Σ^{TM} Reference Manual

Systemic Intelligence

Abstract: This manual presents Σ^{TM}, an object-oriented modeling language dedicated to the description and the simulation of the dynamics of complex technical and socio-technical systems.
Σ^{TM} relies onto two pillars: first, the decomposition of the system into a hierarchy of subsystems, the state of each subsytems being described by means of variables. Second, the description of activities performed by subsystems and modifying their states. Although apparently simple, this way of describing complex systems dynamics turns out to be extremely powerful. Discrete event simulation performed on Σ^{TM} models make it possible to play *what-if* scenarios, to assess key performance indicators, and to apply optimization techniques.
Σ^{TM} is at the core of the WorldLab technology that aims at developing systemic digital twins of complex systems, thereby helping decision-makers to manage their strategic assets.
This manual introduces Σ^{TM} concepts and describes the syntax and the semantics of Σ^{TM} constructs. It provides a reference for WorldLab users. Some of the constructs described here are not yet implemented in the Σ^{TM} tool suite. We shall mention explicitly when it is the case.

Author(s) Antoine Rauzy
Reference SIG-RM-2023-003
Version 1.5.6
Date September 2023

Table of Contents

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 (deWeck, 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 the heterogeneous characteristics of systems in the models.

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 document aims at providing a guided tour of the Σ™ modeling framework and at illustrating its constructs by means of examples. It provides also a reference for WorldLab users.

Organization of the manual

The remainder of this manual is organized as follows.

- Section 2 introduces the Σ^{TM} modeling framework.
- Section 3 presents the role and the declaration of Σ^{TM} variables, parameters, observers and indicators.
- Section 4 describes Σ^{TM} expressions.
- Section 5 describes the set of instructions of Σ^{TM}.
- Section 6 describes S2ML constructs implemented in Σ^{TM}.
- Section 7 provides more insight into how activities work.
- Section 8 provides more insight about variables and expressions.

In addition to the above sections, it contains an appendix.

- Appendix A gives the grammar of Σ^{TM} in extended Backus-Naur form (EBNF).

2 Getting Started

2.1 Σ^{TM} Ontology

As explained in the introduction, in the Σ^{TM} 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^{TM} 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 state of system and possibly its structure are 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 the `Supplier`, which then delivers them to the `Producer`.

Figure 1 shows the system breakdown structure (the hierarchical decomposition) of this production system. It consists of the `World` and three subsystems: `Supplier`, `Producer` and `Consumer`.

Each system owns stocks, i.e. certain quantities of raw materials, products and orders. The above figure 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

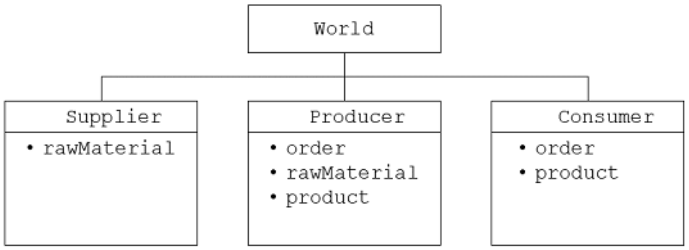

Figure 1: Producer system breakdown structure

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.

2.2 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 the production system, decomposed as shown in Figure 1. We assume here that all stocks are represented by means of integers.

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 an integer variable `rawMaterial`, the subsystem `Producer` declares three integer variables `order`, `rawMaterial` and `product`, and finally the subsystem `Consumer` declares two integer variables `order` and `product`. Initially, all the variables take the value `0`, which is indicated by their attribute `init`.

2.3 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`.

```
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 a $\Sigma^{™}$ model that represents the production system.

Redeclarations are a means to avoid nested descriptions. A $\Sigma^{™}$ 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*. These transformations preserve indeed 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 a code, equivalent to the one given in Figure 2 that applies this idea.

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

2.4 Activities

As explained above, in $\Sigma^{™}$, 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*, 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.

```
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 a Σ^{TM} model with redeclarations that represents the production system.

Figure 4 gives 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`.

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 start of the activity books the resources necessary to perform it. Namely, it sets the variable `renewing` to true.

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

As we shall see, in Σ^{TM}, instructions can implement more or less complex calculation procedures. They may involve the call of functions and they may 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

```
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: Σ™ code for the activity `RenewRawMaterialStock` of the `Supplier`.

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. Consequently, one needs a means to refer to this variable within the activity `RenewRawMaterialStock` of 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:

– Using 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`.

```
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: Σ^{TM} code for the activity `RenewRawMaterialStock` of the `Producer`.

– Using 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.

2.5 Executions

The semantics of a $\Sigma^{™}$ 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 Section **??**.

2.6 Terminology and Additional Syntactic Constructs

2.6.1 Terminology

$\Sigma^{™}$ 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, $\Sigma^{™}$ 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, $\Sigma^{™}$ makes it possible to declare a subsystem$T$ in the system T and then move it to another part of the hierarchy. It makes also possible to create and delete subsystems. 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.

2.6.2 Identifiers and Paths

Σ™ *identifiers* for variables, systems, activities... are those of most of programming and modeling languages: an identifier starts with a letter or an underscore, followed by any number of letters, digits and underscores. E.g. `oreStock`, `_working`, `road66`.

Paths are basically identifiers separated with dots. `MiningSupportVessel.oreStock`, `Plant.SourceConnector.Line1`. Paths can also contain references to the parent system, using the keyword `owner`, or to the top system of the hierarchy, using the keyword `main`. E.g. `owner.Train1.Unit1.In`, `main.Train1.Unit1.In`.

2.6.3 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
```

Multiline comments start with `/*` and finish with `*/`. E.g.

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

2.6.4 Units

The experience shows that it is often very convenient to indicate the unit of stocks (cubic meters, tons, euros, days...). 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.

3 Variables

As we have seen in the previous section, variables are used to describe the state of the system under study. There are actually four types of variables in Σ™: variables strictly speaking, parameters, observers and indicators. This section presents their different roles, the way they are declared and used.

Throughout this section, we shall use a revisited version of the code we already looked at in the previous section. The corresponding code is given in Figure 6.

3.1 Basic Types and Domains

Variables are value holders. Σ™ is a strongly typed language: the value hold by a variable is given by a type when the variable is declared, and does not change while it is used.

The type of a variable is either a basic type or an user defined domain.

Basic types are the following.

```
domain State {STANDBY, WORKING}

system World.Producer
    int order(init = 0);
    int rawMaterial(init = 0);
    int product(init = 0);
    State renewing(init = STANDBY);
    parameter int renewedQuantity = 20;
    parameter int renewingDuration = 10;
    observer RawMaterial = rawMaterial;
end

activity World.Producer.RenewRawMaterialStock
    trigger: {
        int requiredQuantity;
        requiredQuantity = order*5 + 15;
        return main.Supplier.rawMaterial>=renewedQuantity
            and rawMaterial<=requiredQuantity
            and renewing==STANDBY;
        }
    start: {
        renewing = WORKING;
        main.Supplier.rawMaterial -= renewedQuantity;
        }
    completion: {
        renewing = STANDBY;
        rawMaterial += renewedQuantity;
        }
    duration:
        return renewingDuration;
end
```

Figure 6: Revisited Σ^{TM} code for the activity `RenewRawMaterialStock` of the `Producer`.

– `bool` for Boolean values;

– `int` for integers;

– `float` for floating-point numbers;

– `id` for identifiers;

– `str` for strings.

A *user defined domain* is a finite, usually small, set of symbolic constants. Domains are declared as follows (see also Figure 6).

```
domain State {STANDBY, WORKING}
```

The keyword `domain` introduces the declaration of a domain. It is followed by the name of a domain, here `State`, and the set of symbolic constants belonging to the domain, here `STANDBY`, and `WORKING`.

In the above declaration, we wrote symbolic constants using capital letters. Although this is not required by the Σ™ grammar, this is a common usage that makes possible to distinguish symbolic constants from variables.

Note also that a symbolic constant can belong to several domains.

3.2 State and Temporary Variables

As already said, (regular) *variables* are used to describe the state of the system. They are declared (or redeclared) withing systems. E.g.

