System Structure Modeling Language (S2ML)

Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy

System Structure Modeling Language (S2ML)

Language Specification – Version 1.0
November 2015

Abstract: This document defines the S2ML language, version 1.0. S2ML is developed in the framework of the OpenAltaRica project, leaded by IRT SystemX, Palaiseau, France. S2ML is a freely available, prototype-oriented language for both representing the structure complex systems and structuring models of these systems. It aims at providing a minimal but sufficient set of constructs for these purposes.

Authors: Michel Batteux, Tatiana Prosvirnova and Antoine Rauzy
Affiliation: AltaRica Association
Version: 1
Date: November 2015

Preface

The complexity of industrial systems is steadily increasing. To face this complexity the different engineering disciplines have virtualized their contents to a large extent. The design and operation of any modern system involve dozens if not hundreds of models. As systems are complex, one cannot expect models of these systems to be simple. They too are complex. Therefore, they too need to be organized and structured.

Designing models has much to do with designing computer programs. To start with, to model, one needs a modeling language; just as to program, one needs a programming language. In 1976, Niklaus Wirth, one of the pioneer of computer programming, wrote a famous book entitled “Algorithms + Data Structures = Programs”. This sentence can be revisited and applied to models:

Behaviors + Structures = Models

Most of modeling languages are actually the combination of a mathematical framework, e.g. differential equations (as for Modelica) or Mealy machines (as for Lustre), and constructs to structure models, e.g. a tree-like hierarchy of components.

The structure of a model is in general close to the functional or physical architecture of the system it describes. There is however no such a thing as a one-to-one correspondence between system and model structures, for at least three reasons. First, a model is always a specific point of view on the system. It aims at capturing a particular aspect or to evaluate a particular property of that system. Therefore, it abstracts away irrelevant parts of the system and is useful because it does so. Second, it is often necessary to model not only the system, but also its environment. Therefore, the frontiers of the system and the model are not always the same. Third, the organization of a model has its own logic and its own constraints that can be quite different from those of the architecture of a (physical) system. The choice of the right mathematical framework is of paramount importance to capture this or that physical aspect of the system, and to do it at the suitable level of abstraction. It turns out that the productivity of the modeling process stands also in the way models are structured. It is actually thanks to their structures that models can be versioned and configured throughout their life-cycle, that models of different engineering disciplines contributing to the design of a system can be synchronized, and that knowledge can be capitalized from projects to projects.

Constructs to structure models are derived from those to structure computer programs. The main programming paradigms, such as object-oriented programming, or functional programming have their modeling counterparts. In programming, the choice of a paradigm is to some extent a matter of taste, although object-oriented programming is nowadays dominant in industrial practice. The situation is rather different in the case of models. First, their structuring paradigm is more independent of their behavioral content than in the case of computer programs. Second, prototype-orientation seems the best suited paradigm for models for it corresponds to their level of abstraction.

System Structure Modeling Language (S2ML) is a small prototype-oriented system architecture language. S2ML aims at providing a minimal yet sufficient set of constructs for two different purposes.

First, S2ML can be used to represent the structure of systems (and their environments). It is worth noting that the structure described by the S2ML model does not necessarily pre-exists to the design

of that model. With that respect, designing a S2ML model contributes to architect the system under study.

Second, S2ML can be used as a paradigm to structure models. Ideally, any system modeling language could be designed by adding S2ML on top of the relevant mathematical framework. In practice, such a perspective is probably much too ambitious because language designers follow their own path and more importantly because existing modeling languages have their own structuring constructs. These constructs may differ from those provided by S2ML. S2ML can however be useful in two different ways. First, it can contribute to increase the awareness of the system engineering community on the topics of model structuring. It can inspire system modeling language designers and provide them with conceptual tools to ensure the convergence of the structuring part of modeling languages. Second, it can be used directly to support the model synchronization process, i.e. the process by which one can ensure that two possibly heterogeneous models are "speaking" about the same system. Two models, written possibly into two different languages, cannot in general be compared directly. The idea, illustrated Figure 1, is thus to abstract them away into a common third language, here S2ML, and to compare their abstractions.

