# A Guided Tour of the Systemic Modeling Language Σ

Daniel Krob and Antoine Rauzy

February 22, 2022

**Abstract**

Σ is both a language and a method for describing and studying the dynamics of complex technical and socio-technical systems and their environment. It makes it possible to implement computer simulations, to assess key performance indicators by means of these simulations, to play "what-if" scenarios and to apply optimization techniques. In a word, the framework we propose here supports the design of systemic digital twins of complex technical systems.

This article aims both at providing a guided tour of the Σ modeling framework and at illustrating its use by means of examples.

## 1 Introduction

Our world runs on increasingly complex technical systems. Engineers face a critical challenge in designing, managing, and optimizing these systems. One of the key issues is that traditional development methods, based on local optimization and silo-ed engineering disciplines do not suffice anymore (de Weck, Roos, and Magee 2011). One needs a holistic perspective on systems and their environment, encompassing technical, organizational, economical, environmental and regulatory opportunities and constraints. Systems engineering aims at providing concepts, methods and tools to support such an approach (Walden et al. 2015).

To tackle the complexity of systems, engineers more and more on computer models and simulations. By designing these digital twins of the systems, they pursue two main objectives: first, to better understand the systems and to ensure that stakeholders share a common understanding of the problems at stake; second, to assess key performance indicators without having to perform physical experiments, which would be too costly, or simply impossible.

Models are already pervasive in most of the engineering disciplines like mechanical, electrical, or reliability engineering. As of today, their introduction into systems engineering is still an on going process and the subject of active researches and developments. Modeling technologies to be applied are still debated. One of the main difficulties is to capture the key features of the system under study while staying at the suitable level of abstraction. Another difficulty is to integrate in the models the heterogeneous characteristics of systems.

The Σ modeling framework aims at providing a generic, mathematically sound and computationally efficient, solution to these difficulties. It relies on two pillars. First, one describes the architecture of the system under study, i.e. the system is decomposed into subsystems. These subsystems can be themselves further decomposed until the suitable granularity is reached. The state of each subsystem is described by means of discrete (symbolic) and continuous variables. Second, one describes activities performed by subsystems. Activities are guarded, i.e. they are performed when a certain condition on the state of the system is satisfied. They take time. This time may be deterministic or stochastic. Finally, they modify twice the state of the system. First at their beginning, to book the resources they need. Second at their completion, to release these resources and to describe their effect on the state of the system. Activities can not only modify the values of variables, but also create, move and delete components.

The Σ modeling framework enters thus into the wide category of discrete event systems (Cassandras and Lafortune 2008). As a matter of facts, it embeds a good deal of ideas and algorithms developed for the AltaRica 3.0 modeling language (Batteux, Prosvirnova, and Rauzy 2019). The Σ language belongs to the S2ML+X family (Batteux, Prosvirnova, and Rauzy 2018; Rauzy and Haskins 2019), i.e. it results from the combination of mathematical framework, here hierarchical actor networks (the X) with a versatile set of object- and prototype-oriented primitives to structure models (S2ML). The Σ modeling framework is agnostic, i.e. not dedicated to any particular application domain. Moreover, conversely to most of discrete event systems Σ makes it possible to describe seamlessly the dynamic evolution of the architecture of the system.

Once a Σ model designed, it is possible to perform Monte-Carlo simulations on this model and thereby to assess key performance indicators. Beyond, it is possible to play various "what-if" scenarios and to apply optimization techniques so to improve the performance of the system. The Σ modeling framework provides thus an essential brick in the construction of systemic digital twins of complex digital systems.

This article aims at providing a guided tour of the Σ modeling framework and at illustrating its constructs by means of examples.

The remainder of this article is organized as follows. Section 2 discusses related works. Section 3 introduces the Σ modeling framework. Section 4 presents how uncertainties and external data are handled in Σ. Section 5 shows how object-oriented constructs of S2ML can be used to represent complex architectures. Section 6 presents Σ constructs to handle deformable systems. Finally, Section 7 concludes this article.

## 2 Related Works

As pointed out in the introduction of this note, the design of modeling frameworks for model-based systems engineering is still an active field of research, development and experiments. It is probably too early to design a definitive taxonomy of the proposed modeling frameworks. We can however distinguish two main approaches.

The first approach consists in using more or less standardized graphical notations. SysML (Friedenthal, Moore, and Steiner 2011; Holt, Perry, and Brownsword 2016) is the paradigmatic example of such notations, but we can also cite OPM (Dori 2016), Arcadia/Capella (Voirin 2017) and to some extent BPMN (White and Miers 2008). Such notations are extremely useful to support the discussion among stakeholders. However, they lack a clear mathematical semantics. Consequently, they are ineffective when it turns to simulate systems or to calculate performance indicators. More precisely, to make it possible to perform simulations using such notations, one has to turn them into fully mathematically specified languages. The experience shows that this is quite challenging as they have not been designed for this purpose in the first place. As a result, these formalization tend to be tool dependent and to lead eventually to poor implementations of the second approach.

The second, and in some sense opposite, approach consists in using well established mathematical frameworks, and to represent the dynamics of systems within these frameworks.

Systems of differential equations are indeed the first candidate framework. They are extensively used in physics and many other scientific disciplines. Modeling languages and environments such as Matlab/Simulink (Klee and Allen 2011) and Modelica (Fritzson 2015) are very well suited to build up complex models by assembling elementary, reusable modeling bricks. Coming from a very different community, tools that support systems dynamics (Sterman 2000), e.g. Vensim (Garcia 2018), rely also on differential equations to represent the dynamics of systems.

If differential equations present numerous advantages, they have also severe drawbacks regarding the objective of describing the dynamics of complex technical systems from a holistic perspective. First, it is often quite difficult to describe the behavior of systems by means of differential equations at the level of abstraction required by systems engineering. Phenomena that one wants typically to capture, such as changes of modes of operation, changes of state of components, or consequences of actions, are better described by means of discrete transitions (Rauzy and Haskins 2019). Aside continuous variables, the description of systems involves naturally a large proportion of symbolic ones. Second, change rates of the different components of a system are often very different one another. Reducing all these rates as to their "least common divisor" is not only difficult from a modeling point of view, but also inefficient when it comes to simulation. Third, systems of differential equations representing the behavior of complex technical systems tend to be numerically very unstable. Tuning parameters of models and simulators to cope with this inherent instability proves to be challenging.

Discrete event simulation (Cassandras and Lafortune 2008) is an alternative to differential equations. In this approach, the evolution of the state of the system is represented by discrete jumps (events). The time elapsed between two events may vary and can be either deterministic or stochastic.

Discrete event simulation is a general principle. It can be implemented in many different ways. A first question is whether one uses *ad-hoc* programs or modeling environments supporting a modeling language. *Ad-hoc* programs offers a great flexibility as the underlying modeling language is the programming language in which they are written (C, C++, Java, Python...). This approach has however the drawback that the analyst who designs the model must be at the same time an advanced software developer. Moreover, it makes it difficult to benefit from optimizations developed for one project in another project. The authors believe that the development of *ad-hoc* programs is not sustainable for a large scale industrial deployment. The alternative approach consists in designing a modeling language and a set of assessment tools, including indeed a Monte-Carlo simulator, applying to models written in this language. This approach makes it possible to separate the concerns: software engineers design the language and develop efficient assessment tools while system analysts focus on the design of models. This means that the modeling language and consequently the expressive power at hand is fixed once for all. One makes a big win in efficiency at the cost of a loss in flexibility.

Any attempt to list all modeling languages proposed to support discrete event simulation would unavoidably fail to be complete. According to (Rauzy and Haskins 2019), we can nevertheless split them into two subgroups: those that define the architecture of the system once for all, and those making possible to modify this architecture throughout a simulation. The latter provide the analyst with a higher expressive power, but models are harder to design, to validate and to maintain and assessment tools can be much less efficient than with the former. This is the reason why the first group is, as of today, by far predominant.