```
int rawMaterial(init = 0, unit = "t");
State renewing(init = STANDBY);
```

First comes the type (basic type or domain) of the variable, here `int`. Then its identifier, here `rawMaterial`. Then an optional list of attributes surrounded with parentheses and separated with commas. The declaration ends with a semicolon. The declaration of the variable `renewing` involves the domain `State`.

Attributes are pairs (name, value) separated with an equal sign. The value of an attribute is an expression, see Section 4.

Variables takes two attributes:

– The attribute `init`, which gives the initial value of the variable.

– The attribute `unit`, which gives the unit of the variable (see Section 2.6.4).

The attribute `init` is mandatory when the variable is a *state variable*, i.e. it is declared in a system. It is optional (and preferably not given) when the variable is a *temporary variable*, i.e. declared, for the sake of convenience, within an activity. In the code given in Figure 6, the variables `order`, `rawMaterial`, `product` and `renewing` are state variables, while variable `requiredQuantity` is a temporary variable.

The attribute `unit` is always optional.

Attributes of variables can be redeclared, possibly in a different system that they were initially declared (therefore using a path rather an identifier in the declaration). E.g.

```
int main.Producer.rawMaterial(init = 42, unit = "t");
```

3.3 Constants and Parameters

3.3.1 Parameters

Parameters can be seen as variables are set once for all. They are used instead of constants, so to document the models and to make their modification easier. In the code given in Figure 6, the parameters `renewedQuantity` and `renewingDuration` are used for this very purpose.

The declaration of parameters is as follows.

```
parameter int renewedQuantity(unit = "ton") = 20;
parameter int renewingDuration(unit = "day") = 10;
```

First comes the keyword `parameter`, then the type of the parameter, then its identifier, then optionally some attributes, and finally its value (after an equal sign), which is an expression. The declaration ends with a semicolon.

The only attribute of interest for parameters is their units.

Values of parameters can be redeclared, possibly in a different system that they were initially declared (therefore using a path rather an identifier in the declaration). E.g.

```
int main.Producer.renewingDuration = 12;
```

Another important interest of parameters is that it is possible to change their value at the beginning of a simulation (either interactive or stochastic). The model stays otherwise the same. Consequently, there is no need to recompile it.

3.3.2 Constants

Constants are just like parameters, except that they cannot be changed at the beginning of a simulation.

The declaration of constants is as follows.

```
constant float G (unit = "m/s2") = 9.81;
```

3.4 Observers

The main objective of Σ™ modeling is to assess performance indicators. Performance indicators are numerical quantities. In Σ™, they are defined via *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. Observers serve as an interface between the model and the tool(s) making statistics on executions of the model.

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 integral of its value since the beginning of the current execution;
- The first date at which it changed of value after 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 material of the Producer. Then, we can declare an observer as illustrated in Figure 6.

As for parameters, it is possible to associate a unit with observers. E.g.

The declaration of observers is as follows.

```
observer RawMaterial(unit = "ton") = rawMaterial;
```

The observer `RawMaterial` makes it possible to follow the evolution of the stock of raw material. 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 0/1 expression, as follows.

```
observer LowLevelRawMaterial = if rawMaterial<=30 then 1 else 0;
```

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

This example shows that observers as Σ™ defines them give thus a significant amount of information on quantities of interest. The experience with AltaRica 3.0 (Batteux, Prosvirnova, and Rauzy, 2019) shows that all practical needs can be actually covered by this approach.

Interactive simulators make it possible to save the *time series* of the values of observers in a CSV file at the end of the simulation, i.e. when the mission time is reached, see the Σ Workshop™ user manual for more details (Rauzy, 2023). This requires specifying the name of the file by means of the attribute `timeSeries` when declaring the observer. It works as follows.

```
observer RawMaterial(unit = "ton", timeSeries = "Results/rawMaterial.tsv
    ↪ ") =
     rawMaterial;
```

From there, the time series is automatically saved at the end of each interactive simulation.

3.5 Indicators

When performing *Monte-Carlo simulations*, statistical indicators are defined from observers. Conversely to observers, that are continuously updated throughout executions, these 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 wants 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. E.g.

```
system World.Producer
    indicator probabilityTooLowStock(mean=true, standardDeviation=true)
        ↪ = max(lowLevelRawMaterial);
```

Table 1: Attributes to specify the statistical measures to be calculated with indicators

Attribute	*Type*	*Statistical measure*
`mean`	bool	Mean
`standardDeviation`	bool	Standard-deviation
`min`	bool	Minimum
`max`	bool	Maximum
`confidenceRange90`	bool	90% confidence range
`confidenceRange95`	bool	95% confidence range
`confidenceRange99`	bool	99% confidence range
`bins`	int	Bins
`shrinkFactor`	int	Shrink factor of the bin table

```
    indicator sojournTimeTooLowStock(mean=true, standardDeviation=true,
        ↪ bins=10) = mean(lowLevelRawMaterial);
end
```

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 minimum and maximum values;
- Its mean value, standard deviation, 90, 95 and 99 percents confidence interval.
- Quantiles and distributions.

The statistics to be made are specified by means of attributes, as shown above. Table 1 gives these attributes, their types and their meanings.

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.

Further explanations on statistics performed by stochastic simulation can be found in the Σ Workshop™ user's manual (Rauzy, 2023).

4 Expressions

This section presents the syntax and the semantics of Σ™ expressions. Their syntax is specified using the extended Backus-Naur form (EBNF), see Appendix A. Their semantics is specified using their signatures, see next section.

Σ™ expressions belong to one of the following categories.

- Constants;
- References to variables and parameters;

– Boolean operations;

– Inequalities;

– Arithmetic expressions;

– Builtin expression;

– Conditional expressions;

– Probability distributions and random deviates;

– Time primitives.

The syntax of Σ™ expressions is thus as follows.
The following production rule is the root of the EBNF grammar of Σ™ expressions.

```
Expression ::=
      VariableReference
    | BooleanExpression
    | Inequality
    | ArithmeticExpression
    | BuiltInExpression
    | ConditionalExpression
    | ProbabilityDistribution
    | RandomDeviate
    | TimePrimitive
```

Some expressions and instructions take as arguments expressions that return a Boolean. The latter are either references to variables and parameters, Boolean expressions strictly speaking, or inequalities. We call such expressions *conditions*.

```
      VariableReference // if type bool
    | BooleanExpression
    | Inequality
    | ConditionalExpression // if return type bool
```

The remainder of this section specifies expressions of each of these categories in turn. But before doing so, we need to introduce signatures.

4.1 Signatures

Sigma provides a wide range of operators to construct expressions. Expressions are themselves used in instructions, which are themselves used in triggers, instructions at start, instructions at completion, and durations of activities.

Operators ranges from usual Boolean and arithmetic operators, to arithmetic builtins and random deviates. Each operator takes a certain number of arguments. Each of these arguments must have a certain basic type. The operator itself returns a value, that has also a certain basic type.

Sigma *basic types* are the following.

– `bool` for Boolean values;

– `int` for integers;

– `float` for floating-point numbers;

– `num` which are either integers or floating-point numbers;

– `id` for identifiers (including symbolic constants);

– `str` for strings; and finally,

– `any` for any of the above types.

The *signature* or the *type* of an operator taking k arguments of respective types $t_1,\ldots t_k$ and returning a value of type t is denoted $t_1 \times \cdots \times t_k \to t$.

For instance, the signature of the Boolean negation `not` is `bool` $\to$ `bool`, while the signature of the division `/` is `num` $\times$ `num` $\to$ `float` (the division always return a floating point number).

When the operator takes any positive number of arguments of the same type a and returns a value of type t, we shall denote its signature $a+ \to t$.

For instance, the signature of the Boolean conjunction `and` is `bool`$+ \to$ `bool`, while the signature of the multiplication is `num`$+ \to$ `num`. As usual, we shall admit implicitly that if all arguments are integers, the result is also an integer, and that it is a floating-point number otherwise.

Throughout the section, we shall thus specify signatures of operators.

4.2 Constants

Σ^{TM} implements usual *constants* of programming and modeling languages:

– The two Boolean constants `false` and `true`;

– Integers, e.g. `123`;

– Floating point numbers, e.g. `0.456`, `1.23e-4`.

– Strings, i.e. any sequence of characters surrounded with double quotes, e.g. `"This is a string"`;

In addition, some identifiers are considered as symbolic constants, e.g. `STANDBY`.

4.3 References to Variables and Parameters

References to variables and parameters are identifiers or paths. E.g.

```
status==TODO and main.SelectStaff.status==COMPLETED
```

In the above expression, both `status` and `main.SelectStaff.status` are references to variables. The type of a reference to a variable or a parameter is indeed the type of the referred object.

The syntax of paths is as follows.