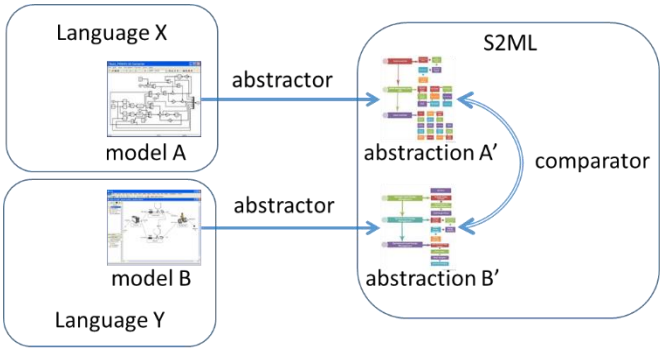

Figure 1. Model Synchronization.

The present document describes the version 1.0 of S2ML. It specifies a textual syntax as well as a XML representation for S2ML. It proposes also some normalized graphical representations for S2ML models. These graphical representations should not however be confused with the model itself. Even though they are an excellent support for the communication amongst stakeholders, they are able to represent only partially the models, except for formalisms with very low (or very ambiguous) expressiveness. Moreover, there may be several graphical representations of the same concept, each more or less convenient depending on the context. In a word, concepts should come first. The graphical representations we shall use throughout this document are strongly inspired from SysML (more specifically from the so-called block definition diagrams and internal block diagrams of SysML) because this notation acquired recently a rather large popularity. It is therefore pointless to "reinvent the wheel".

S2ML is developed in the framework of the OpenAltaRica project, leaded by IRT SystemX, Palaiseau, France. S2ML is fully free of use. Although this document has been written by Michel Batteux, Tatiana Prosvirnova and Antoine Rauzy, it results of many interactions with academic colleagues, PhD students and industrial partners. They should be thanked here.

Table of Contents

Table of figures

1 Introduction

1.1 Scope of the Specification

System Structure Modeling Language (S2ML) is a small prototype-oriented system architecture language. S2ML aims at providing a minimal yet sufficient set of constructs for three different purposes.
First, S2ML can be used to represent the structure of systems (and their environments). Second, it can be seen as a paradigm to structure models. Third, S2ML can be used to abstract away heterogeneous models so to be able to compare, and eventually to synchronize them.
S2ML comes with a syntax, some graphical representations, and two sets of rules:

- The first one to transform an implicit S2ML model -- typically involving instantiations of classes and/or cloning of prototypes and/or substitutions -- into an explicit one (we shall give a precise meaning to these terms latter on). This transformation is called instantiation in the S2ML jargon.
- The second one to transform a hierarchical explicit model into a non-hierarchical one. This transformation is called flattening in the S2ML jargon.

The composition of these two transformations (in order) can be seen as the semantics of the language, although it lefts intentionally unspecified the meaning of non-hierarchical models.
The key issues of S2ML specification are thus:

- The definition of the different constructs of the language and their relationships, and
- The definition of sets of rules for expansion and flattening.

This specification does not draw however the full S2ML picture. Issues not addressed in the present document include:

- Best modeling practices, modeling patterns…;
- The definition of abstractors (from other modeling language) and comparators (of S2ML models).

1.2 Notations and Grammar

Throughout this document, grammar rules are given for the various S2ML constructs. The syntax for these grammar rules is a simplified version of the syntax used by parser generators such as ANTLR:

- Terminal symbols are either surrounded with quotes when they are given literally, e.g. “class”, “port”, or written in capital letters when they need to be defined, e.g. FLOAT, IDENTIFIER (denoting respectively floating-point numbers and identifiers).
- Non-terminal symbols are capitalized, e.g. ClassDeclaration. Their definition starts with the symbol “::=”.
- Symbols are grouped using parentheses “(“ and “)”.
- Alternatives are denoted with the symbol “|”.

- A symbol or a group of symbols followed by “*”, “+” or “?” denotes respectively any number of occurrences, at least one occurrence and at most one occurrence of that symbol or group of symbols.

For instance, the rule depicted Figure 2 describes a non-empty list of “Object” separated with commas. It means lists of objects is a unique object, followed by any number of times a comma then an object.

```
ObjectList ::=
        Object ( "," Object )*
```

Figure 2. Example of a rule.

Keywords are reserved words and should not be used as identifiers.