Even if we restrict our attention to stochastic models, we can cite, among many other formalisms that belong to this first group, stochastic Petri nets (Ajmone-Marsan et al. 1994; Signoret and Leroy 2021), languages such as Figaro (Bouissou et al. 1991) and AltaRica (Batteux, Prosvirnova, and Rauzy 2019). Σ borrows its timing model to these formalisms, but belongs to the second group. The timing model makes these formalisms very different from input languages of model checkers such as PRISM (Kwiatkowska, Norman, and Parker 2011) and SAML (Güdemann and Ortmeier 2010) and stochastic automaton networks (Plateau and Stewart 1997) which are restricted to exponential distributions (Markov chains). Other timed model checkers, such as Uppaal (Larsen, Pettersson, and Yi 1997), provides more flexibility in the definition of the duration of the transitions, but still not as much as the former formalisms. Moreover, model checkers are mostly designed to check reachability properties, not to the assess key performance indicators. They are rather academic proof of concepts than industrially deployed tools.

Formalisms belonging to the second group are much less numerous. Moreover, they tend not to separate clearly the model and the simulation algorithm, as for instance in agent-based languages such as NetLogo (Wilensky and Rand 2015). We can however cite colored Petri nets (Jensen 2014) as an attempt lift up the expressive power of the modeling language while keeping models clearly separated from assessment algorithms.

The challenge was to design a powerful modeling language, dedicated to description of the dynamics of complex technical systems and their environment, while not sacrificing too much the efficiency of assessment algorithms. The key enabler has been the introduction the notion of activity. In a Σ model, actors (subsystems) perform activities conditionally to the state of the system. These activities take deterministic or stochastic time. The difference with other mathematical models supporting discrete event simulation, for instance guarded transition systems (Rauzy 2008) that are at the core of AltaRica, is that Σ activities execute instructions both at the beginning and at their completion. As we shall see, this, apparently minor change, is actually a *sine qua none* condition to handle dynamic changes in the architecture of systems, and therefore to make it possible to represent the dynamics of systems of systems (Maier 1998).

To conclude this section, it is worth insisting that although Σ actors in some sense react to environmental conditions (by performing activities), Σ is very different from reactive languages such as Lustre (Halbwachs 1993). First, Σ is a modeling language and not a programming language. Σ models represent both the systems and their environment. Once designed, they can be assessed by means of various tools, notably by means of stochastic simulation. Second, Σ activities have an explicitly specified duration that is decorrelated from the actual computation time taken by computer simulations.

## 3 Getting Started

### 3.1 The Σ Ontology

As explained in the introduction, in the Σ approach, the system under study is decomposed into subsystems, which can be themselves further decomposed. In a word, each and every part of system is viewed as a system. This translates directly into the Σ language that consists essentially in three types of components:

– Systems, that are containers for components;
– Variables, that are value holders and that are used to describe the state of the system;
– Activities, that are mechanisms by which the system is modified.

As an illustration, consider a small production company, the `Producer`. The `Producer` produces products that are consumed by the consumer `Consumer`. More exactly, the `Consumer` orders regularly a quantity of products. The `Producer` produces these products and delivers them to the `Consumer` that can then consumes them. To produce products, the `Producer` needs raw materials. The raw materials are produced by then `Supplier` which then delivers them to the `Producer`.

Figure 1 shows the hierarchical decomposition of our system of interest. It consists of the `World` and three subsystems: `Supplier`, `Producer` and `Consumer`.

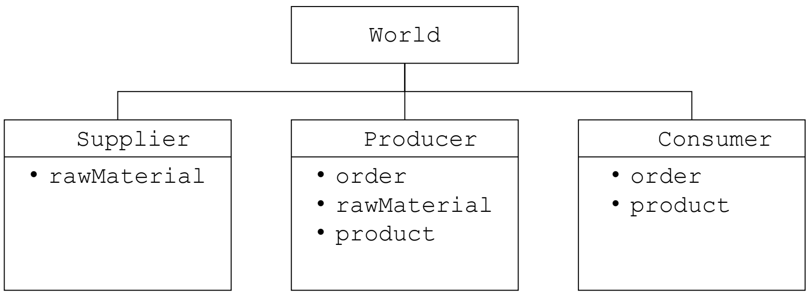

Figure 1: Hierarchical decomposition of the “world” of the `Producer`

Each system owns stocks, i.e. certain quantities of raw materials, products and orders. Figure 1 shows these stocks that concretely encoded as numerical variables, integers or floating point numbers.

The state of the system at a given time is thus described by the hierarchy on the one hand, and by the values of the stocks on the other hand. For the sake of simplicity, we shall first consider that the hierarchy of our system stay the same throughout its life-cycle. Its state is thus described essentially by the values of stocks.

Now, we must describe how the system changes of state. In the Σ framework, this is achieved by describing activities performed by each actor. Namely:

– The `Supplier` renews periodically its stock of raw materials.

- The `Producer` gets periodically some raw materials from the `Supplier`.
- The `Consumer` orders periodically products to the `Producer`.
- The `Producer` produces periodically products using its stock of raw materials.
- The `Producer` delivers periodically products to the `Consumer` according to its stock of demands and to its stock products.
- Finally, the `Consumer` consumes periodically part of its stock of products.

All these activities take place under certain conditions, have a certain duration, and have a certain effect on stocks of one or more actors, i.e. subsystem.

## 3.2 Systems and Variables

According to the ontology described in the previous section, a Σ model is thus essentially a hierarchy of systems where each system may hold some variables and perform some activities.

Figure 2 shows the skeleton of a Σ model that implements this hierarchy. We assumed here that all stocks are represented by means of integers.

```
system World
  system Supplier
    int rawMaterial(init = 0);
  end
  system Producer
    int order(init = 0);
    int rawMaterial(init = 0);
    int product(init = 0);
  end
  system Consumer
    int order(init = 0);
    int product(init = 0);
  end
end
```

Figure 2: Skeleton of the model representing system breakdown structure of the producer world

Keywords are printed out in bold and blue.

The system `World` declares three sub-systems: `Supplier`, `Producer` and `Consumer`. In turn, the subsystem `Supplier` declares a variable `rawMaterial`, the subsystem `Producer` declares three variables `order`, `rawMaterial` and `product`, and finally the subsystem `Consumer` declares two variables `order` and `product`. Initially, all the variables take the value 0, which is indicated by the attribute `init`.

## 3.3 Redeclarations

In the above example, the hierarchy of systems is shallow: two levels, three if we count variables. Models of complex systems may involve much deeper hierarchies. It would be tedious to nest descriptions of inner levels into a single block, not to speak about the source of errors of having the closing `end` of a system description coming thousands of lines below the opening `system`.

Redeclarations are a means to avoid nested descriptions. A Σ model can actually be seen as a script that creates the actual model. The model *as assessed* is obtained by automated transformations from the model *as design*. Indeed, these transformations preserve the semantics.

In our example, the system `World` could be declared first with its three subsystems, but letting the declarations of the latter incomplete. The subsystem `Supplier`, `Producer` and `Consumer` can be redeclared latter to complete their description.

Figure 3 shows the possible code applying this idea.

```
system World
  system Supplier ... end
  system Producer ... end
  system Consumer ... end
end

system World.Supplier
  int rawMaterial(init = 0);
end

system World.Producer
  int order(init = 0);
  int rawMaterial(init = 0);
  int product(init = 0);
end

system World.Consumer
  int product(init = 0);
end
```

Figure 3: Skeleton of the model representing system breakdown structure of the producer world, with redeclarations

Ellipses "`...`" have no specific meaning. Rather they are an (optional) indication that the system will be further developed.

## 3.4 Activities

As explained above, in Σ, the dynamics of systems is described via the notion of activity. An activity is characterized by the following elements.