```
Identifier ::= [_A-Za-z][_A-Za-z0-9]+
Path ::=
      Identifier
    | Identifier '.' Path
    | 'main' '.' Path
    | 'owner' '.' Path
```

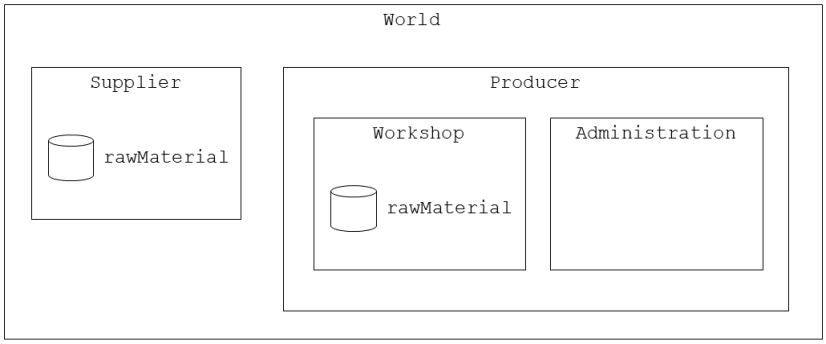

Figure 7: A production system

An *identifier* is thus a sequence of characters, digits and underscores starting with a letter or an underscore. E.g. `rawMaterial`, `production_line_3`...

A *path* is thus a sequence of identifiers separated with dots. Moreover, paths can involve the two keywords `main` and `owner`. `main` main refers to the top system of the hierarchy. `owner` main refers to the parent system in the hierarchy. Paths containing the keyword `main` are called *absolute paths*. All other paths are *relative paths*.

As an illustration, consider the system pictured in Figure 7.

The system `World` composes (declares) two subsystems: `Supplier` and `Producer`. The system `Supplier` composes the variable `rawMaterial`. The system `Producer` composes two subsystems: `Workshop` and `Administration`. Finally, the system `Workshop` composes the variable `rawMaterial`.

Anywhere in the model, we can access the variable `rawMaterial` composed by the system `Supplier` by means of the absolute path:

```
main.Supplier.rawMaterial
```

Similarly, we can access the variable `rawMaterial` composed by the system `Workshop` by means of the absolute path:

```
main.Producer.Workshop.rawMaterial
```

We can also use relative paths to access these variables:

- In the system `Supplier`, its variable `rawMaterial` is simply accessed by the identifier `rawMaterial`.
- In the system `World`, this variable is accessed by the relative path:

```
Supplier.rawMaterial
```

- In the system `Producer`, this variable is accessed by the relative path:

```
owner.Supplier.rawMaterial
```

  `owner` refers to the parent system, here `World`, then the dot notation applies.

- In the systems `Workshop` and `Administration`, this variable is accessed by the relative path:

Table 2: Boolean connectives

Symbol	*Signature*	*Semantics*
`and`	`bool+` → `bool`	Conjuction
`or`	`bool+` → `bool`	Disjunction
`not`	`bool` → `bool`	Negation

```
owner.owner.Supplier.rawMaterial
```

The above principle works in all directions:

– In the system `World`, the variable `rawMaterial` of the system `Workshop` is accessed by the relative path:

```
Producer.Workshop.rawMaterial
```

– In the system `Administration`, it is accessed by the relative path:

```
owner.Workshop.rawMaterial
```

And so on...

4.4 Boolean Expressions

Σ^{TM} implements the usual *Boolean expressions*: the constants `false` and `true`, and the connectives `and`, `or`, and `not` with their usual meaning. Their syntax is as follows.

```
BooleanExpression ::=
      Expression ('or' Expression)*
    | Expression ('and' Expression)*
    | 'not' Expression
```

Here follows an example of Boolean expression.

```
status==TODO and main.SelectStaff.status==COMPLETED
```

The signatures of Boolean connectives are given in Table 2.

As usual, the operator `not` has priority over the operator `and` which itself has the priority over the operator `or`. For instance, the expression `A and B or not A and C` is thus interpreted as `(A and B) or ((not A) and C)`.

4.5 Inequalities

Σ^{TM} implements the usual *inequalities* to compare values of left and right expressions:

– `F == G` is true if the value of `F` equals the value of `G`, and false otherwise.

– `F != G` is true if the value of `F` differs from the value of `G`, and false otherwise.

– `F < G` is true if the value of `F` is less than the value of `G`, and false otherwise.

– `F <= G` is true if the value of `F` is less or equal to the value of `G` , and false otherwise.

Table 3: Inequalities

Symbol	*Signature*	*Semantics*
==	any × any → bool	$=$
!=	any × any → bool	$\neq$
<	num × num → bool	$<$
<=	num × num → bool	$\leq$
>	num × num → bool	$>$
>=	num × num → bool	$\geq$

– `F > G` is true if the value of `F` is greater than the value of `G` , and false otherwise.

– `F >= G` is true if the value of `F` is greater or equal to the value of `G` , and false otherwise.

Equality and difference apply to all types of values, while the four other inequalities apply only to numerical values.

Table 3 gives the signatures of inequalities.

Inequalities have a higher priority than Boolean operators. Hence `F>=0 and F<4` is interpreted as `(F>=0) and (F<4)`.

4.6 Arithmetic Expressions

Σ^{TM} implements usual arithmetic operators: + (addition), – (subtraction), ⋆ (multiplication), / (division), `div` (Euclidian division), `mod` (modulo) and unary – (opposite).

Their syntax is as follows.

```
ArithmeticExpression ::=
      Expression ('+' Expression)*
    | Expression '-' Expression
    | Expression ('*' Expression)*
    | Expression '/' Expression
    | Expression 'div' Expression
    | Expression 'mod' Expression
    | '-' Expression
```

E.g.

```
3*x + 6*y*z + 4
(x+1) / 2
-y
42 mod 11
```

Table 4 gives the signatures of usual arithmetic expressions.

There are however several subtleties here:

– The result of the division `/` is always a floating point number, even though both its arguments are integers and the numerator is a multiple of the denominator, e.g. `6 / 3` gives `2.0` and not `2`.

– Both arguments of Euclidian division `div` and its result are integers, e.g. `7 div 3` gives `2`. Same thing for `mod`.

– `-`, `/`, `div` and `mod` are binary operators and should not be mixed with + and ⋆. E.g. `1 - 2 - 3` is not a valid expression, because of its fundamental ambiguity. It must be written either `(1 - 2) - 3` or `1 - (2 - 3)`. The same applies for instance for `2 / 3 * 4`.

Table 4: Arithmetic expressions

Symbol	*Signature*	*Semantics*
+	num+ → num	Addition
-	num × num → num	Subtraction
*	num+ → num	Multiplication
/	num × num → float	Division
div	int × int → int	Euclidian division
mod	int × int → int	Modulo
-	num → num	≥

Table 5: Usual mathematical operations

Symbol	*Signature*	*Semantics*
abs	num → float	Absolute value
ceil	num → float	Smaller integer larger or equal to the argument
exp	num → float	Exponential
floor	num → float	Larger integer smaller or equal to the argument
log	num → float	Natural logarithm
log10	num → float	Base 10 Logarithm
pow	num × num → float	Power
sqrt	num → float	Square root
max	num+ → float	Maximum (any number of arguments)
min	num+ → float	Minimum (any number of arguments)
#	bool+ → int	Number of true arguments

4.7 Built-in Functions

Σ^{TM} implements "usual" builtin functions.

Their syntax is as follows.

```
BuiltIn ::= UnaryBuiltIn | BinaryBuiltIn | AssociativeBuiltIn

UnaryBuiltIn ::= UnaryBuiltInSymbol '(' Expression ')'
UnaryBuiltInSymbol ::=
      'abs' | 'exp' | 'log' | 'log10' | 'sqrt' | 'ceil' | 'floor'
    | 'acos' | 'asin' | 'atan' | 'cos' | 'sin' | 'tan'
    | 'bool' | 'int' | 'float' | 'id' | 'str'

BinaryBuiltIn ::= BinaryBuiltInSymbol '(' Expression ',' Expression ')'
BinaryBuiltInSymbol ::= 'pow'

AssociativeBuilIn ::=
    AssociativeBuiltInSymbol '(' Expression (',' Expression)* ')'
AssociativeBuiltInSymbol ::= 'min' | 'max' | '#'
```

4.7.1 Usual mathematical operations

Table 5 gives the implemented usual mathematical operations, their expected number of arguments and their semantics.

E.g.

Table 6: Trigonometric functions

Symbol	*Signature*	*Semantics*
`cos`	`num` → `float`	Cosine
`sin`	`num` → `float`	Sine
`tan`	`num` → `float`	Tangent
`acos`	`num` → `float`	Arc cosine
`asin`	`num` → `float`	Arc sine
`atan`	`num` → `float`	Arc tangent

Table 7: Casts

Symbol	*Signature*	*Semantics*
`bool`	`any` → `bool`	Cast to Boolean
`int`	`any` → `int`	Cast to integer
`float`	`any` → `float`	Cast to float
`id`	`any` → `id`	Cast to symbol
`str`	`any` → `str`	Cast to string

```
abs(-5.67)
min(-1, 33.0, -12)
pow(2.0, 10)
```

Note the special construct `#` that counts the number of its arguments that have the value `true` . E.g. `#(A, B, C)` takes the value 2 if both `A` and `C` are true and `B` is false.

4.7.2 Trigonometric Functions

Table 6 gives the implemented trigonometric functions, their expected number of arguments and their semantics.

E.g.