1.3 S2ML Code

For each construct of S2ML, we shall give a textual encoding as well as at least one possible graphical representation. The textual encoding is very close to a pure mathematical formulation. For this reason and in order not to introduce too many concepts, we shall not give the latter. Nevertheless, a pure mathematical formulation, abstracting away even textual “gadgets” like keywords, commas, brackets and the like, can be easily derived from the textual representation.
Throughout this document, S2ML code will be given using `Courier New` font and be surrounded with outside borders.

1.3.1 Character Set

The character set of S2ML is ASCII. Unicode characters are also authorized, but most of the S2ML tools do not accept this extension.
As in most of the modeling programming languages, any number of white spaces can be inserted between two lexical elements. White spaces include tabulations and end of lines.

1.3.2 Identifiers

S2ML Identifiers are sequences of letters, digits, and other characters such as underscore, which are used for naming various items in the language. They are of two forms:

- The first form (regular identifiers) always starts with a letter or an underscore “_”, followed by any number of letters, digits, or underscores. Case is significant, i.e., the names “Pump” and “pump” are different.
- The second form is for special identifiers (typically containing spaces). A special identifier is any sequence of characters surrounded with simple quotes, e.g.

  ‘this is a special identifier’

  Within a special identifier, simple quotes and anti-slashes must be escaped, e.g.

  ‘this is a \’special\’ identifier \\ with escaped characters \\’.

The use of special identifiers is a continuous source of problems. They should be avoided as much as possible. We shall not use them in the examples presented in this document.

1.3.3 Keywords

Keywords are reserved words in the S2ML grammar and therefore are not available as identifiers for user defined constructs. Throughout this document, keywords will be represented using **blue bold Courier New** font.

1.3.4 Comments

There are two kinds of comments in S2ML. Comments are not lexical units in the models and therefore are a priori ignored by S2ML parsers. The comment syntax is similar to that of C++. The following comment variants are available:

- // comment: Characters from // to the end of the line are ignored.
- /* … */ comments: Characters between /* and */ are ignored, including line terminators.

Throughout this document, comments will be written using orange/braun Courier New font.

1.4 Organization of the Document

The remainder of this document is organized as follow:

- Chapter 2 presents basic elements of S2ML.
- Chapter 3**Ошибка! Источник ссылки не найден.** introduces the notion of prototype and class.
- Chapter 4**Ошибка! Источник ссылки не найден.** shows how block aggregation is implemented in S2ML.
- Chapter 5 shows how polymorphism is implemented in S2ML.
- Chapter 6 is dedicated to the semantics of S2ML, i.e. to set of rules for expansion and flattening.

Appendices extend the specification with some additional material.

- Appendix A gives the complete BNF grammar of S2ML.
- Appendix B proposes a XML representation for S2ML.

2 Basic Elements of S2ML

This chapter introduces basic elements of S2ML: ports, connections, blocks, classes and attributes.

2.1 Blocks, Ports and Connections

2.1.1 Description

The block is the basic S2ML modeling unit. A block has a name and may contain ports and connections (and, as we shall see, other blocks and references to blocks). The three terms “block”, “port” and “connection” are borrowed to the SysML terminology, although they are used with a slightly different meaning. In the S2ML context, a port is something that has a name and that holds some atomic information. It is similar to variables of programming languages. A connection is a relation between ports.

Figure 3 gives a graphical representation of a block named “pump”. A block is usually represented as a rectangle with its name and possibly an icon (as on the left). On the figure, the block contains three ports named respectively “input”, “output” and “command” and represented by small black squares, usually located on the border of the rectangle. Connections are represented with bold plain (here green) lines (as on the right). Here there are two connections: one linking ports “command” and “state” and the other one linking ports “input”, “output” and “state”.

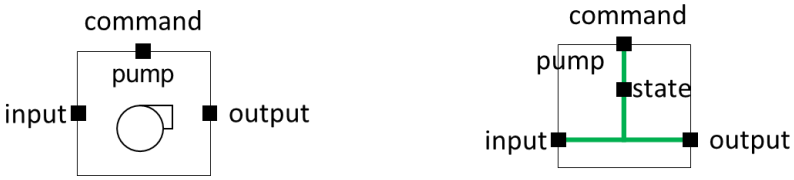

Figure 3. Graphical representations of a block “pump”.

