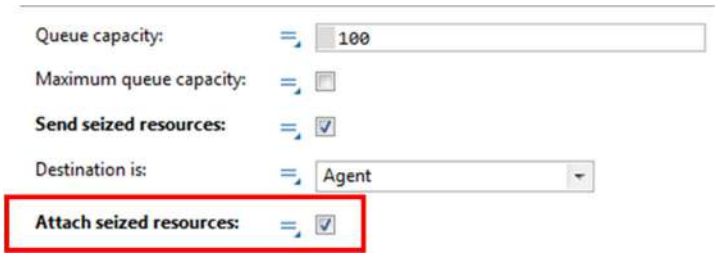


Figure C-37: Attaching the seized unit

For example, a forklift is seized and moves to the package that seized it (**Send seized resource** is set to **Agent**), and then gets attached to it. Afterward, the forklift and the package move together. Animation of the coupled entity and resource unit assumes a default offset between the original animations (that is, image representing the package and forklift). You can use the **PMLSettings** block to adjust this default offset.

Assigning priority to the incoming tasks

When an entity enters the **Seize** block, its associated task needs to be assigned to a resource unit by the **Seize** block (Figure C-38). We can assign a priority value to each incoming task in order to differentiate between them. If several tasks are requesting units from a mutual resource pool and there's no idle unit, these requests will queue up in the resource pool. As soon as the resource becomes available, the task with the higher priority value will seize it. For a detailed explanation of task priority, please refer to **TECHNIQUE 4 IN CHAPTER 3**.

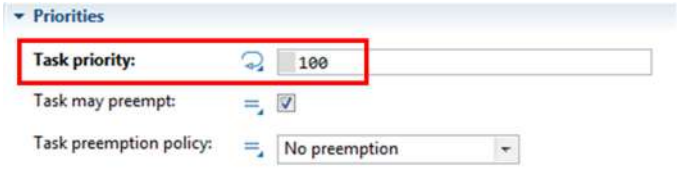


Figure C-38: Task priority setting of the Seize block

The **Task priority** field is a dynamically evaluated field. This means its value is updated many times during the simulation run.

Preemption policy

There are two settings related to the preemption policy in the **Properties** of the **Seize** block. These two setting are placed close to each other, but they're pointing to two separate mechanisms that govern the preemption policy.

The first is the **Task may preempt** option (Figure C-39). If this checkbox is selected, the tasks coming from *this* **Seize** block can preempt other tasks that have a lower task priority, stealing their resource.

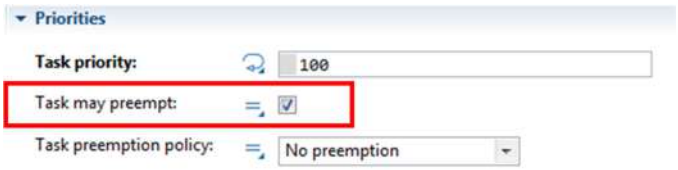


Figure C-39: Task may preempt checkbox in the Seize properties

Being able to preempt other tasks is only possible if those tasks allow it to be preempted. This permission is modeled with the **Task preemption policy** checkbox, which is about giving permission to others to act on the tasks coming from this **Seize** block. This is contrary to the other checkbox (**Task may preempt**) which references the action *this* **Seize** block can take on *others.* By default, the **Task preemption policy** of seize blocks is set to **No preemption**, meaning no other tasks (regardless of their priority value) can preempt the task coming from this seize block.

If we want others to be able to preempt the task coming from a seize block, we need to specify what should happen next. A preempted task results in an unfinished task that needs to be managed; this is done by one of several options given for the **Task preemption policy** (Figure C-40):

- **Wait for original resource**: the interrupted task waits for the original resource unit to finish with the other, higher priority tasks and come back and finish this process.
- **Terminate serving (simply remove from the flowchart)**: the entity that was associated with the task is removed from the flowchart. This behavior usually makes sense for entities that represent non-material things like orders, calls, and requests.
- **Seize any resource**: the interrupted task waits for any other free resource unit to finish it.

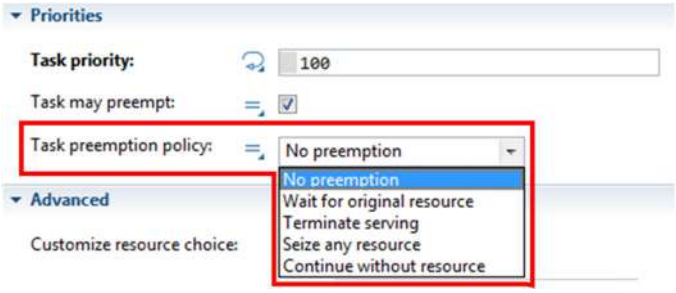


Figure C-40: Task preemption policy of the Seize block

Release

In this section, we'll discuss more selective ways of releasing seized resource units, connecting a wrap-up branch to the **Release** block, and assigning common movements to released resource units.