- A triggering condition, also called a guard, that is an instruction (a calculation) returning true when the activity must be started and false otherwise. The activity is started as soon as its triggering condition is satisfied.
- An action at start, i.e. an instruction that is executed when the activity starts. This instruction books the resources required by the activity. It may also be used to ensure that the same activity cannot be started again before the activity is completed.
- An action at completion, i.e. an instruction that is executed when the activity is completed. This action changes the state of the system to reflect the effects of the activity.
- A duration calculation, i.e. an instruction that is executed when the activity starts to determine how long it will take to complete the activity.

We shall see later that activities can be also interrupted.

As a first illustration, consider the activity of the `Supplier` that consists in renewing its stock of raw material, i.e. incrementing it by a certain amount, say 100. For now, we shall consider that variables have no unit. Assume that this activity is performed if the stock is not bigger than 1000 (as the supplier does not want to accumulate unsold materials), Assume moreover that this activity takes 30 days (or time units) to be completed.

Figure 4 shows a possible code to implement the `RenewRawMaterialStock` activity of the `Supplier`.

This code implements the four elements of the activity `RenewRawMaterialStock`, introduced respectively by the keywords `trigger`, `start`, `completion` and `duration`.

```
system World.Supplier
  int rawMaterial(init = 0);
  bool renewing(init = false);
end

activity World.Supplier.RenewRawMaterialStock
  trigger:
    return rawMaterial<=1000 and not renewing;
  start:
    renewing = true;
  completion: {
    renewing = false;
    rawMaterial += 100;
    }
  duration:
    return 30;
end
```

Figure 4: Code for the `RenewRawMaterialStock` activity of the `Supplier`

The triggering condition is when the stock of raw materials goes below 1000. There is however a second condition: that the `Supplier` is not already in the process of renewing its stock, hence the introduction of the Boolean variable `renewing`. Initially, this variable takes the value `false`. It takes the value `true` while the activity is on going and takes back the value `false` once the latter is completed.

When the triggering condition of an activity is true, one says that this activity is enabled. Initially, the stock of raw material of the `Supplier` is null and the variable `renewing` is false. Consequently, the activity `RenewRawMaterialStock` is enabled.

The instruction at completion is thus a block (a sequence) of two instructions: an instruction that sets back `renewing` to `false` and an instruction that increments the stock `rawMaterial` by 100. Finally, the duration returns simply 30, as expected.

As we shall see, in Σ, instructions can implement more or less complex calculation procedures, involve the call of functions and the access to externally stored data.

As a second illustration, consider the activity `RenewRawMaterialStock` but this time of the `Producer`. This activity is similar to the one of the `Supplier` except that:

– The decision to renew the stock of raw material is taken based not only on the current stock but also on stock of orders.
– Moreover, the `Producer` can renew its stock of raw materials only if the `Supplier` is able to deliver these materials.
– The quantity of raw material acquired by the `Producer` can be constant or depending on the needs. Here, we shall assume for the sake of simplicity that the `Producer` buys raw materials by chunk of 20 units.
– We shall assume that it takes 10 days for the `Producer` to buy a chunk of raw material and to be delivered.

Figure 5 shows a possible code to implement the `RenewRawMaterialStock` activity of the `Producer`.

This code is slightly more complex than the previous one.

First, the calculation of the triggering condition involves the intermediate variable `requiredQuantity` (of raw material). This variable is local. It exists only during the calculation of the triggering condition. Its value is calculated based on the stock of order and a security margin the `Producer` takes, here 15.

More importantly, both the calculation of the value of the triggering condition and of the instruction at start involve the stock of raw material of the `Supplier`, i.e. the variable `rawMaterial` of the latter.

```
system World.Producer
  int order(init = 0);
  int rawMaterial(init = 0);
  int product(init = 0);
  bool renewing(init = false);
end

activity World.Producer.RenewRawMaterialStock
  trigger: {
    int requiredQuantity;
    requiredQuantity = order*5 + 15;
    return main.Supplier.rawMaterial>=20
      and rawMaterial<=requiredQuantity
      and not renewing;
    }
  start: {
    renewing = true;
    main.Supplier.rawMaterial -= 20;
    }
  completion: {
    renewing = false;
    rawMaterial += 20;
    }
  duration:
    return 10;
end
```

Figure 5: Code for the `RenewRawMaterialStock` activity of the `Producer`

Consequently, one needs a means to refer to this variable within the (activity `RenewRawMaterialStock`) the `Producer`.

Σ provides the notion of path to do so. This notion comes actually from S2ML (Batteux, Prosvirnova, and Rauzy 2018). A path is an absolute or a relative reference to a modeling element.

In the body of the system `Supplier`, the path `rawMaterial` refers to the variable `rawMaterial` of the `Supplier`.

To refer to this variable in the system `World`, one can use the dot notation. Hence, in the system `World`, `Supplier.rawMaterial` refers to the variable `rawMaterial` of the `Supplier`. Similarly, `Producer.rawMaterial` refers to the variable `rawMaterial` of the `Producer`.

Now, to refer to the variable `rawMaterial` of the `Supplier` in the system `Producer`, one has two choices:

– To use an absolute path, here `main.Supplier.rawMaterial`. The keyword `main` refers to the top most system of the hierarchy, here `World`. Consequently, `main.Supplier.rawMaterial` refers to the variable `rawMaterial` of the subsystem `Supplier` of the system `World`.

– To use a relative path, here `owner.Supplier.rawMaterial`. The keyword `main` refers to the parent system of the current system. Consequently, `owner.Supplier.rawMaterial` refers to the variable `rawMaterial` of the subsystem `Supplier` of the parent system of the system `Producer`, i.e. the system `World`.

The same kind of description can be provided from each activity involved in the description of our use case.

## 3.5 Executions

The semantics of a Σ model is defined as the set of all possible executions of that model, starting from the initial state.

In our example, in the initial states, all the stocks are empty. Two activities are enabled: the activity `RenewRawMaterialStock` of the `Supplier` and the activity `OrderProduct` of the `Consumer`. Assume the latter takes 5 days and consists of an order of 10 products at a time. Assume moreover that the `Consumer` orders products until it has ordered 20 products.

These two activities start at time $t = 0$. Then, at time $t = 5$, the second one is completed. As a result, the variables `order` of both the `Producer` and the `Consumer` are increased by 10.

As the stock of orders of the `Consumer` is still less than 20. Consequently, the `Consumer` launches again an order. At time $t = 5+5 = 10$, this activity is completed. Now, both variables `order` of both the `Producer` and the `Consumer` have the value 20. The activity `OrderProduct` of the `Consumer` is thus no longer enabled.

As the completion activity `RenewRawMaterialStock` of the `Supplier` is scheduled at time 30, nothing happen til this date. A time $t = 30$, the activity is completed, the stock of raw materials of the `Supplier` (which was empty) is increased by 100. This makes it possible for the producer to launch its own activity `RenewRawMaterialStock`.

At time $t = 30+10 = 40$, this activity is completed. 10 units of raw materials are transferred from the stock of the `Supplier` to the stock of the `Producer`. The latter can thus start the production as it is not currently producing and it has a non empty stocks of orders and of raw materials.

And so on.

As both results of activities and their durations are all deterministic, there is in our example only one possible execution. When stochastic results or durations are introduced, there are usually infinitely many possible executions, as we shall see in the Section 4.

## 3.6 Terminology and Additional Syntactic Constructs

### 3.6.1 Terminology

Σ is an object-oriented language. It is thus worth to use the terminology of object-oriented theory to refer to its constructs.