A possible S2ML code for this block is given Figure 4. Keywords are in bold blue font. This code declares a block named “pump” which contains four ports named respectively “state”, “input”, “output” and “command” and two connections linking the ports “state”, “input” and “output” on the one hand and “state” and “command” on the other hand.

```
block pump
    port state, input, output, command;
    connection [input, state, output];   // flow equation
    connection [command, state];         // command action
end
```

Figure 4. Possible S2ML code for a block “pump”.

On this example, all the ports are declared within the same clause. It would be also possible to declare them separately or by subgroups, e.g.

```
    port input, output;
    port state;
    port command;
```

The same remark applies to connections (but in the reverse way), e.g.

```
    connection [input, state, output], [command, state];
```

The code of Figure 4 is richer than its (partial) graphical representation. Figure 3 (left) represents the interface of the component: the internal port “state” and the internal connections are not represented in order not to overload the graphics. Both the code and its graphical representation are useful. Nevertheless, the code is the reference.
S2ML identifiers (names) are those of most of the programming and modeling languages, i.e. a letter or an underline “_” character, followed by any number of letters, digits or underline “_” characters. This type of convention may seem restrictive (especially for Asian languages) but alleviate dramatically the tool development and model assessment.

2.1.2 Named Connections

Conversely to ports (and blocks), connections can be declared without associating them a name. The reason is that they are usually not referred to. In some cases however, they are and therefore need to be named. The name of a connection is given after the keyword “connection”. For instance, the declaration of connection for pump could be as follows.

```
    connection flowEquation [input, state, output];
    connection commandAction [command, state];
```

2.2 Attributes

S2ML provides a mean to associate data with blocks, ports and connections via the notion of attribute.
An attribute is a pair (name, value), where name is an identifier and value is any text (more technically a string).
For instance, the code given Figure 4 could be enriched as shown Figure 5.

```
block pump (note="to illustrate the notion of attribute")
    port input, output (type="Boolean");
    port state (type="on/off", init="off");
    port command (type="signal");
    connection [input, state, output]
               (assertion="output=(input and state=on)");
    connection [command, state]
               (action="on command do switch state");
end
```

Figure 5. An enriched version of the code for a block “pump”.

S2ML does not associate any meaning with values of attributes. Several attributes can be associated with a port, a block or a connection, as illustrated by the port “state”.

Representing attributes graphically tends to overload the drawings. Figure 6 shows a possible representation for attributes. Attributes are written in rectangles with one folded corner. Then the attribute box is attached to the modeling element it is associated with by means of a dashed line. In an integrated modeling and simulation environment, attribute boxes could be folded/unfold on demand.

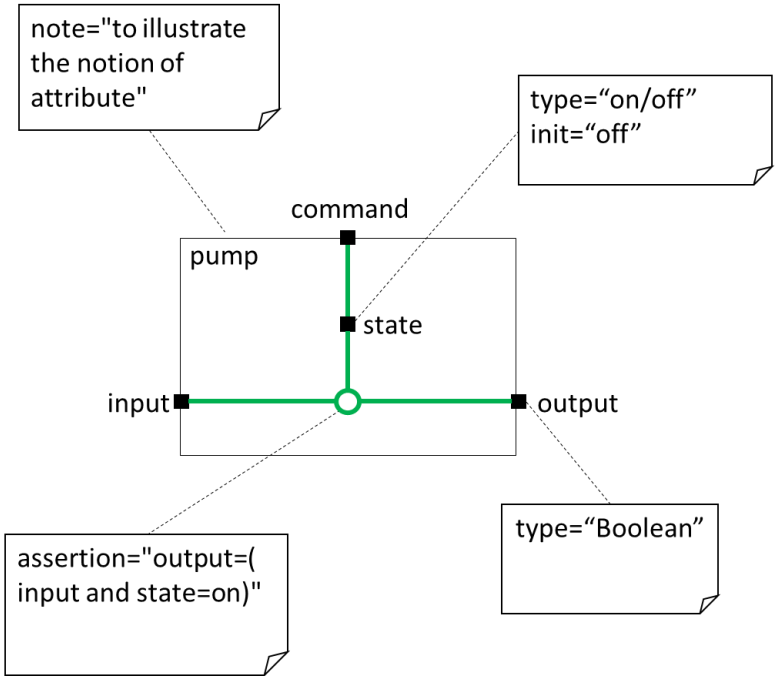

Figure 6. Graphical representation of attributes.