Selective releasing of resource units with Release block

We know about the capability of **Release** to separate seized a resource unit from an entity and return it to its associated resource pool. However, an entity may have seized several resource units by different **Seize** blocks or from different **ResourcePool** objects. Therefore, release blocks have five options to let you release specify the resource units that should be released:

- **All seized resources (of any pool)**: All resource units seized by the entity will be released.

- **All resources seized by given Seize block(s)**: The resources seized from one or more specified **Seize** blocks released. This option is especially useful since all the units seized by a specific block are related to a specific task; at the end of that task, we want to release all of the ones for that task.
- **All seized resources from given pool(s)**: Resource pools are individually specified to release all seized resource units of that pool to. One scenario that this option could be useful is when an entity seized several units of the same type (from the same resource pool) but needs to release all of them at once.
- **Specified resources (list of pools)**: Single resource units from the specified pools are released. If you wish to release more units of the same pool, you can add them to the list as many times as needed.
- **Specified quantity of resources**: This option is used that we want to release a specific number of resource units from a specific resource pool. The options allow you to select a specific resource pool and the number of units that you want to be release back to it.

Connecting a wrap-up branch to the Release block

The **Release** block has a "*wrapUp*" out port that lets you redirect the released units to a wrap up process branch, as shown in (Figure C-41).

Figure C-41: "wrapUp" port of the Release block

Execution of wrap up tasks can be controlled via the **Wrap-up tasks** section of the **Release** block's properties; wrap-up tasks can be set to run:

- **each time**: every time a resource unit is released by the specific **Release** block, it will go through the tasks in the wrap up branch.
- **if no other tasks**: released units will go through the wrap up branch only if no other task is waiting to be assigned to the release unit. Otherwise, they skip the wrap up and get assigned to a new entity immediately after the release.

The units released via the "*wrapUp*" port won't be readily available for another task until they complete the branch. Despite being no longer used, these resource units will remain in a busy state will not be available to be seized by other entities. The released unit via the "*wrapUp*" port will become idle and available when they completely traversed the wrap up process and reach the **ResourceTaskEnd** block.

Assigning common movements to released resource units

Beside releasing resource units, **Release** block can tell those units where to go after it releases them. The movement instructions only work for released resource units of type **Moving**.

If the **Release** block has a wrap up branch connected to it, in almost all cases it is better to select the **Stay where they are** and delegate the needed movements to the warp-up branch. If you assign a **Stay where they are**, to a **Release** block without a wrap up branch, basically no movement will be assigned to the release unit.

In case there's no wrap up branch attached to the **Release** block and you want to force the moving units to return to their home location (that is set in their **ResourcePool** block), you have several options in the **Release** block:

- **Return to home location** (Moving resources field) + **each time** (Wrap-up field): each release unit will return to its home location after the release.
- **Return to home location** (Moving resources field) + **if no other tasks** (Wrap-up field): released units only return to their home location if there's no other entity waiting to seize them. Otherwise the return to home location task is canceled and the unit will be seized immediately after release.

ResourceTaskStart

If you have a preparation branch that is connected to a **Seize** block, then you need a **ResourceTaskStart** block as the starting point of your preparation branch (please refer to **TECHNIQUE 4** for more information about the preparation branch).

Selection of resource units that should complete the preparation tasks

The **ResourceTaskStart** block lets you to be selective of the resource units that were seized. You have two options in **Start here** field:

- **All the resources**: All resource unit that were seized must go through the preparation tasks
- **Specific resources**: Only specific resource units must go through the preparation tasks. Choosing this option gives you the option to select a specific resource pool.

ResourceTaskEnd

If you have a wrap-up branch that is connected to a **Release** block, then you need a **ResourceTaskEnd** block as the end point of your wrap-up branch (please refer to **TECHNIQUE 4** for more information about the wrap-up branch).

Releasing the resources that were seized by the unit

Units traversing through the wrap-up branch can also seize resource units. If we look at the preparation or wrap-up branch in isolation, we can imagine the unit passing through them as an entity – therefore, they can seize resource units to complete their tasks. **ResourceTaskEnd** has a checkbox **Release resources seized by this unit** that let us release all the seized units and has not been released by the unit during the preparation or wrap-up branches.

Move the released resource units back to their home location

When a unit reaches to the end of wrap-up branch, we can send it to its home location by checking the **Move resource to its home location** checkbox. Alternatively, you can add a **MoveTo** block to the wrap-up branch and move the unit to a desired location as part of the wrap-up process.