```
cos(1.23)
```

4.7.3 Casts

Table 7 gives the implemented casts, their expected number of arguments and their semantics.

E.g.

```
int(1.23)
str(42)
```

Note that these operations may be invalid. E.g. it is impossible to cast the string `"a string"` to a symbol, because `a string` is not a valid identifier.

4.8 Conditional Expressions

The current version of Σ™ implements only one condition expression: the *if-then-else expression*. Its syntax is as follows.

Table 8: Conditional expressions

Symbol	*Signature*	*Semantics*
`if-then-else`	`bool` × `any` × `any` → `bool`	If-Then-Else

```
ConditionalExpression ::=
    IfThenElseExpression

IfThenElseExpression ::=
    'if' Condition 'then' Expression 'else' Expression
```

E.g.

```
if rawMaterial>=10 then rawMaterial else 0
```

Table 8 gives the signature of this expression.

4.9 Probability Distributions and Random Deviates

Probability distributions and random deviates come in some sense in pair.

A *probability distribution* is a non decreasing function from $\mathbb{R}^+$ into $[0,1]+$. Intuitively, a probability distribution associates to each time t, the probability that a certain event is realized before t.

A *random deviate* is the inverse of a probability distribution. Intuitively, a random deviate picks up a number z uniformly at random between $[0,1]+$ and returns the first time t such that the probability that a certain is realized before t is greater or equal to z.

Probability distributions and random deviates are at the core of the treatment of uncertainties in Σ^{TM}.

Σ^{TM} generalizes the notions of probability distributions and random deviates, in particular by allowing the codomain of distributions (and consequently the domain of random deviates) to be other sets than $[0,1]+$.

Moreover, it considers two types of probability distributions (respectively random deviates):

- *Parametric probability distribution* that are predefined, builtin, functions that depend on a few parameters.
- *Empirical probability distribution* that are defined by sets of pairs (time, value), themselves given in a file. In between two points, the value of the distribution is obtained by interpolation.

We shall give here the mathematical definitions of available distributions and random deviates as well as syntax and signatures.

4.9.1 Parametric Probability Distributions

The current version of Σ^{TM} implements the following probability distributions:

- The *exponential distribution* that takes two parameters: a rate λ and an time t, $\lambda > 0, t \geq 0$, and returns:

$$\texttt{exponentialDistribution}(\lambda, t) \stackrel{def}{=} 1 - \exp(-\lambda \times t)$$

Table 9: Parametric probability distributions

Symbol	*Signature*
`exponentialDistribution`	$\mathbb{R}^+ \times \mathbb{R}^+ \to [0,1]$
`WeibullDistribution`	$\mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+ \to [0,1]$
`DiracDistribution`	$\mathbb{R}^+ \times \mathbb{R}^+ \to [0,1]$

See `https://en.wikipedia.org/wiki/Exponential_distribution` for more details on the exponential distribution.

– The *Weibull distribution* that takes three parameters: a scale factor α, shape factor β and an time t, $\alpha, \beta > 0, t \geq 0$, and returns:

$$\texttt{WeibullDistribution}(\alpha, \beta, t) \stackrel{def}{=} 1 - \exp\left(-\left(\frac{t}{\alpha}\right)^{\beta}\right)$$

See `https://en.wikipedia.org/wiki/Weibull_distribution` for more details on the Weibull distribution.

– The *Dirac distribution* that takes two parameters: a time T and a time t, $T, t \geq 0$, and returns:

$$\texttt{DiracDistribution}(T, t) \stackrel{def}{=} \begin{cases} 0 & \text{if } t < T \\ 1 & \text{otherwise} \end{cases}$$

The syntax of parametric probability distributions is as follows.

```
ParametricProbabilityDistribution ::=
      'exponentialDistribution' '(' Expression ',' Expression ')'
    | 'WeibullDistribution' '(' Expression ',' Expression ',' Expression
      ↪ ')'
    | 'DiracDistribution' '(' Expression ',' Expression ')'
```

E.g.

```
exponentialDistribution(arrivalRate, currentTime())
WeibullDistribution(alpha, beta, t)
```

Table 9 gives the signature of available parametric probability distributions.

4.9.2 Parametric Random Deviates

The current version of $\Sigma^{\text{™}}$ implements the following random deviates:

– The *uniform deviate* that takes two parameters: a lower bound l and an upper bound h, $l < h$, and returns a number between l and h :

$$\texttt{uniformDeviate}(l, h) \stackrel{def}{=} l + (h - l) \times z$$

where z in a number drawn uniformly at random in $[0, 1]$.

– The *triangular deviate* that takes two parameters: a lower bound l, an upper bound h, and a mode l, $l \leq m \leq h$, and returns a number drawn at random according to a triangular distribution (`https://en.wikipedia.org/wiki/Triangular_distribution`) with these three parameters.

– The *normal deviate* that takes two parameters: a mean μ and a standard deviation σ, $\sigma > 0$, and returns a number drawn at random according to a normal distribution (`https://en.wikipedia.org/wiki/Normal_distribution`), with these two parameters.

– The *lognormal deviate* that takes two parameters: a mean μ and a standard deviation σ, $\sigma > 0$, and returns a number drawn at random according to a lognormal distribution (`https://en.wikipedia.org/wiki/Log-normal_distribution`) with these two parameters.

– The *exponential deviate* that takes one parameter: a rate λ, $\lambda > 0$, and returns a number drawn at random such that:

$$\texttt{exponentialDeviate}(\lambda) \stackrel{def}{=} \frac{\log(z)}{\lambda}$$

where z is a number drawn uniformly at random in $[0, 1]$.

– The *Weibull deviate* that takes two parameters: a scale factor α and shape factor β, $\alpha, \beta > 0$ and returns a number drawn at random such that:

$$\texttt{WeibullDeviate}(\alpha, \beta) \stackrel{def}{=} \alpha \times \log(z)^{\frac{1}{\beta}}$$

where z is a number drawn uniformly at random in $[0, 1]$.

– The *range deviate* that takes two parameters: a lower bound l and an upper bound h, both integers and such that $l \leq h$, and returns an integer drawn at random uniformly and random in the interval $[l, h]$

The syntax of parametric random deviates is as follows.

```
ParametricRandomdeviate ::=
      'uniformDeviate' '(' Expression ',' Expression ')'
    | 'triangularDeviate' '(' Expression ',' Expression ',' Expression
      ↪ ')'
    | 'normalDeviate' '(' Expression ',' Expression ')'
    | 'lognormalDeviate' '(' Expression ',' Expression ')'
    | 'exponentialDeviate' '(' Expression ')'
    | 'WeibullDeviate' '(' Expression ',' Expression ')'
    | 'rangeDeviate' '(' Expression ',' Expression ')'
```

E.g.