The small (green) circle that joins lines of the second connection has no specific meaning. It is just here as an help to visualize that the three ports “state”, “input” and “output” participate to the connection.

2.3 Composition

Blocks can contain other blocks. In object-oriented programming jargon, one says that the containing block composes the contained blocks. Composition is a fundamental operation in system modeling. Models can be built top-down by decomposing a system (or a function) into interacting sub-systems (sub-functions). They can also be built bottom-up by assembling components and describing their interactions.

2.3.1 Principle

In S2ML, to compose a block B within a block A, it suffices to declare B within the declaration of A, as illustrated Figure 7 where the block “solventSupply” composes the two blocks “tank” and “pump”.

Objects declared within the composed blocks can then be referred to using the dot "." notation: within the block "solventSupply", the port "output" of the tank is referred to as "tank.output", as illustrated in the assertion.
Names are local. It is thus possible to have a port named "output" in the block "tank" and a port also named "output" in the block "solventSupply".

```
block solventSupply
    block tank
        port level, output;
        connection [level, output];
    end
    block pump
        port state, input, output, command;
        connection [input, state, output];
        connection [command, state];
    end
    connection [tank.output, pump.input];
end
```

Figure 7. The S2ML for the composition of a block "tank" and a block "pump".

2.3.2 Graphical representations

Figure 8 shows a possible graphical representation for the block "solventSupply". This representation hides internals of contained blocks "tank" and "pump". It would be also possible to show, totally or partially, internal ports and connections of these blocks, but it would of course add some complexity to the drawing. This type of representation is close to Process & Instrumentation Diagrams, but also to those provided by modeling and simulation environments for languages such as Simulink, Modelica, Lustre, AltaRica and to some extent SysML internal block diagrams.

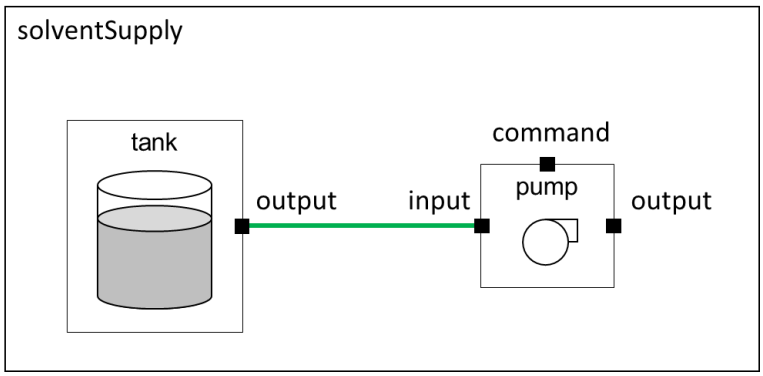

Figure 8. Graphical representation for the block "solventSupply".

The composition is in essence a "is-part-of" relation. The block "tank" is part of the block "solventSupply". The port "level" is part of the block "tank".
There is not a unique way to represent the "is-part-of" relation. Tree-like representations are also widely used to represent structure breakdowns. Figure 9 shows such a tree-like representation for the block "solventSupply". Each modeling element is represented by a rectangle. The "is-part-of"

relation is represented by a black filled diamond. This graphical element comes from UML/SysML. No diamond shows up under rectangles representing ports for nothing can be part of port.
The diamonds contain either the sign minus "-", when the block is expanded or the sign plus "+" when the block is folded. The blocks "solventSupply" and "tank" are expanded, while the block "pump" is folded. In a S2ML integrated modeling environment, it would be typically possible to fold and expand the representation of blocks by clicking on the diamond. Note that UML/SysML do not provide, or at least normalize, such folding/expansion mechanism.