With that respect, Σ systems are prototypes, i.e. objects with a unique occurrence in the model. They are also containers for declarations of variables, activities and other systems. When the system $S$ contains the variable $V$, the activity $A$ or the subsystem $T$, one says that $S$ composes $V$, $A$ or $T$. Note that, Σ makes it possible to declare $T$ in the system $S$ and then move it to another part of the hierarchy. It makes also possible to create and delete subsystem. Consequently, the composition relation is dynamic.

Systems declaring activities are called actors. Activities can be seen, at least to some extent, as the methods of the system that declares them.

### 3.6.2 Comments

As in any programming or modeling language, it is possible to introduce comments in Σ models. Basically, Σ are the same as those of C, C++, and ... AltaRica 3.0.

There are two types of comments.

Single line comments start with a double slash "`//`" and spread until the end of the line. E.g.

```
duration:
  return 10; // To be checked
```

Multi-lines comments start with "`/*`" and finish with "`*/`". E.g.

```
/*
 * Producer/Consumer model
 */
```

### 3.6.3 Units

The experience shows that it is often very convenient to indicate the units of stocks. The current version of the Σ makes it possible to do so via the attribute `unit`. E.g.

```
  float rawMaterial(init = 0, unit = "t");
```

The convention is to use the MKS system. However, in their current version, Σ assessment tools perform no verification. Nevertheless, it is recommended to follow the convention, for the sake of upward compatibility.

## 4 Handling Uncertainties and External Data

### 4.1 Stochastic Behaviors

So far, we have considered only deterministic executions. The dynamic of technical and socio-technical systems is however subject to uncertainties that must be taken account into models. If these uncertainties were pure random variations, obeying no known statistical law, reflecting them in the models would be hopeless. Fortunately, many uncertainties do obey statistical laws, that can be characterized either by means experience feedback, physical models or expert judgment. We can then speak about stochastic behaviors, i.e. behaviors that are essentially non-deterministic but whose randomness follows known probability distributions.

In Σ, such stochastic behaviors can be introduced in two ways: via durations and via results of activities.

As an illustration, consider again our producer/consumer example. The consumption of goods by the `Consumer` may be subject to fluctuations. A first way to reflect these fluctuations is assume that the duration of the consumption of a product is stochastic. It may for instance obey a triangular distribution with a lower bound of 1 day, an upper bound of 5 days and a mode at 2 days. Then we can use the corresponding built-in distribution, as illustrated in Figure 6.

```
activity World.Consumer.ConsumeProduct
  trigger:
    return product>=1000 and not consuming;
  start:
    consuming = true;
  completion: {
    consuming = false;
    product -= 1;
    }
  duration:
    return triangularDeviate(1, 5, 2);
end
```

Figure 6: Code for the `ConsumeProduct` activity of the `Consumer` with a stochastic duration

We may focus on the contrary on the quantity of products that the consumer consumes every day. This would probably require to consider this quantity as a real number rather than as an integer. We may imagine for instance that this quantity is uniformly distributed between 1 and 2.5 products. Figure 7 shows the corresponding Σ code.

```
activity World.Consumer.ConsumeProduct
  trigger:
    return product>=1000 and not consuming;
  start:
    consuming = true;
  completion: {
    consuming = false;
    product -= uniformDeviate(1, 2.5);
    }
  duration:
    return 1;
end
```

Figure 7: Code for the `ConsumeProduct` activity of the `Consumer` with a stochastic result

## 4.2 Built-In Parametric Distributions

As illustrated in the previous section, Σ provides a panoply of built-in parametric cumulative probability distributions: exponential, Weibull, normal, lognormal, uniform, triangular...

These distributions can be used either to characterize the duration of activities, or their results.

Their key characteristic is that they encode, in a way of another, monotone functions from $[0,1]$ into $\mathbb{R}$. By drawing a number uniformly at random between 0 and 1, one can get a real value, typically a delay, according the distribution. This process is illustrated in Figure 8 for the Weibull distribution.

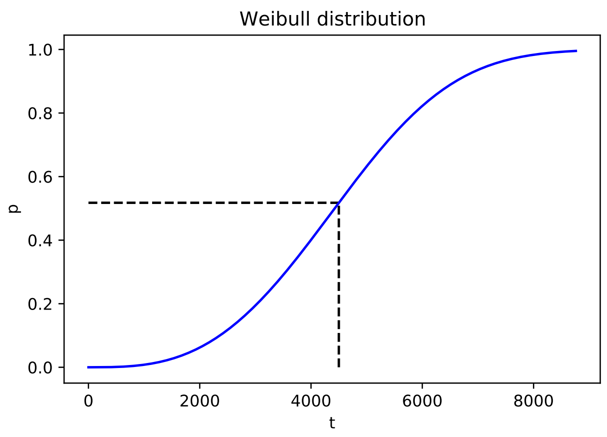

Figure 8: Weibull distribution and its use to generate delays at random

If the number $z = 0.5176$ is drawn at random and the Weibull deviate is applied, the the return value is close to 4500.

## 4.3 Empirical Distributions

It is not always possible to match experience feedback data with available parametric distributions. In that case, one case use instead empirical distributions, i.e. eventually lists of points in between which the values are interpolated (typically using a linear interpolation).

This process is illustrated in Figure 9 for the empirical distribution characterized by the points $(0,0)$, $(1000,0.2)$, $(2500,0.3)$, $(3500,0.4)$, $(5000,0.6)$, $(7000,0.7)$ and $(8760,1.0)$.

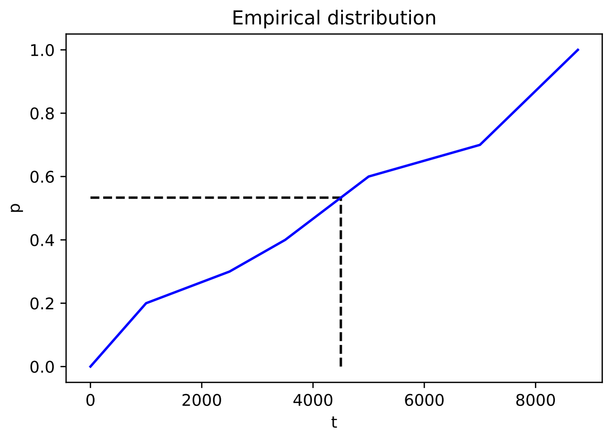

Figure 9: An empirical distribution and its use to generate delays at random

If the number $z = 0.5333$ is drawn at random and the empirical distribution is applied, the the return value is close to 4500.

Empirical distributions are typically obtained from Kaplan-Meier estimators (Kaplan and Meier 1958).

## 4.4 External Data

In Σ, empirical distributions (and more generally any type of relational data) are stored externally into text files at the TSV format. The TSV format, TSV stands for tabulations separated values, consists of lines of values separated with tabulations. A file at the TSV format can thus be seen as a matrix. Lines encode rows of the matrix. Tabulations separates its columns. In the above example, the distribution would be described by means of 7 rows and 2 columns.

Once this distribution saved into a the file `"Data/consumptionDelay.tsv"`, it can be called within the Σ model in the following way.

```
duration:
  return empiricalDistribution("Data/consumptionDelay.tsv", linear);
```

The first argument of the built-in `empiricalDistribution` is the name of the TSV file, the second one the interpolation to be performed.

Storing external data into TSV files makes it possible to extract information from the latter in several ways. For instance, one can calculate a delay from a randomly generated number, but one can reciprocally calculate a probability of occurrence at a given date.

## 4.5 Stochastic Simulations

### 4.5.1 Principle

Once stochastic elements introduced in models, their executions are not deterministic anymore. Monte-Carlo simulation then becomes the key assessment tool. The basic idea is fairly simple: performs a large number of executions and make statistics on various indicators in order to get a clear picture of the possible evolutions of the system described by the model.