```
uniformDeviate(3.1, 3.6)
rangeDeviate(0, 10)
```

Table 10 gives the signature of available parametric random deviates.

4.9.3 Empirical Distributions and Deviates

Empirical distributions and *empirical deviates* are characterized by set of points (time, value). These points are stored into a CSV file, where values are separated by tabulations. Figure 8 shows such a file.

An empirical distribution D can thus be seen as a list of points $(t_0, v_0) \cdots (t_n, v_n)$, such that $t_0 < \cdots < t_n$. Σ^{TM} implements two ways of calculating the value of D at time t:

Table 10: Parametric random deviates

Symbol	*Signature*
`uniformDeviate`	$\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
`triangularDeviate`	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
`normalDeviate`	$\mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$
`lognormalDeviate`	$\mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$
`exponentialDeviate`	$\mathbb{R}^+ \to \mathbb{R}^+$
`WeibullDeviate`	$\mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$
`rangeDeviate`	$\mathbb{N} \times \mathbb{N} \to \mathbb{N}$

```
0    0.0
1000     0.1
2000     0.3
3000     0.6
4000     0.95
5000     0.99
```

Figure 8: A CSV file containing an empirical distribution

– By means of a *stepwise interpolation*:

$$D(t) \stackrel{def}{=} \begin{cases} v_0 & \text{if } t \leq t_0 \\ v_i & \text{if } t_i < t \leq t_{i+1} \\ v_n & \text{if } t > t_n \end{cases}$$

– By means of a *linear interpolation*:

$$D(t) \stackrel{def}{=} \begin{cases} v_0 & \text{if } t \leq t_0 \\ v_i + (v_{i+1} - v_i) \times \frac{t - t_i}{t_{i+1} - t_i} & \text{if } t_i < t \leq t_{+1} \\ v_n & \text{if } t > t_n \end{cases}$$

Note that values of empirical distributions do not need to be between 0 and 1. This extension is extremely useful to deal with external data.

A random deviate R can thus be seen as a list of points $(t_0, v_0) \cdots (t_n, v_n)$, such that $t_0 < \cdots < t_n$ and $0 \leq v_0 < \cdots < v_n \leq 1$.

Σ^{TM} implements two ways of calculating the value of R:

– By means of a *stepwise interpolation*:

$$R \stackrel{def}{=} \begin{cases} t_0 & \text{if } z \leq v_0 \\ t_i & \text{if } v_i < z \leq v_{i+1} \\ t_n & \text{if } z > t_n \end{cases}$$

where z is a number drawn uniformly at random in $[0, 1]$.

– By means of a *linear interpolation*:

$$R \stackrel{def}{=} \begin{cases} t_0 & \text{if } z \leq v_0 \\ t_i + (t_{i+1} - t_i) \times \frac{z - v_i}{v_{i+1} - v_i} & \text{if } v_i < z \leq v_{+1} \\ t_n & \text{if } v > v_n \end{cases}$$

Table 11: Time Primitives

Symbol	*Signature*	*Semantics*
`currentTime`	`bool`	Current simulation time
`missionTime`	`int`	Mission time

where z is a number drawn uniformly at random in $[0, 1]$.

The syntax of empirical distributions and deviates is as follows.

```
EmpiricalDistribution ::=
    'empiricalDistribution' '(' String ',' Interpolation ')'

EmpiricalDeviate ::=
    'empiricalDeviate' '(' String ',' Interpolation ')'

Interpolation ::=
    'step' | 'linear'

String = "[^"]+"
```

E.g.

```
empiricalDistribution("../Data/WaveHeight.csv", linear)
```

⚠ The path to the CSV file containing data (defining the empirical distribution) should better not contain backslashes or these backslashes must be doubled. The reason is that the backslash is an escape character in nearly all programming languages. It is very unfortunate that Windows uses it as the separator in paths. The solution consists in substituting slashes for backslashes.

4.10 Time Primitives

Σ^{TM} implements two *time primitives*:

- `currentTime` that takes no argument and returns the current simulation time; and
- `missionTime` that takes no argument and returns the mission time.

E.g.

```
remainingTime = missionTime() - currentTime()
```

Table 11 gives their signature.

5 Instructions

The current version of Σ^{TM} provides a limited set of instructions. This section reviews them in turn.

5.1 Skip

The instruction `skip` does... nothing. It is seldom used in practice. It consists of the keyword `skip` followed by a semicolon `;`. E.g.

```
skip;
```

5.2 Assignment

Changes of values of variables are performed by means of assignments. An *assignment* consists of the path of the variable to be assigned, the symbol =, the expression defining the value of the variable, and finally a semicolon `;`. E.g.

```
requiredQuantity = order*5 + 15;
```

There are two specialized forms of assignments, the *increment* that adds a value to a variable and the *decrement* that removes a value from the variable. Symbols for increment and decrement are respectively += and −=. E.g.

```
Supplier.rawMaterial -= renewedQuantity;
rawMaterial += renewedQuantity;
```

A variable must be assigned a value with a type compatible with its own type. E.g. if the variable is Boolean, the right-hand side expression must evaluate to a Boolean value. Similarly, if the variable is a float, the right-hand side expression must evaluate either to an int or to a float value. Cast operators are provided in case adjustments must be made.

The left hand side of an increment must be either a int, a float or a string variable (and indeed its right hand side value must be compatible with the left hand side type). Similarly, The left hand side of an increment must be either a int or a float variable.

5.3 If-Then-Else

As most, if not all, programming and modeling languages, Sigma provides a *conditional instruction* `if` condition `then` instruction `else` instruction. The `else` clause is optional.

```
if waveHeight <= 1.25 then
    seaCondition = CALM;
else if waveHeight <= 4 then
    seaCondition = MODERATE;
else
    seaCondition = ROUGH;
```

5.4 While

Since version 1.5 of Sigma tools, Sigma provides a *while loop* that repeats the same instruction until a given condition is fulfilled. Its syntax is `while` condition instruction. E.g.

```
int i;
float z;
i = 1;
count = 0;
while i<10 {
    z = randomDeviate(0.0, 1.0);
    randomDeviate z <= 0.2 then
```

```
        count += 1;
    i += 1;
    }
```

5.5 Return

The instruction `return` is used to return a value while exiting the current sequence of instructions. Clauses `trigger` and `duration` of activities must finish with such `return` instruction. The instruction consists of the keyword `return` followed the expression defining the value to be returned and terminated with a semicolon `;`. E.g.

```
duration:
    return 33;
```

5.6 Blocks of Instructions

Blocks of instructions are sequences of instructions executed in a row. A block of instruction starts with the curly brace { and ends with the curly brace }. E.g.

```
completion: {
    rawNoduleStock -= dewateredQuantity;
    dewateredNoduleStock += dewateredQuantity;
    dewateringState = STANDBY;
    }
```

6 S2ML Constructs

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 (Rauzy, 2022).

6.1 Cloning

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

For instance, the system `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.

The code given in Figure 9 illustrates the use of this mechanism.

This code creates a second machine, `Machine2`, by cloning the subsystem `Machine1`. The `clones` directive duplicates `Machine1`, its variables, subsystems and activities, and gives a new name to the clone, here `Machine2`.

Cloning is thus a very powerful mechanism, stemmed from prototype-oriented programming.

```
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;
        end
    end

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

Figure 9: 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
end

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

Figure 10: Code for the `ProduceProduct` of the producer instantiating of the class `Machine`

Cloning 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 activity `ProduceProduct` of `Machine1`, because with such declaration order, `Machine1` would be cloned without its activity.

6.2 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 10.

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 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.

```
system World.Producer
    ...
    Machine Machine1
        parameter Real lowerBoundProductionTime = 4;
        parameter Real upperBoundProductionTime = 6;
    end
    Machine Machine2
        parameter Real upperBoundProductionTime = 7;
    end
    ...
end
```

Figure 11: Instantiating the class `Machine` with redefining the value of a parameter

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

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

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

Figure 12: The class `Machine` inherits from the class `RepairableComponent`

6.3 Polymorphism and Inheritance

Parameters are an excellent way to make on-the-shelf, reusable, modeling components generic. In Σ^{TM}, parameters can be defined with a default value and redefined when the class is instantiated. Assume for instance that the duration of the activity `ProduceProduct` of our machines obey a uniform distribution between two bounds. It is then recommended to define these bounds as parameters. Their value can then be changed as illustrated in Figure 11.

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 as a "is-part-of" relation, inheritance can be seen as a "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^{TM} this operation is performed by the `extends` directive. Figure 12 illustrates this mechanism.

```
domain UnitState {STANDBY, WORKING}

class Unit
    UnitState state(init = STANDBY);
    ...
end

system Main
    ...
    Unit A;
    Unit B;
    attribute A.state.init = WORKING;
    ...
end
```

Figure 13: Redeclaration of the attribute `init` of the variable `state` of the subsystem `A`

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.

6.4 Attribute (re)Declaration

Σ™ makes it possible to (re)declare attributes. Figure 13 illustrates this mechanism. By default, units are initially in standby. We want however that the unit `A` to be initially working. This is achieved by redeclaring its attribute `init`, as show in the figure.

6.5 Splitting the Model into Several Files

Σ™ makes it possible to split a model into several files. This is either done at tool level, using the notion of *Project* in the Sigma Workshop, or by means of the `import` directive. The latter works as follows.

```
import "Components/Valve.sigma"
```

To get more information about the Sigma Workshop, visit:
`../SigmaWizardUserManual/SigmaWizardUserManual.html`

7 More on Activities

This section digs further in the semantics of Σ™ regarding the activities.

7.1 Vocabulary

When the trigger of an activity returns `true`, one says that it is *satisfied* and that the activity is *enabled*. If the trigger returns `false`, one says that it is *falsified* and that the activity is *disabled*.