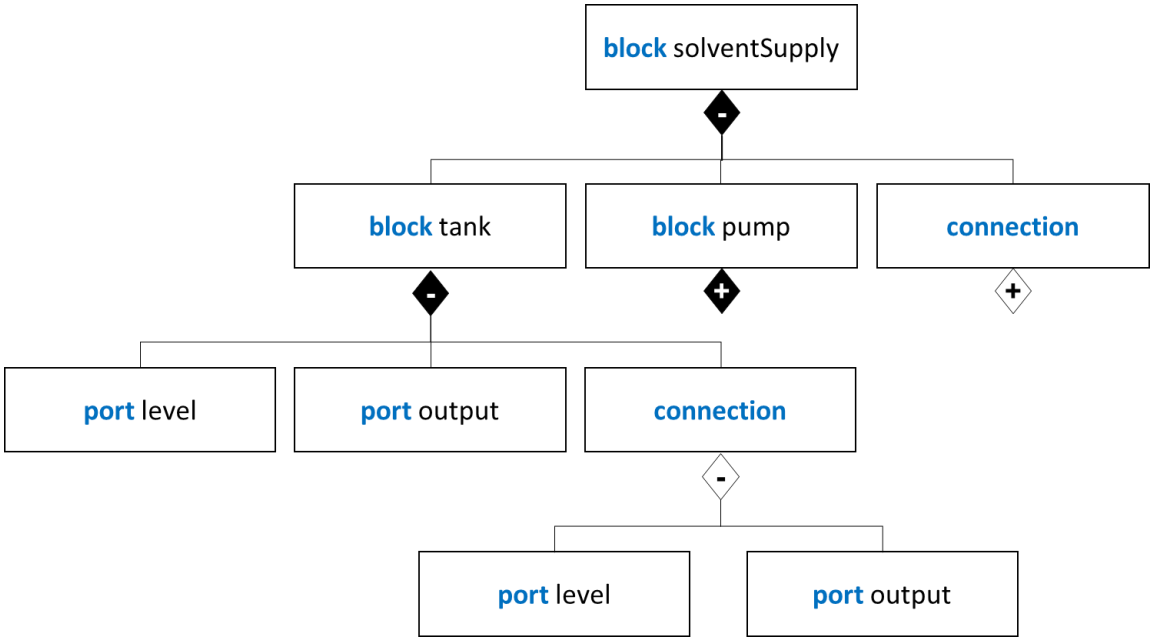

Figure 9. Tree-like representation for the block "solventSupply".

Strictly speaking, a connection does not contain the ports it connects. Rather, a connection refers to the ports it connects, or "uses" them. In the object-oriented programming jargon, this "uses" relation is called an aggregation. The connection aggregates ports "level" and "output" of the block "tank". In UML/SysML, aggregation is represented with a white filled diamond. S2ML obeys this convention, with the additional idea of adding a sign ("+" or "-") into the diamond so to indicate which elements are folded and which are expanded. Chapter 4 discusses the notion of aggregation, applied to blocks, in the S2ML context. In the diagram pictured Figure 9, ports "level" and "output" are repeated, first under the block "tank" that composes them and second under the connection that refers them. The experience shows that this solution (fold/expand mechanism plus repetition) is preferable to the solution that would consists in keeping only one copy of each repeated modeling element and drawing complicated and overlapping links from the parent elements.

It is worth noting that, conversely to planar representations (such as the one of Figure 8), tree-like representation can be easily automatically generated from the model (as for instance in Fault Tree tools). It is therefore possible to filter the representation so, for instance, to show only blocks and ports, or only modeling elements with a given attribute. It is also much easier to compare graphically two tree-like representations. One can for instance highlight elements showing up in the first tree but not in the second one with a first color, those showing up in the second tree but not in the first one with a second color, and finally those showing up in both trees with a third color.

Another version of tree-like representation (using the same graphical conventions) is shown Figure 10. A similar representation is used for instance in Windows explorer for files and directories. As the previous one, it can be automatically generated from the model and filtered at will. This one dimension representation may be less readable, but more convenient for model browsing purposes than the two dimensions representation of Figure 9.

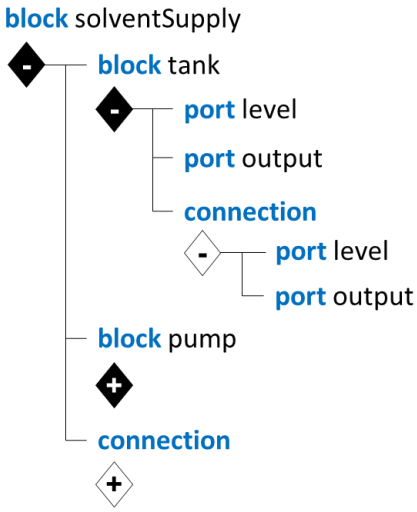

Figure 10. One dimension tree-like representation for the block "solventSupply".