Queue, Seize, Service

In Chapter 1 we've discussed some common behaviors of entities in regard to queues. We'll see how we can model two of those behaviors very easily with two properties of the **Queue** block.

Balking behavior (preemption of entity's entrance to the queue)

If happens very often that entities cannot enter the queue because the queue is at its capacity. You should take into consideration that the capacity limit of a queue could be due to a physical constraint (e.g., a checkout line with space for 20 people, with 20 people currently waiting in it) or a cut-off decision (e.g., no more than five people allowed in the waiting list). As you may remember, "balking" is common behavior of entities where an entity makes a decision to not join a long queue. This behavior could be easily modeled in the process by enabling the preemption property of the **Queue** block. To do this, check the **Enable preemption** checkbox in the **Advanced** section of a **Queue** properties window (Figure C-42).

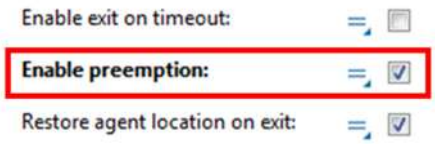


Figure C-42: activating the preemption behavior of a Queue

In this case, when an entity attempts to get into a full queue, instead of being blocked from the entrance, it will be rerouted via the "*outPreempted*" port leading to a different flowchart. In the example model shown in Figure C-43, entities unable to enter the queue when its full will go to the **preemption_sink**. Instead of sending the preempted entities to a **Sink** block, you can have a full-fledged flowchart that is connected to the "*outPreempted*" port.

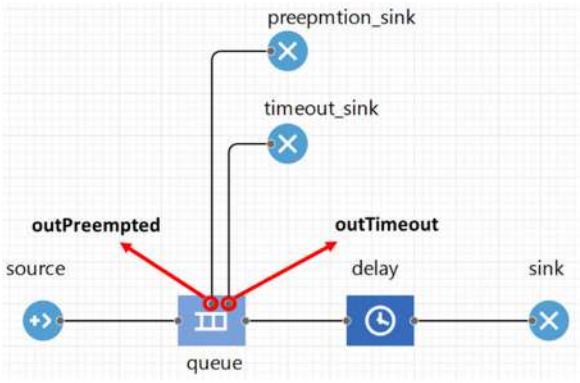


Figure C-43: "outPreemted" and "outTimeout" ports of a Queue block

Timeout or renege (leaving the queue after a certain time)

There are cases in which entities are only willing to stay for a limited time in the queue and will leave if they cannot exit the queue after reaching their time limit. To activate this timeout mechanism in a **Queue**, you should check the **Enable exit on timeout** checkbox and then set the timeout threshold (Figure C-44). The value does not need to be a fixed value and could be a distribution as well. In the latter case, each entity will have its own timeout threshold but each value are picked from the specified distribution.

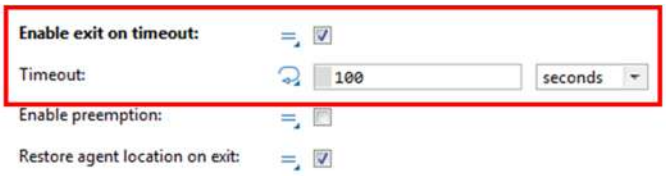


Figure C-44: Enabling the timeout mechanism in a queue

As shown in Figure C-43, **Seize** block have an "*outTimeout*" port. If the **Enable exit on timeout** checkbox of the **Queue** block is checked and an entity reaches its timeout limit, it will leave the **Queue** and enters the flowchart that is connected to the "*outTimeout*" port. In the example shown in Figure C-43, the timed-out entities go straight to a **Sink** (**timeout_sink**). However, this can instead be made into a full-fledged flowchart.

TimeMeasureEnd

As mentioned in the previous section, **TimeMeasureEnd** blocks have two embedded datasets to collect traverse times: a regular dataset and a histogram data (distribution). You can use access those inner datasets by the following codes:

- `<timeMeasureEnd_name>.dataset` : returns a **Data Set** (Where the X-axis is an entity's index and the Y-axis is the traverse time) that you can assign to a **Plot** object
- `<timeMeasureEnd_name>.distribution` : return a **Histogram Data** that you can assign to a **Histogram** object

However, if you right-click a **TimeMeasureEnd** block, you will find an option called **Create Chart** that will automatically build a distribution (**Histogram**) or dataset (**Plot**).