Monte-Carlo simulation has advantages and drawbacks. The two main drawbacks are that it can give only approximated results. The larger the number of executions, the more accurate the results. The second one is that it is resource consuming: to get accurate enough results, one may need to perform a very large

```
system World.Producer
  ...
  int rawMaterial(init = 0);
  ...
  observer levelRawMaterial = rawMaterial;
  ...
end
```

Figure 10: Declaration of a numerical observer

number of executions (up several millions), which has indeed a cost in terms of computational resources. The main advantages on Monte-Carlo simulation mirror in some sense its drawbacks: first, it is a versatile tool, applicable on any type of stochastic behavioral models. Second, it is an anytime algorithm: one can perform a first series of executions, check the result, perform a second series and so on.

A key question about Monte-Carlo simulation, rarely discussed in the engineering literature, is how to define the indicators on which statistics are performed. On the one hand, these indicators should be sufficiently many to give a clear picture on trends of system's possible dynamics their relative likelihoods. On the other hand, accumulating statistical data is both time and space consuming. A trade-off must thus be found.

In Σ, the definition of statistical indicators is done in two steps: first, so-called observers are defined from variables, then indicators are defined from observers. We shall discuss now the whys and wherefores of this process, which is inspired what has been put in place from AltaRica 3.0 (Batteux, Prosvirnova, and Rauzy 2019).

### 4.5.2 Observers

Observers are like real valued variables, except that they are defined by means of a single assignment and they cannot be used in expressions. Expressions that define observers can numerical or Boolean. In the latter case, they are interpreted as characteristic functions, i.e. their value is 1.0 if they are true and 0.0 otherwise.

For each observer, the following quantities are continuously updated along an execution:

- Its current value;
- Its minimum, maximum and mean values since the beginning of the current execution;
- The first date at which it changed of value after the beginning of the execution;
- The last date at which it changed of value since the beginning of the execution;
- The number of times it changed of value since the beginning of the execution.

Changes of values are considered only if they last for a strictly positive time.

As an illustration, assume we want to follow the level of the stock of raw materials of the `Consumer`. Then, we can declares an observer as illustrated in Figure 10.

The observer `RawMaterial` makes it possible to follow the evolution of the stock.

Now assume that we consider that the stock should normally be over 30 units or, to put it differently that a problematic situation occurs when it is below this threshold. To known whether such situation occurred since the beginning of the execution, it suffices to look at the minimum value the observer took since then. Looking at this value is however insufficient to know how much time we spent in a dangerous state. To get this information, we can declare another observer, defined by a Boolean expression, as illustrated in Figure 11.

By looking at the maximum value of `lowLevelRawMaterial`, we know whether we encountered the problematic situation at least once. Now by multiplying the mean value of the observer by the current date, we obtain the expected sojourn time.

This example shows that observers as Σ defines them give thus a significant amount of information on

```
system World.Producer
  ...
  int rawMaterial(init = 0);
  ...
  observer lowLevelRawMaterial = rawMaterial<=30;
  ...
end
```

Figure 11: Declaration of a Boolean observer

```
system World.Producer
  indicator probabilityTooLowStock = maximum lowLevelRawMaterial(
    mean=true, standardDeviation=true);
  indicator sojournTimeTooLowStock = mean lowLevelRawMaterial(
    mean=true, standardDeviation=true, quantiles=10);
end
```

Figure 12: Declaration of an indicator

quantities of interest. The AltaRica 3.0 experience shows that all practical needs can be actually covered by such observers.

### 4.5.3 Indicators

Statistical indicators are defined from observers. Conversely to observers, that are continuously updated throughout executions, indicators are calculated at predefined dates, usually at the mission time and possibly some additional intermediate dates.

Due to memory usage and computation time considerations, it is preferable not to make statistics on all quantities calculated for each observer. Usually, only one or two of them are of actual interest. This is the reason why indicators are defined. An indicator is essentially a pair made of an observer and the quantity one want to make statistics on.

Indicators are declared in a similar way as observers. Thanks to the model-as-script principle, it is possible to declare them separately from the core of the model, as illustrated in Figure 12 (the code is assumed to come after the one of Figure 11).

By making statistics on the maximum value of the observer `lowLevelRawMaterial` from the beginning of the execution to the current date, one assesses the probability that the problematic situation occurred at least once. By making statistics on its mean value, one assesses the sojourn time in such potentially problematic situation.

The following statistics can be made on each indicator and each date at which this indicator must be calculated.

– Its mean value, standard deviation, 95% confidence interval and error factor.
– Quantiles and distributions.

The statistics to be made are indicated as shown in Figure 12.

Note that some indicators related to dates and numbers of changes of values of the observers may be applicable on a restricted subset of executions, as the observer may keep the same value throughout the whole execution.

Statistics are exported into CSV files, so that they can be easily worked out by in Spreadsheet tools such as Excel® or using scripting languages such as Python.

```
domain MachineState {STANDBY, WORKING, FAILED}

system World.Producer
  ...
  system Machine
    MachineState state(init = STANDBY);
  end
  ...
  activity ProduceProduct
    trigger:
      return order>=2 and rawMaterial>10 and Machine.state==STANDBY;
    start: {
      Machine.state = WORKING;
      rawMaterial -= 10;
      }
    completion: {
      Machine.state = STANDBY;
      order -= 2;
      product += 2;
      }
    duration:
      return 5;
    interruption:
      Machine.state==FAILED; then rawMaterial += 3;
  end
end
```

Figure 13: Code for the `ProduceProduct` of the producer with a possible interruption

## 4.6 Interruptions

Uncertainties, especially uncertainties regarding the durations, may introduce some competition among activities. As an illustration, consider again our example.

When he received orders from the `Consumer` and his stock of raw material is high enough, the `Producer` starts the production of products. We may imagine that he uses for that a machine and that this machine may fail. We have thus here two activities in competition: the production and the failure. They start simultaneously. The completion of one of them interrupts the other, or at least the occurrence of a failure interrupts the production.

Figure 13 shows the code for the activity `ProduceProduct` of the `Producer` with a possible interruption due to a failure of the `Machine`.

This code starts with a domain declaration. Domains are finite set of symbolic constants. In addition to predefined types like `int` or `bool`, variables can take their value into such domains. This is the case here for the variable `state` of the subsystem `Machine` of the `Producer`. Initially, the `Machine` is in standby.

The code declares then the activity `ProduceProduct`. To be launched this activity requires that the machine is in standby. The actions at start and completion as well as the duration are declared similarly to what we have seen so far. The novelty stands in the declaration of an interruption.

An interruption consists of two parts. First, a Boolean condition telling when the interruption takes place. Here, the activity is interrupted as soon as the `Machine` gets failed. Second, an action to be performed once the interruption has occurred. Here, 3 units of raw materials are saved. This action is introduced by the keyword `then`. It is optional.

An activity can declared more than one interruption.

## 5 Representing Complex Architectures

This section reviews S2ML constructs, as they are implemented in Σ. S2ML stands for system structure modeling language (Batteux, Prosvirnova, and Rauzy 2018; Rauzy and Haskins 2019). However, more than a modeling language *per se*, it is a versatile set of constructs used to structure models. These constructs are stemmed from both object-oriented programming (Abadi and Cardelli 1998) and prototype-oriented programming (Noble, Taivalsaari, and Moore 1999). They are already used in several modeling languages, notably S2ML+SBE (Rauzy 2020) and AltaRica 3.0 (Batteux, Prosvirnova, and Rauzy 2019), see also one of the authors' book on model-based reliability engineering (Rauzy 2022).

### 5.1 Parameters

Parameters are similar to variables, except that they are given a value once for all, at the creation of the model. They are useful to document models and, we shall see, to create generic reusable modeling components.