When an activity gets enabled at time t, its start is *scheduled* at t. When the activity is actually started, its completion is scheduled at time $t + d$, where d is the value returned by the clause `duration` of the activity.

```
system WorldChecker
    parameter int worldValue = 42;
    activity CheckWorld
        trigger: return worldValue>0;
        start: skip;
        completion: skip;
        duration: return 1;
    end
end
```

Figure 14: Trigger without variables

```
system UndecidedGuy
    bool available(init = true);
    activity DoSomething
        trigger: return available or not available;
        start: available = not available;
        completion: available = not available;
        duration: return 1;
    end
end
```

Figure 15: Repetitive starts

Events, i.e. starts and completions of activities as well as *observations* that are scheduled at the least time are called *candidate events*. By abuse of generality, one says that an activity is *candidate* if either its start or its completion are candidate.

7.2 Triggers

The trigger of an activity is checked each time the state of the system is modified and this modification impacts the trigger. It is assumed that the initialization of the state of system impacts all of triggers of the system.

7.2.1 Triggers without Variables

This leads to a first subtlety. Consider the model given in Figure 14.

The trigger of the activity `CheckWorld` returns `true`. It is thus satisfied. However, it involves no variable. Consequently, it is never impacted by a change of a state of the system, which means in turn that it is never enabled, but right after the initialization.

The activity `CheckWorld` is thus executed only once, starting at time $t = 0$ and completed at time $t = 1$.

7.2.2 Repetitive Starts

Now consider the model given in Figure 15.

Now the trigger of the activity `DoSomething` involves the variable `available`. However, it is a tautology, i.e. returns always `true`.

The activity `DoSomething` is thus started at time $t = 0$. Its completion is scheduled at time $t = 1$. So far, so good.

```
system Agent
    int remainingJobs(init = 3);
    activity TakeJob
        trigger: return remainingJobs>0;
        start: remainingJobs -= 1;
        completion: skip;
        duration: return 1;
    end
end
```

Figure 16: A zealous agent

The problem here is that its start modifies the state of the system and impacts . . . the trigger of the activity `DoSomething`. This trigger is thus evaluated and found true. Consequently, a new instance of the activity `DoSomething` is thus started, still at $t = 0$.

The completion of this new instance is scheduled at time $t = 1$. Its start creates modifies the state of the system. . .

The system enters into an infinite loop and is stuck at time $t = 0$.

7.2.3 A Zealous Agent

The two previous examples are arguably stupid. The one given in Figure 16 is more interesting.

At start, it remains 3 jobs to be performed by the agent. As the trigger of the activity `TakeJob` is satisfied, this activity is started at time $t = 0$. Its completion is scheduled at time $t = 1$.

The start of the activity modifies the state of the system, which impacts the trigger of the activity `TakeJob`. A new instance of this activity is thus started. And so on until it remains no available job.

The 3 jobs are performed in parallel and completed at time $t = 1$. Our agent is thus especially zealous.

Now consider the model given in Figure 16 in which the actions at start and at completion are switched:

```
start: skip;
completion: remainingJobs -= 1;
```

In this case, the activity `TakeJob` is started at $t = 0$. As the state of the system is not modified by the action at start, no new activity is started at time $t = 0$. The activity `TakeJob` is thus completed at time $t = 1$, which modifies the state of the system and starts a new instance of the activity. And so on. The 3 jobs are thus completed at time $t = 3$. Our zealous agent became lazy.

7.3 Conflicts

7.3.1 Alice versus Bob

Some activities may be in competition for a resource, thereby creating *conflicts*. As an illustration, consider the following case.

Alice and Bob work at the packaging station in a production line. Their job consists in taking a product in the input buffer of the station, packaging it and deliver it into the output buffer of the station. Alice and Bob work at the same place and there is no established rule on

```
domain WorkerState {STANDBY, WORKING}

system ProductionLine
    system InputBuffer
        int products(init = 3);
    end
    system PackagingStation
        system Alice
            int products(init = 0);
            WorkerState state(init = STANDBY);
            activity PackageProduct
                trigger:
                    return main.InputBuffer.products>0 and state==
                        ↪ STANDBY;
                start: {
                    main.InputBuffer.products -= 1;
                    products += 1;
                    state = WORKING;
                    }
                completion: {
                    state = STANDBY;
                    products -= 1;
                    main.OutputBuffer.products += 1;
                    }
                duration:
                    return 2;
            end
        end
        clones Alice as Bob;
    end
    system OutputBuffer
        int products(init = 0);
    end
end
```

Figure 17: Conflict at the workstation

who takes a product in the input buffer first, in case both of them are ready to work on a new product at the same time.

A code representing this situation is given in Figure 17.

At time $t = 0$, the activities `PackageProduct` of both `Alice` and `Bob` are enabled. Both are thus started, no matter in which order. Two products are removed from the input buffer.

At time $t = 2$, both activities are completed. Still at time $t = 2$, both activities are enabled again. There is a problem however: it remains only one product in the input buffer. Consequently, only can actually start.

In such a situation, the semantics of Σ^{TM} consists in drawing at random one of the candidate activities, and to start it. All candidate activities have the same probability to be selected. Before the selected activity is started, its trigger is checked again. If it is falsified, the activity is simply abandoned. Otherwise, it is actually started.

Assume in our example that the activity `PackageProduct` of `Alice` is selected. Its trigger is checked. It is satisfied, consequently the activity is started. The remaining product in input buffer is removed from the latter. Now, all candidate activities are considered again. It remains

```
system ProductionLine
    system InputBuffer
        int products(init = 3);
    end
    system PackagingStation
        int products(init = 0);
        int availableWorkers(init = 2);
        activity PackageProduct
            trigger:
                return main.InputBuffer.products>0 and availableWorkers
                    ↪ >0
            start: {
                main.InputBuffer.products -= 1;
                products += 1;
                availableWorkers -= 1;
                }
            completion: {
                availableWorkers += 1;
                products -= 1;
                main.OutputBuffer.products += 1;
                }
            duration:
                return 2;
        end
    end
    system OutputBuffer
        int products(init = 0);
    end
end
```

Figure 18: A model allowing multiple instances of an activity

only one, namely the activity `PackageProduct` of `Bob`. It is thus selected. Its trigger is falsified, consequently the activity is abandoned.

This semantics introduces an element of non-determinism in otherwise fully deterministic models. It is however the only elegant and efficient way of dealing with conflicts among activities.

7.3.2 Multiple Instances

Another way to represent the production line of the previous section is to consider that there is a pool of indistinguishable workers (with two workers in that case). A possible code for this approach is given in Figure 18.

At time $t = 0$, the activity `PackageProduct` is enabled. The activity is thus started. Its completion is scheduled at time $t = 2$.

As its instruction at start modifies the state of the system and impacts the trigger of the activity `PackageProduct`, the latter is evaluated and found satisfied. A second instance of the activity `PackageProduct` is thus started. Its completion is scheduled at time $t = 2$, as the first one.

As the instruction at start modifies the state of the system and impacts the trigger of the activity `PackageProduct`, the latter is evaluated again. But this time, it is falsified as there is no more available workers.

A time $t = 2$, one of the instance of the activity `PackageProduct` is completed (af-

```
system Producer
    bool available(init = true);
    int quantityToProduce(init = 0);
    int totalProduction(init = 0);
    activity Produce
        trigger:
            return available;
        start: {
            available = false;
            quantityToProduce = rangeDeviate(10, 20);
            }
        completion: {
            available = true;
            totalProduction += quantityToProduce;
            }
        duration:
            return 2*quantityToProduce;
    end
end
```

Figure 19: Calculating the duration in the action at start

ter a random selection), which modifies the state of the system. The trigger of the activity `PackageProduct` is thus evaluated and found satisfied. A new instance of this activity is thus started at time $t = 2$. Its completion is scheduled at time $t = 4$.

We have now two events scheduled: the completion remaining of the instance of the activity `PackageProduct` started at time $t = 0$, at the start of the instance that just scheduled. One of these events is thus selected at random and executed. No matter which one is executed, this modifies the state of the system and the trigger of the activity `PackageProduct`. However, the latter is now falsified as there is no more products in the input buffer.

7.4 Action at Start versus Duration

When an activity is launched, two instructions (aside the assessment of its trigger) are executed in order:

1) Its action at start;

2) The calculation of its duration.

This makes it possible to use the action at start to determine the duration.

As an illustration, consider a producer who is periodically ordered a certain quantity to produce. Assume that this quantity is not always the same, but obeys a some known distribution. Assume moreover that the duration of the production depends linearly on this quantity to produce.