2.3.3 Crossing the walls

In S2ML, it is possible to refer to ports of nested subsystems, as illustrated Figure 11. At the level of the block "chemicalPlant" it is possible to declare the following connection.

```
connection [controller.command, solventSupply.pump.command];
```

The port "command" of the block "solventSupply" can be referred to without the need to go through a port of this block. As pointed out by the small red arrow on the figure, the connection crosses the wall of block "solventSupply". The terms "port" and "connection" should be taken in a broad sense and not interpreted as representations of physical entities.

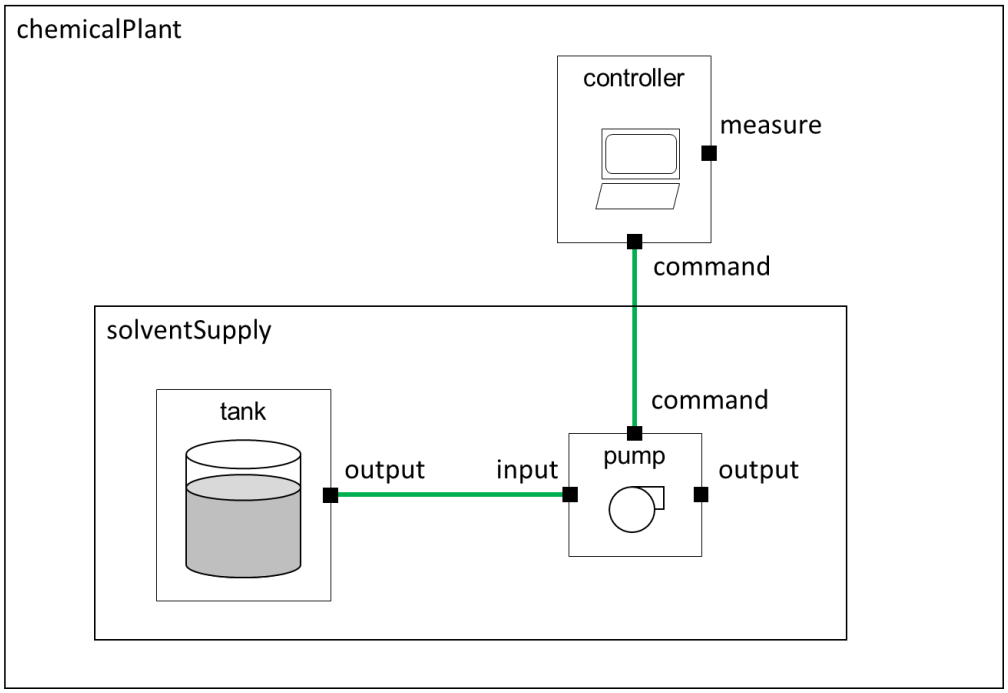

Figure 11. Crossing the walls.

2.3.4 Bringing down the walls

Although the structure of the model plays a very important role into its clarity, it normally plays no role in the behavioral description of the system. Any hierarchical model can be flattened into an equivalent non-hierarchical one.

The flattening process depends on the modeling language. It can usually be defined by means of so-called flattening rules that apply bottom-up. For S2ML, flattening rules are purely structural. This question is discussed in details Chapter 6.

In the case of composition, the flattening process just consists in bringing down the walls of nested blocks. As an illustration, consider again the code given Figure 7. Figure 12 shows an equivalent flattened code. Nested ports are declared by prefixing their name with the name of their (former) containing block. Similarly, nested connections are just copied by prefixing the references to ports with the name of their (former) containing block.

```
block solventSupply // flattened version
    port tank.level, tank.output;
    port pump.state, pump.input, pump.output, pump.command;
    connection [tank.level, tank.output];
    connection [pump.input, pump.state, pump.output];
    connection [pump.command, pump.state];
    connection [tank.output, pump.input];
end
```

Figure 12. Flattened version of the code given Figure 7.

2.4 Models and Files

In S2ML, files are not modeling entities: a file may describe several models and the same model can spread over several files.