As an illustration, consider again the activity `RenewRawMaterialStock` of the `Supplier`. This activity involves three constants: the threshold under which the stock needs to renewed (1000), the amount of new raw material constituted at each renewal (100), and the duration (30). These three constants can be advantageously defined as parameters, as shown in Figure 14.

```
system World.Supplier
  parameter int renewalThreshold = 1000;
  parameter int renewalAmount = 100;
  parameter int renewalDuration = 30;
  int rawMaterial(init = 0);
  bool renewing(init = false);
end

activity World.Supplier.RenewRawMaterialStock
  trigger:
    return rawMaterial<=renewalThreshold and not renewing;
  start:
    renewing = true;
  completion: {
    renewing = false;
    rawMaterial += renewalAmount;
    }
  duration:
    return renewalDuration;
end
```

Figure 14: Code for the `RenewRawMaterialStock` activity of the `Supplier` with parameters

Parameters can be defined by means of expressions. However, these expressions can only involve constants and parameters (not variables). Of course, a parameter cannot depend eventually on itself.

### 5.2 Functions

As many other modeling languages, Σ provides some (limited) capability to define functions. Functions take a number of arguments and return a value.

As an illustration, assume that the quantity produced by the activity `ProduceProduct` of `Producer` requires some relatively complex calculation depending on how much products should be produced and the available stock of raw material. It is then possible to take the code of this calculation out of the declaration of `ProduceProduct` and declare it as a function, as illustrated in Figure 15.

```
function CalculateQuantity(order, rawMaterial): int
  // calculation algorithm
  return quantity
end
```

Figure 15: Declaration of a function

The function `CalculateQuantity` can then be used as in any other modeling or programming language.

## 5.3 Cloning

It is often the case that complex technical and socio-technical systems involves several identical or at least similar subsystems.

For instance, in our case study, the `Producer` may use two identical machines working in parallel to deliver the required amount of products. It would be indeed possible to duplicate the code representing the behavior of each machine. This would be however both tedious, error prone and would not reflect that the two machines are actually identical. Σ provides the cloning mechanism to solve this issue.

Figure 16 illustrates the use of this mechanism.

This code is similar to the one we already proposed in Figure 13, with two differences however.

First, we took down now the activities `ProduceProduct` and `Failure` at machine level. Note the use of relative path to refer, in the subsystem `Machine1` to variables declared in the parent system `Producer`.

Second, we created a second, identical, machine, `Machine2`, by cloning the subsystem `Machine1`. The `clones` directive duplicates `Machine1`, its variables, subsystems and activities, and gives a new name the clone, here `Machine2`. Hence the interest of declaring activities at machine level: they are automatically duplicated by the cloning.

Cloning is thus a very powerful mechanism (stemmed from prototype-oriented programming). It must however be handled with care: according to the model as script principle, the `clones` directive is applied when the declaration of the `Producer` is "executed". `Machine1` is duplicated as of the point where the `clones` directive is applied. Consequently, it is impossible to declare first `Machine1`, then to clone it into `Machine2` and eventually to declare the activities `ProduceProduct` and `Failure` of `Machine1`, because with such declaration order, `Machine1` would be cloned without its activities.

## 5.4 Classes and Instances

Cloning makes it possible to reuse modeling components within a model, as illustrated in the previous section. It is often the case however, that on-the-shelf modeling components can be declared into libraries and then used at will into different models. The object-oriented class/instance mechanism implements this idea.

As an illustration, consider again the declaration of the `Producer`. We may consider that the description of machines is worth to put into a dedicated library. We can then declare a class `Machine` and instantiate this class twice in our model, as illustrated in Figure 17.

The cloning and class/instance mechanisms produce eventually the same results. The former corresponds to top-down approach to design models, while the latter corresponds to a bottom-up one. The top-down approach is more often used in system architecture and in reliability analyses, while the bottom-up one is more often used in multi-physic simulation. The bottom-up approach involves the reuse of modeling components while the top-down approach involves the reuse of modeling patterns. As discussed

```
system World.Producer
  ...
  system Machine1
    MachineState state(init = STANDBY);

      activity ProduceProduct
        trigger:
          return owner.order>=2 and owner.rawMaterial>10 and state==STANDBY;
        start: {
          state = WORKING;
          owner.rawMaterial -= 10;
          }
        completion: {
          state = STANDBY;
          owner.order -= 2;
          owner.product += 2;
          }
        duration:
          return 5;
        interruption:
          state==FAILED; then owner.rawMaterial += 3;
      end

      activity Failure
        ...
      end
  end

  clones Machine1 as Machine2;
  ...
end
```

Figure 16: Code for the `ProduceProduct` of the producer with cloning of a machine

```
domain MachineState {STANDBY, WORKING, FAILED}

class Machine
  MachineState state(init = STANDBY);

  activity ProduceProduct
    // description of the activity (as previously)
  end

  activity Failure
    // description of the activity (as previously)
  end
end

system World.Producer
  ...
  Machine Machine1, Machine2;
  ...
end
```

Figure 17: Code for the `ProduceProduct` of the producer instantiating of the class `Machine`

```
system World.Producer
  ...
  Machine Machine1
    parameter Real failureRate = 1.23e-5;
  end
  Machine Machine2
    parameter Real failureRate = 3.45e-5;
  end
  ...
end
```

Figure 18: Instantiating the class `Machine` with redefining the value of a parameter

in already cited references (Batteux, Prosvirnova, and Rauzy 2018; Rauzy 2022; Rauzy and Haskins 2019), this corresponds to different way of building knowledge about a particular domain. This phenomena has been well explained by Hatchuel and Weill with their CK theory of innovation (Hatchuel and Weill 2009). Indeed, in practice, both a mixed approach is often used, involving both cloning and class/instance mechanisms. Hence the interest of S2ML that gathers in a unified framework constructs stemmed from prototype- and object-oriented programming.

## 5.5 Polymorphism and Inheritance

Parameters are an excellent way to make on-the-shelf, reusable, modeling components generic. In Σ, parameters can be defined with a default value and redefined when the class is instantiated. Assume for instance that the duration of the activity `Failure` of our machines obey an exponential distribution with some failure rate. It is then recommended to define this failure rate as a parameter `failureRate`. Its value can then be changed as illustrated in Figure 18.

In object-oriented theory, one speaks about polymorphism or genericity of components (Abadi and Cardelli 1998).

When designing libraries of on-the-shelf modeling components, it is often the case that however that the description of a component can be obtained by refining the definition of a more abstract component. One says then that the concrete component inherits from the abstract one. If composition can be seen a "is-part-of" relation, inheritance can be seen as "is-a" relation.

As an illustration, consider again our machines. The description of a machine can be seen as the specialization of a more abstract repairable component. This repairable component would declare two activities, `Failure` and `Repair`, and could be inherited by many other types of components than machines. In Σ this operation is performed by the `extends` directive. Figure 19 illustrates this mechanism.

When the class `Machine` is instantiated, all elements of the class `RepairableComponent` are duplicated in the instance as if they were declared in the class `Machine`. The difference with declaring an instance of the class `RepairableComponent` in the class `Machine` is that modeling elements of the former are directly elements of the latter, and not of a subsystem of the latter.

## 5.6 Aggregation

The last important S2ML construct is the notion of aggregation. In object-oriented programming language, the term aggregation is applied when an object *A* refers to an object *B* without managing this object. In other words, *B* is created and deleted outside *A*, conversely to what happens with the composition. *A* "uses" *B*, but *B* is not "a-part-of" *A*.

This reference mechanism exists also in Σ, as we shall see in the next section. In the context of modeling languages, aggregation can be seen as an aliasing mechanism: rather than to refer to elements

```
class RepairableComponent
  ...
  activity Failure
    // Description of the activity
  end

  activity Repair
    // Description of the activity
  end
  ...
end

class Machine
  extends RepairableComponent;
  ...
end
```

Figure 19: The class `Machine` inherits from the class `RepairableComponent`

of *B* via potentially long relative or absolute paths, one aggregates the object *B* into the object *A* so to more direct access to its elements.