A possible model for this case study is given in Figure 19.

The variable `quantityToProduce`, which is set in the action at start, is used to calculate the duration.

8 More on Variables and Expressions

This section digs further in the use of variables in Σ^{TM}.

8.1 The Three Categories of Variables

8.1.1 Vocabulary

Variables can be declared at three different places in Σ™ models: in systems, in activities and in blocks of instructions. Their *life-cycle* and their *scope* differ in each case:

- Variables declared in systems are reachable from everywhere in the model. Moreover, they are reachable from the time they are created to the time they are deleted, both directly or indirectly (via their parent system). The current version of Σ™ does not implement yet directive to create, delete or move objects dynamically. Consequently, variables declared in systems are always reachable.
- Variables declared in activities (outside of the clauses `trigger`, `start`, `completion` and `duration` are created when an instance of the activity is started. They are reachable only in clauses `start`, `completion` and `duration`. They are deleted when the instance is completed.
- Finally, variables declared in blocks instructions are reachable only in the instructions of the block located after their declaration.

We call variables of each of the above categories respectively *system variable*, *activity variable*, and *instruction variable*.

8.1.2 Example

To illustrate the differences among these categories of variables, consider a storage unit that stores initially a certain number of products, and that serves one after the other its clients, i.e. that it delivers them the quantity of products the client demands. Indeed, it cannot deliver more products than it has in stock. Assume moreover, that the numbers of products the clients demand obey some random distribution.

The model given in Figure 20 is a possible implementation of such a unit.

The system `StorageUnit` declares two system variables: `storedProducts` and `deliveredProducts`. The activity `DeliverProducts` declares one activity variable, `deliveredQuantity`. Finally, the clause `start` of this activity declares one instruction variable `expectedQuantity`.

Variables `storedProducts` and `deliveredProducts` exist as long as the system `StorageUnit` exists, i.e. during the whole mission time.

Each time a new instance of the activity `DeliverProducts` is started, a variable `deliveredQuantity` attached to this instance is created. This means that if several instances of the activity `DeliverProducts` exist simultaneously, there exist several copies of the variable `deliveredQuantity`. This is not a problem because the variable is visible only in the activity. In a way, this is similar to call by the same name two variables declared by two different systems.

The variable `deliveredQuantity` attached to an instance of the activity `DeliverProducts` is created at the beginning of its action at start, i.e. just before the creation of the instruction variable `expectedQuantity`. It is deleted at the end of its action at completion, i.e. after the instruction:

```
deliveredProducts += deliveredQuantity;
```

The variable `expectedQuantity` is created with the instruction:

```
int expectedQuantity;
```

```
system StorageUnit
    int storedProducts(init=10);
    int deliveredProducts(init=0);
    activity DeliverProducts
        int deliveredQuantity;
        trigger:
            return storedProducts>0;
        start: {
            int expectedQuantity;
            expectedQuantity = rangeDeviate(2, 6);
            deliveredQuantity = min(storedProducts, expectedQuantity);
            storedProducts -= deliveredQuantity;
            }
        completion:
            deliveredProducts += deliveredQuantity;
        duration:
            return 2*deliveredQuantity;
    end
end
```

Figure 20: Trigger without variables

and deleted after the instruction:

```
storedProducts -= deliveredQuantity;
```

⚠ Activity and instruction variables do not have initial values. Consequently, they must be assigned before being referred to.

8.1.3 Execution

A possible execution of the model given in Figure 20 could be as follows.

At time $t = 0$, the activity `DeliverProducts` is enabled. A first instance of this activity is thus started, which creates a variable `deliveredQuantity` associated with this instance. Assume that the instruction variable `expectedQuantity` receives the value 3. At the end of the action at start of this first instance of the activity `DeliverProducts`, the activity variable `deliveredQuantity` has the value 3 and the system variable `storedProducts` has the value $10 - 3 = 7$. The completion of this first instance of the activity is scheduled at time $t = 2 \times 3 = 6$.

The activity `DeliverProducts` is still enabled. A second instance of this activity is thus started (still at time $t = 0$), under the same conditions as the previous one. Assume that this time the instruction variable `expectedQuantity` receives the value 5. At the end of the action at start of this second instance of the activity `DeliverProducts`, the activity variable `deliveredQuantity` has the value 5 and the system variable `storedProducts` has the value $7 - 5 = 2$. The completion of this second instance of the activity is scheduled at time $t = 2 \times 5 = 10$.

The activity `DeliverProducts` is still enabled. A third instance of this activity is thus started (still at time $t = 0$), under the same conditions as the two previous ones. Assume that this time the instruction variable `expectedQuantity` receives the value 4. At the end of the action at start of this second instance of the activity `DeliverProducts`, the activity variable `deliveredQuantity` has the value $min(2, 4) = 2$ and the system variable `storedProducts`

```
class WindMill
    ...
    activity Start
        trigger:
            return main.windForce>10 and owner.windForce<25
        ...
    end
    ...
end

system WindFarm
    float windForce(init=0);
    ...
    WindMill WM01;
    ...
end
```

Figure 21: Incorrect references in classes

has the value $2 - 2 = 0$. The completion of this third instance of the activity is scheduled at time $t = 2 \times 2 = 4$.

Now, the activity `DeliverProducts` is disabled. The time can eventually elapse.

The next events are in order:

- The completion of the third instance of the activity `DeliverProducts`, at time $t = 4$;
- The completion of the first instance of the activity `DeliverProducts`, at time $t = 6$;
- Finally, the completion of the second instance of the activity `DeliverProducts`, at time $t = 10$.

8.2 Classes and Paths

Just as systems, classes are name spaces for variables. What can be referred to from a class is however restricted compared to what can be referred from a system in the main hierarchy. Classes can actually be reused in any context. Consequently, no "bet" can be taken on what the context in which a class will be instantiated is.

The (excerpt of a) model of a wind farm given in Figure 21 illustrates this problem.

As there are many windmills in a wind farm, it is a good idea to declare a class for windmills. Now, a windmill can be started only if the force of the wind is above a certain threshold and below an another one. Although local variations must probably be considered, the force of the wind is probably not a property of a particular windmill. It is rather common to all windmills of the wind farm. It is thus tempting, in the class `WindMill` to refer to the variable `windForce` of the main system `WindFarm`. This is what the two references `main.windForce` and `owner.windForce` are doing. This is however not allowed as it muts be possible to instantiate the class `WindMill` in any context, including in a context in which the parent of the instance of `WindMill` does not declare a variable `windForce`.

The Σ^{TM} compiler may not reject a model such as the one of Figure 21. Nevertheless, it is certainly not a good practice to count on that!

A workaround consists in declaring a variable `windForce` local to the class `WindMill` and an activity `UpdateWindForces` in the system `WindFarm` that updates the local `windForce` by means of the global one.

References

Abadi, Mauricio and Luca Cardelli (1998). *A Theory of Objects*. New-York, USA: Springer-Verlag. ISBN: 978-0387947754 (cited on pages 33, 36).

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 6, 11, 33, 35).

— (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 6, 18, 33).

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 page 6).

deWeck, 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 6).

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 35).

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 33).

Rauzy, Antoine (2020). *Probabilistic Safety Analysis with XFTA*. Les Essarts le Roi, France: AltaRica Association. ISBN: 978-82-692273-0-7 (cited on page 33).

— (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 33, 35).

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 6, 33, 35).

Rauzy, Antoine B. (2023). Σ™*W*ORKSHOP *User Manual*. Reference Manual SIG-RM-2023-004. Systemic Intelligence. URL: ../SigmaWorkshopUserManual/SigmaWorkshopUserManual.html (cited on pages 18, 19).

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 6).

Index

Table 12: EBNF constructs

Expression	Meaning
`S ::= E`	The non-terminal symbol `S` is defined by the expression `E`
`E F`	`E` followed `F`
`E | F`	`E` or `F`
`E*`	Any number of `E`
`E+`	Any positive number of `E`
`E?`	0 or 1 `E`
`(E)`	Grouping
`'...'`	Terminal symbol

A Grammar

A.1 Extended Backus-Naur Form

This section presents EBNF grammar of Sigma.

Table 12 recalls the EBNF constructs and their meaning.