MoveTo

The **MoveTo** block is a versatile block that lets you move entities or **Moving** resource units from their current location to a specified destination. In process-centric models that have a network specifying possible routes between different nodes, AnyLogic automatically finds the shortest route from the current location of the entity (or resource unit) and moves it along that path. Assuming you don't change the speed of moving entity (or resource unit) in the middle of its movement, the time that the entity (or resource unit) will spend in this block is deterministic and is calculated by dividing the length of the chosen route by the speed of the entity (or resource unit). If you want the movement to have a random trip time, you must change the **Movement is defined by** field to **Trip time** and then specify a distribution. If the entity has seized units attached to it, the entity will be the controlling object and its speed governs their collective speed.

One important aspect of the **MoveTo** block is that it has infinite capacity (like a **Delay** block with maximum capacity). This means you cannot put a limit on the number of entities (or resource units) that concurrently move with the help of a single **MoveTo** block. This behavior means that this block will never block it's "*in*" port which may have some undesired effects in a pull system. If your model logic requires a limited number of concurrent movements, one option is to wrap the **MoveTo** block within a **Seize** and **Release** blocks and control the number of concurrent movements with the capacity of the associate **ResourcePool**. The other option would be to use a coupled pair of **RestrictedAreaStart** and **RestrictedAreaEnd**.

Different types of destination in a MoveTo block

MoveTo block has many different options for setting the destination of the movement:

- **Network/GIS Node (with or without attractors):** A Node (Point, Rectangular, or Polygonal) object in the environment will be set as the destination of the movement. If the selected node has attractors, they specify the exact destination after the entity or moving unit enters the node. If the model's environment is a GIS map and entities will move toward the specified GIS point or GIS region.
- **Attractor (inside a Node):** Destination of the movement is set to a *specific* attractor inside a Rectangular or Polygonal Node. This is in contrast to the previous option in which assignment to attractors are automatic.
- **Seized resource unit:** Sets the destination to a resource unit that was previously seized by a predecessor block. Since the entity might have seized units from multiple resource pools, a specific pool in the **Resource** drop-down menu should be selected.
- **Home of seized unit:** The destination will be the home location of a previously seized unit. Since the entity might have seized units from multiple resource pools, you must select a specific pool in the **Resource** drop-down menu.
- **Agent/unit:** The destination is set to a specified agent (entity or resource unit).
- **x, y, z:** The destination will be at the specified coordinates.
- **Node + (x,y,z)**: The destination is set to the specified node and then moves to the point with the specified X, Y, Z coordinates (absolute).
- **latitude, longitude (in GIS Map)**: On a GIS map, the destination will be the specified latitude and longitude.
- **Geographical place name (in GIS Map)**: On a GIS map, the destination will be a particular location on the GIS map. AnyLogic puts the entity on the first result of a map search for the name provided in this field.
- **[res.] Agent which possesses me**: This option is only available in preparation and wrap up branches and intended for the moving resource unit to move to the entity that seized it (in the primary branch).
- **[res.] Other unit seized by my agent**: This option only works in preparation and wrap up branches when the Seize block has seized multiple units from multiple resource pools. You also must select the **specific resource** option in the ResourceTaskStart block and assign a specific pool to the preparation branch. Then in a MoveTo block that is in the same preparation branch, you can select this option and select one of the other pools in the **Resources** drop-down menu.
- **[res.] Other unit's home**: This is much like the previous option; the only difference is the unit in the preparation branch (the selected option for the **specific resource** field in the ResourceTaskStart block) will move toward the "home" location of the other unit that is inside a pool selected in the **Resources** drop-down menu.

For a comprehensive discussion about moving entities and units, please refer to the **BUILDING ANIMATION FOR PROCESS CENTRIC MODELS** section in Chapter 3.

Setting the movement to be based on stating lines

As mentioned, entities or resource units follow the paths in a network (or GIS routes). However, if you select the **Straight movement** checkbox, they ignore the paths and move to the destination in a straight line.

Defining the movement based on trip time

By default, the **MoveTo** block calculates the time needed for the movement from the speed of entity (or resource unit) and the distance to the specified destination. However, there are scenarios in which we know the trip time but not the require speed for that trip time. In these cases, we can choose the movement to be defined by **Trip time** and then input the trip time in the field. This option also gives you the capability to set a random value to the trip time and therefore the time it takes for the movement of different entities won't be the same.

Modify the speed of movement

By default, each entity or resource unit has a default speed that is set at the **Source** for entities and **ResourcePool** for resource units. It can be modified with this option.

PMLSettings

PMLSettings is an optional block that defines some global settings of all the PML blocks inside the same graphical editor as the **PMLSetting**. Each agent type (or model building environment) can only have one **PMLSettings** block.

Changing the default offset between the entity and its attached units

When an entity and its seized units are attached together (via an **Attach** block or the attach option of **Seize** block) AnyLogic sets a default offset between their animations (as they move together after the attachment). You can set a different value for that offset (in pixel) from the **Offset for attached units** field of this block.