As an illustration, assume that some relevant information (like price, availability...) related to raw materials is available the system `RawMaterials` itself located in the subsystem the `Supplier`. It would be tedious to write long paths such as `main.Supplier.RawMaterial.price`. An alternative consists in using aggregation, via the directive `embeds`, as illustrated in Figure 20.

The identifier `RM` serves as an alias for `main.Supplier.RawMaterial` in the `Producer`.

## 6 Handling Deformable Systems

### 6.1 Rational

Formally speaking, a deformable system is a system whose architecture changes during its mission. Systems of systems (Maier 1998) are typical examples deformable systems, but they are not the only ones. In more classical systems, it is sometimes of interest to represent mobile components, i.e. subsystems that are dynamically created, moved around in the hierarchy and eventually deleted.

As an illustration, in our producer/consumer example, we may be interested in following individual orders and products. The process could be typically as follows. The `Consumer` creates an order, then places it to the `Producer`. The `Producer` takes that order, produces the corresponding product, deletes the order and eventually delivers the product to the `Consumer`. Finally, the `Consumer` consumes the product, which is thus deleted.

The difficulty here is that several orders and products may circulate simultaneously between the `Producer` and the `Consumer`. We shall discuss now how to proceed.

### 6.2 Preliminaries

The first step consists in defining classes for orders and products. For the sake of simplicity, we shall assume that the information carried out by orders consists of three data: the status of the order, the code of the product and the ordered quantity. Consequently, the information carried out by products consists also of three data: the status of the product, its code and the produced quantity. Figure 21 shows the declaration of these two classes.

```
system World
  ...
  system Producer ... end
  ...
  system Supplier
    ...
    system RawMaterial
      ...
    end
  end
  ...
end

system World.Producer
  embeds main.Supplier.RawMaterial as RM;
  ...
  activity ProduceProduct
    trigger:
      return RM.price<priceThreshold ... ;
    ...
  end
end
```

Figure 20: The system `Producer` aggregates the system `RawMaterial`

```
domain OrderStatus {NONE, IN_PREPARATION, PREPARED, PLACED}

class Order
  OrderStatus status(init = NONE);
  int code(init = 0);
  int quantity(init = 0);
end

domain ProductStatus {NONE, ORDERED, IN_PRODUCTION, DELIVERED}

class Product
  ProductStatus status(init = NONE);
  int code(init = 0);
  int quantity(init = 0);
end
```

Figure 21: Classes describing orders and products

```
1  activity World.Consumer.CreateOrder
2    Order* newOrder;
3    trigger:
4      return onGoingOrders<2 and not creatingOrder;
5    start: {
6      creatingOrder = true;
7      newOrder = new Order;
8      newOrder->status = IN_PREPARATION;
9      Orders.append(newOrder)
10     }
11   completion: {
12     newOrder->status = PREPARED;
13     newOrder->code = /* some expression */ ;
14     newOrder->quantity = /* some expression */ ;
15     onGoingOrders += 1;
16     creatingOrder = false;
17     }
18   duration:
19     return 2;
20 end
```

Figure 22: A possible code for the activity `CreateOrder` of the `Consumer`

Both the `Producer` and the `Consumer` must manage a stock of orders and a stock of products. The easiest way to implement that both they compose two subsystems `Orders` and `Products`. Initially, these two subsystems are empty.

## 6.3 Implementation of Activities

The first activity we have to implement is the creation of a new order by the `Consumer`. The triggering condition of this activity must reflect that the `Consumer` is not already creating an order that the number of non yet satisfied orders is less than a given threshold (2 according to our previous discussion). Figure 22 shows a possible code for this activity.

This code deserves a thorough explanation.

First, it declares a variable `newOrder`. This variable is a reference to an instance of the class `Order`, as indicated by the "*" symbol after the name of the class. We use here the same notations as C++. Initially, `newOrder` refers to no instance. Its value is thus undefined.

Checking whether the triggering condition of an activity is like creating a new instance of this activity. If the condition is not verified, the instance is immediately deleted. Otherwise, the instance is deleted at the completion of the activity. The variable `newOrder` is thus local to the instance of the activity.

The triggering condition presents no difficulty (and does involve the variable `newOrder`).

The action at the beginning of the activity starts by creating a new instance of the class `Order` and setting the value of the variable `newOrder` as a reference to this instance (line 7). At this point, the instance is still hanging, i.e. is located nowhere in the hierarchy of the system.

The variable `status` of the instance is then assigned the value `IN_PREPARATION`. Note the use of the arrow "`->`" instead of a dot "`.`" to refer to the variable. This is because `newOrder` is a reference to an instance of `Order` and not directly this instance. Here again, we use the same notation as in C++.

Finally, the instance is inserted (appended) into the system hierarchy, namely as a subsystem of the system (stock) `Orders` (or `World.Supplier.Orders` to be precise). `append` is a built-in function applying on systems. We shall see several others.

The completion of the activity presents no difficulty. Note again that the variable `newOrder` ceases to exist at the end of the completion action. This is not a problem because the created instance of the class

```
activity World.Consumer.PlaceOrder
  Order* order;
  trigger: {
    if placingOrder then return false;
    for order in Orders
      if order->status==PREPARED then return true;
    return false;
    }
  start: {
    placingOrder = true;
    Orders.remove(order)
    }
  completion: {
    order->status = PLACED;
    main.Producer.Orders.append(order)
    placingOrder = false;
    }
  duration:
    return 1;
end
```

Figure 23: A possible code for the activity `PlaceOrder` of the `Consumer`

`Order` is now composed by the system `World.Supplier.Orders`.

The second activity to implement is the activity `PlaceOrder` of the `Supplier`. It consists in picking up a command already prepared in the stock `World.Supplier.Orders` and to move it to the stock `World.Producer.Orders`. Figure 23 shows a possible code for the activity `PlaceOrder`.

As previously, the activity declares a local variable `order` which holds a reference to an instance of the class `Order`.

The triggering condition looks for a prepared order in the stock `World.Supplier.Orders`. It returns true only if such an order is found, in which case the variable `order` refers to this instance.

The action at the beginning of the activity consists in removing the order from the stock `Orders` of the `Supplier`, while the action at completion consists in inserting it into the stock `Orders` of the `Producer` (and changing its status to `PLACED`.

The other activities are implemented in the same way. The primitive `delete` is used to delete instances.

## 6.4 Discussion

The codes shown in Figure 22 and 23 are significantly more complex than the ones we have seen so far. The reason is that the operations they implement are also significantly more complex.

Note that it would be meaningless to compose observers and indicators in dynamically created components. It is however possible that these components compose activities.

Dynamically creating, inserting, looking for, removing and deleting systems requires handling references to objects. Σ provides a versatile set of primitives to implement these operations. It remains that using them demands a certain programming discipline: One must ensure that instances of classes (or clones of systems) are not left hanging, that they are never inserted at two different locations or deleted twice. In return, they make Σ an extremely powerful language.

## 7 Conclusion

In this article, we provided the reader with a guided tour of the Σ modeling language. We showed by means of examples how this language can be used to simulate the dynamic behavior of complex technical and socio-technical systems.

The design of Σ benefited of decades of experience acquired by the authors in the design and the application of modeling languages, notably the researches and developments that have been done around AltaRica.

Σ is an extremely powerful language, making it possible to represent systems with dynamically created, moved and deleted components. This, in turn, makes Σ a best-in-class candidate for the design of systemic digital twins.

It remains that the expressiveness of the language, although a *sine qua none* condition to implement systemic digital twins, is not enough in itself. A solid methodology must be put in place for the design of models. With that respect, the integration of Σ into the now well established CESAMES architecture framework (Krob 2017) is extremely promising, see also (Rauzy 2022).

## 8 References

Abadi, Mauricio and Luca Cardelli (1998). *A Theory of Objects*. New-York, USA: Springer-Verlag. ISBN: 978-0387947754 (cited on pages 16, 19).