A.2 Models and Systems

The grammar of models, classes and systems is as follows.

```
Model ::=
    Declaration*

Declaration ::=
      DomainDeclaration
    | ClassDeclaration
    | SystemDeclaration
    | ActivityDeclaration
    | ImportDirective

SystemDeclaration ::=
    'system' Path ObjectDeclaration* 'end'

ObjectDeclaration ::=
      ConstantDeclaration
    | ParameterDeclaration
    | VariableDeclaration
    | ObserverDeclaration
    | IndicatorDeclaration
    | SystemDeclaration
    | InstanceDeclaration
    | ActivityDeclaration
    | ExtendsDirective
    | ClonesDirective
    | AttributeDeclaration
```

A.3 Variables

A.3.1 Basic Types and Domains

The grammar of *basic types* and *domains* is as follows.

```
Type ::= BasicType | Domain

BasicType ::= 'bool' | 'int' | 'float' | 'id' | 'str'

DomainDeclaration ::= 'domain' Identifier SymbolicConstantSet
SymbolicConstantSet ::= '{' SymbolicConstants '}'
SymbolicConstants ::= SymbolicConstant (',' SymbolicConstant)*

Domain ::= Identifier
SymbolicConstant ::= Identifier
```

A.3.2 State and Temporary Variables

The grammar of declarations of *state and temporary variables* is as follows.

```
VariableDeclaration ::=
    Type Path AttributeList? ';'

AttributeList ::= '(' Attributes ')'
Attributes ::= Attribute (',' Attribute)*
Attribute ::= Identifier '=' Expression
```

Attributes for variables are the following.

- `init`, mandatory for state variables and optional for temporary ones.
- `unit`, optional for all variables.

A.3.3 Constants and Parameters

The grammar of declarations of *constants* and *parameters* are as follows.

```
ConstantDeclaration ::=
    'constant' Type Path AttributeList? '=' Expression ';'

ParameterDeclaration ::=
    'parameter' Type Path AttributeList? '=' Expression ';'
```

Attributes for constants and parameters are the following.

- `unit`, optional.

A.3.4 Observers

The grammar of declarations of *observers* is as follows.

```
ObserverDeclaration ::=
    'observer' Path AttributeList? '=' Expression ';'
```

Attributes for observers are the following.

- `unit`, optional.

A.3.5 Indicators

The grammar of declarations of *indicators* is as follows.

```
IndicatorDeclaration ::=
    'indicator' Path AttributeList?
    '=' MeasuredQuantity '(' Expression ')' ';'

MeasuredQuantity ::=
      'value'
    | 'min' | 'max' | 'mean' | 'sum'
    | 'firstChangeTime' | 'numberOfChanges'
```

Attributes for indicators are the following.

- `mean`, Boolean.
- `standardDeviation`, Boolean.
- `confidenceRange90`, Boolean.
- `confidenceRange95`, Boolean.
- `confidenceRange99`, Boolean.
- `min`, Boolean.
- `max`, Boolean.
- `bins`, integer.
- `shrinkFactor`, integer.

A.4 Expressions

The grammar of expressions is as follows.

```
Expression ::=
      VariableReference
    | BooleanExpression
    | Inequality
    | ArithmeticExpression
    | BuiltInExpression
    | ConditionalExpression
    | ProbabilityDistribution
    | RandomDeviate
    | TimePrimitive

Condition ::=
      VariableReference // if type bool
    | BooleanExpression
    | Inequality
    | ConditionalExpression // if return type bool
```

A.4.1 Constants

```
Constant ::= Boolean | Integer | Float | Symbol |

Boolean ::=  'true' | 'false'
Integer ::= [0-9]+
Float ::= ([0-9]*[.])?[0-9]+([eE][-+]?[0-9]+)?
Symbol ::= Identifier
```

A.4.2 Identifiers and Paths

```
Identifier ::= [_A-Za-z][_A-Za-z0-9]+
Path ::=
      Identifier
    | Identifier '.' Path
    | 'main' '.' Path
    | 'owner' '.' Path
```

A.4.3 Boolean Expressions

```
BooleanExpression ::=
        Expression ('or' Expression)*
    |   Expression ('and' Expression)*
    |   'not' Expression
```

A.4.4 Inequalities

```
Inequality ::= Expression InequalitySymbol Expression

InequalitySymbol ::= '==' | '!=' | '<' |  '<=' | '>' | '>='
```

A.4.5 Arithmetic Operations

```
ArithmeticExpression ::=
        Expression ('+' Expression)*
    |   Expression '-' Expression
    |   Expression ('*' Expression)*
    |   Expression '/' Expression
    |   Expression 'div' Expression
    |   Expression 'mod' Expression
    |   '-' Expression
```

A.4.6 Builtin Expressions

```
BuiltIn ::= UnaryBuiltIn | BinaryBuiltIn | AssociativeBuilIn

UnaryBuiltIn ::= UnaryBuiltInSymbol '(' Expression ')'
UnaryBuiltInSymbol ::=
```

```
    'abs' | 'exp' | 'log' | 'log10' | 'sqrt' | 'ceil' | 'floor'
  | 'acos' | 'asin' | 'atan' | 'cos' | 'sin' | 'tan'
  | 'bool' | 'int' | 'float' | 'symbol' | 'string'

BinaryBuiltIn ::= BinaryBuiltInSymbol '(' Expression ',' Expression ')'
BinaryBuiltInSymbol ::= 'pow'

AssociativeBuiltIn ::=
    AssociativeBuiltInSymbol '(' Expression (',' Expression)* ')'
AssociativeBuiltInSymbol ::= 'min' | 'max'
```

A.4.7 Count Expressions

```
CountExpression ::= '#' '(' Expression ( ',' Expression )* ')'
```

A.4.8 Conditional Expressions

```
ConditionalExpression ::= IfThenElse

IfThenElse ::= 'if' Condition 'then' Expression 'else' Expression
```

A.4.9 Probability Distribution and Random Deviate

```
ProbabilityDistribution ::=
    ParametricProbabilityDistribution | EmpiricalProbabilityDistribution

ParametricProbabilityDistribution ::=
      'exponentialDistribution' '(' Expression ',' Expression ')'
    | 'WeibullDistribution' '(' Expression ',' Expression ',' Expression
      ↪ ')'
    | 'DiracDistribution' '(' Expression ',' Expression ')'

EmpiricalDistribution ::=
    'empiricalDistribution' '(' FileName ',' Interpolation ')'

Interpolation ::=
    'step' | 'linear'

String = "[^"]+"
```

```
RandomDeviate ::=
    ParametricRandomDeviate | EmpiricalRandomDeviate

ParametricRandomdeviate ::=
      'uniformDeviate' '(' Expression ',' Expression ')'
    | 'triangularDeviate' '(' Expression ',' Expression ',' Expression
      ↪ ')'
    | 'normalDeviate' '(' Expression ',' Expression ')'
    | 'lognormalDeviate' '(' Expression ',' Expression ')'
    | 'exponentialDeviate' '(' Expression ')'
    | 'WeibullDeviate' '(' Expression ',' Expression ')'
```

```
    | 'rangeDeviate' '(' Expression ',' Expression ')'

EmpiricalDeviate ::=
    'empiricalDeviate' '(' FileName ',' Interpolation ')'
```

A.4.10 Time Primitives

```
TimePrimitive ::=
      'currentTime' '(' ')'
    | 'missionTime' '(' ')'
```

A.5 Instructions

The grammar of instructions is as follows.

```
Instruction ::=
      Skip | Assignment | IfThenElse | While
    | Return | InstructionBlock

Skip ::=
    'skip' ';'

Assignment ::=
    Variable ':=' Expression ';'

IfThenElse ::=
    'if' BooleanExpression
    'then' Instruction
    ('else' Instruction)?

While ::=
    'while' BooleanExpression Instruction

Return ::=
    'return' Expression ';'

InstructionBlock ::=
    '{' Instruction+ '}'
```

A.6 Activities

The grammar of activities is as follows.

```
ActivityDeclaration ::=
    'activity'
        VariableDeclaration*
        TriggerClause
        StartClause
        CompletionClause
        DurationClause
    'end'

TriggerClause ::=
```

```
    'trigger' ':' Instruction

StartClause ::=
    'start' ':' Instruction

CompletionClause ::=
    'completion' ':' Instruction

DurationClause ::=
    'duration' ':' Instruction
```

A.7 S2ML Constructs

The grammar of S2ML constructs is as follows.

```
ClassDeclaration ::=
    'class' Identifier ObjectDeclaration* 'end'

InstanceDeclaration ::=
    Identifier Path InstanceDeclarationBody

InstanceDeclarationBody ::=
      (ObjectDeclaration+ 'end')
    | ';'

ExtendsDirective ::=
    'extends' Path ';'

ClonesDirective ::=
    'clones' Path 'as' Path InstanceDeclarationBody

AttributeDeclaration ::=
    'attribute' Path '=' Expression ';'

ImportDirective ::=
    'import' String ';'
```