Ajmone-Marsan, Marco et al. (1994). *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. New York, NY, USA: John Wiley and Sons. ISBN: 978-0471930594 (cited on page 3).

Batteux, Michel, Tatiana Prosvirnova, and Antoine Rauzy (2018). “From Models of Structures to Structures of Models”. In: *IEEE International Symposium on Systems Engineering (ISSE 2018)*. Roma, Italy: IEEE. DOI: 10.1109/SysEng.2018.8544424 (cited on pages 2, 8, 16, 19).

— (2019). “AltaRica 3.0 in 10 Modeling Patterns”. In: *International Journal of Critical Computer-Based Systems* 9.1–2, pages 133–165. DOI: 10.1504/IJCCBS.2019.098809 (cited on pages 2, 3, 13, 16).

Bouissou, Marc et al. (1991). “Knowledge modelling and reliability processing: presentation of the FIGARO language and of associated tools”. In: *Proceedings of SAFECOMP’91 – IFAC International Conference on Safety of Computer Control Systems*. Edited by Johan F. Lindeberg. Trondheim, Norway: Pergamon Press, pages 69–75. ISBN: 0-08-041697-7 (cited on page 3).

Cassandras, Christos G. and Stéphane Lafortune (2008). *Introduction to Discrete Event Systems*. New-York, NY, USA: Springer. ISBN: 978-0-387-33332-8 (cited on pages 2, 3).

de Weck, Olivier, Daniel Roos, and Christopher L. Magee (2011). *Engineering Systems - Meeting Human Needs in a Complex Technological World*. Cambridge, MA 02142-1315, USA: MIT Press. ISBN: 978-0262016704 (cited on page 1).

Dori, Dov (2016). *Model-Based Systems Engineering with OPM and SysML*. Berlin, Heidelberg, New York: Springer Verlag. ISBN: 978-1-4939-3294-8 (cited on page 2).

Friedenthal, Sanford, Alan Moore, and Rick Steiner (2011). *A Practical Guide to SysML: The Systems Modeling Language*. San Francisco, CA 94104, USA: Morgan Kaufmann. The MK/OMG Press. ISBN: 978-0123852069 (cited on page 2).

Fritzson, Peter (2015). *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Hoboken, NJ 07030-5774, USA: Wiley-IEEE Press. ISBN: 978-1118859124 (cited on page 2).

Garcia, Juan Martin (2018). *System Dynamics Modelling with Vensim*. Independently published. ISBN: 978-1718077027 (cited on page 2).

Güdemann, Matthias and Frank Ortmeier (2010). "A Framework for Qualitative and Quantitative Model-Based Safety Analysis". In: *Proceedings of the IEEE* 12$^{th}$ *High Assurance System Engineering Symposium (HASE 2010)*. San Jose, CA, USA: IEEE, pages 132–141. ISBN: ISBN 978-1-4244-9091-2. DOI: 10.1109/HASE.2010.24 (cited on page 3).

Halbwachs, Nicolas (1993). *Synchronous Programming of Reactive Systems*. MIT Electrical Engineering and Computer Science. Dordrecht, The Netherlands: Kluwer Academic Publisher. ISBN: 978-1441951335 (cited on page 4).

Hatchuel, Armand and Benoit Weill (2009). "C-K design theory: an advanced formulation". In: *Research in Engineering Design* 19.4, pages 181–192 (cited on page 19).

Holt, Jon, Simon Perry, and Mike Brownsword (2016). *Foundations for Model-based Systems Engineering: From patterns to models (Computing and Networks)*. Stevenage Herts, United Kingdom: Institution of Engineering and Technology. ISBN: 978-1785610509 (cited on page 2).

Jensen, Kurt (2014). *Coloured Petri Nets*. Berlin and Heidelberg, Germany: Springer-Verlag. ISBN: 978-3642425813 (cited on page 3).

Kaplan, Edward Lynn and Paul Meier (1958). "Nonparametric estimation from incomplete observations". In: *Journal of American Statistics Association* 53.282, pages 457–481. DOI: 10.2307/2281868 (cited on page 12).

Klee, Harold and Randal Allen (2011). *Simulation of Dynamic Systems with MATLAB and Simulink*. Boca Raton, FL 33431, USA: CRC Press. ISBN: 978-1439836736 (cited on page 2).

Krob, Daniel (2017). *CESAM: CESAMES Systems Architecting Method: A Pocket Guide*. CESAMES. http://www.cesames.net (cited on page 24).

Kwiatkowska, Marta, Gethin Norman, and David Parker (2011). "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proceedings 23rd International Conference on Computer Aided Verification (CAV'11)*. Volume 6806. LNCS. New York, NY, USA: Springer, pages 585–591. ISBN: 978-3642221095 (cited on page 3).

Larsen, Kim G., Paul Pettersson, and Wang Yi (1997). "UPPAAL in a Nutshell". In: *International Journal on Software Tools for Technology Transfer* 1.1–2, pages 134–152. DOI: 10.1007/s100090050010 (cited on page 3).

Maier, Mark W. (1998). "Architecting principles for systems-of-systems". In: *Systems Engineering* 1.4, pages 267–284. DOI: 10.1002/j.2334-5837.1996.tb02054.x (cited on pages 4, 20).

Noble, James, Antero Taivalsaari, and Ivan Moore (1999). *Prototype-Based Programming: Concepts, Languages and Applications*. Berlin and Heidelberg, Germany: Springer-Verlag. ISBN: 978-9814021258 (cited on page 16).

Plateau, Brigitte and William J. Stewart (1997). "Stochastic Automata Networks". In: *Computational Probability*. Kluwer Academic Press, pages 113–152 (cited on page 3).

Rauzy, Antoine (2008). "Guarded Transition Systems: a new States/Events Formalism for Reliability Studies". In: *Journal of Risk and Reliability* 222.4, pages 495–505. DOI: 10.1243/1748006XJRR177 (cited on page 4).

— (2020). *Probabilistic Safety Analysis with XFTA*. Les Essarts le Roi, France: AltaRica Association. ISBN: 978-82-692273-0-7 (cited on page 16).

— (2022). *Model-Based Reliability Engineering – An Introduction from First Principles*. Les Essarts le Roi, France: AltaRica Association. ISBN: 978-82-692273-2-1 (cited on pages 16, 19, 24).

Rauzy, Antoine and Cecilia Haskins (2019). “Foundations for Model-Based Systems Engineering and Model-Based Safety Assessment”. In: *Journal of Systems Engineering* 22, pages 146–155. DOI: 10.1002/sys.21469 (cited on pages 2, 3, 16, 19).

Signoret, Jean-Pierre and Alain Leroy (2021). *Reliability Assessment of Safety and Production Systems: Analysis, Modelling, Calculations and Case Studies*. Springer Series in Reliability Engineering. Cham, Switzerland: Springer. ISBN: 978-3030647070 (cited on page 3).

Sterman, John D. (2000). *Business Dynamics: Systems thinking and modeling for a complex world*. New York, NY, USA: McGraw Hill. ISBN: 978-0-07-231135-8 (cited on page 2).

Voirin, Jean-Luc (2017). *Model-based System and Architecture Engineering with the Arcadia Method*. London, United Kingdom: ISTE Press - Elsevier. ISBN: 978-1785481697 (cited on page 2).

Walden, David D. et al. (2015). *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, fourth edition*. Hoboken, NJ, USA: Wiley-Blackwell. ISBN: 978-1118999400 (cited on page 1).

White, Stephen and Derek Miers (2008). *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, FL, USA: Future Strategies Inc. ISBN: 978-0977752720 (cited on page 2).

Wilensky, Uri and William Rand (2015). *An Introduction to Agent-Based Modeling - Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. Cambridge, MA, USA: The MIT Press. ISBN: 978-0262731898 (cited on page 3).