The directive “include” makes it possible to incorporate the content of a file. For instance, the code for the class “Controller” (the notion of class will be discussed in Chapter 3) could be stored in a file “solventSupply.s2ml” and the code for the whole chemical plant is stored in a file “chemicalPlant.s2ml”. The inclusion of the former into the latter could be as illustrated Figure 13.

```
include "solventSupply.s2ml";
block chemicalPlant
    block controller extends Controller;
        …
    end
    …
end
```

Figure 13. Use of the “include” directive.

A S2ML model is a block. Therefore, a file may contain several models. These models are a fully separated. It is not possible, for instance, to create a connection between ports of separate models. It is up to assessment tools (and their command files) to decide which model to assess.

2.5 Names and Paths

In S2ML, names are used in two different ways: first, to declare the objects; second to refer to the objects. When an object is declared, it is associated with an identifier, its name. Two objects declared in the same block must have different names. The syntax for identifiers is the same as in most of modeling and programming languages, namely a letter or an underscore character “_” followed by any number of letters, digits or underscore characters (see Chapter 1 for more details).

An object is referred to by means of a path. Syntactically, a path is a sequence of identifiers separated with dots “.”. We shall illustrate the notion of path by means of the abstract hierarchical model pictured Figure 14.

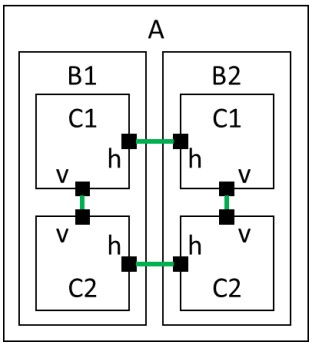

Figure 14. An abstract hierarchical model.

The system “A” represented Figure 14 is made of two similar subsystems “B1” and “B2”, themselves made of two similar subsystems “C1” and “C2”. Each block “Ci” has two ports “h” and “v”. The port

“v” of subsystems “C1” and “C2” are connected together, while the ports “h” are connected with their alter ego in the other “B” block.

A rather natural description for that system would be to declare connections between “v” ports in “B” blocks and connections between “h” ports into the “A” block, as illustrated Figure 15. Within the block “A”, the path “B1.C1.v” refers to the port “v” of the sub-block “C1” of the sub-block “B1”. Similarly, within the block “B1” (or “A.B1” if one will), “C1.h” refers to the port “h” of the sub-block “B1”.
Another way to describe this system could be to declare all of the connections in the block “A” as follows.

```
block A
    …
    connection [B1.C1.v, B1.C2.v];
    connection [B2.C1.v, B2.C2.v];
    connection [B1.C1.h, B2.C1.h];
    connection [B1.C2.h, B2.C2.h];
end
block A
    block B1
        block C1
            port h, v;
        end
        block C2
            port h, v;
        end
        connection [C1.v, C2.v];
    end
    block B2
        block C1
            port h, v;
        end
        block C2
            port h, v;
        end
        connection [C1.v, C2.v];
    end
    connection [B1.C1.h, B2.C1.h];
    connection [B1.C2.h, B2.C2.h];
end
```

Figure 15. A possible code for the system pictured Figure 14.

The two models are strictly equivalent.

S2ML provides actually powerful mechanisms to describe paths. These mechanisms are extensively used for cloning (see Chapter 3), aggregation (see Chapter 4) and polymorphism (see Chapter 5). First, it is possible to refer to objects from anywhere to anywhere in the block hierarchy by means of relative path and the “owner” mechanism. Second, it is possible to refer to objects by means of absolute paths by means of the “main” mechanism.

The keyword “owner” is used in paths to refer to the parent block of the current block. For instance, within the block “A.B2.C2”, it is possible to refer to the port “v” of block “A.B2.C1” as “owner.C1.v”. Similarly, it is possible to refer to port “h” of block “A.B1.C2” as “owner.owner.B1.C2.h”.
It would therefore be possible, although for sure awkward, to declare all connections from “B2.C2” block. For instance, the connections declared in block “A.B2.C2” could be as sketched Figure 16.

```
block A
     …
     block B2
          block C1
               port h, v;
               connection [owner.owner.B1.C1.h, h];
          end
          block C2
               port h, v;
               connection [owner.C1.v, v];
               connection [owner.owner.B1.C2.h, h];
          end
     end
end
```

Figure 16. Relative paths using the “owner” construct.

It is also possible to refer to objects by means of absolute paths. Absolute paths starts with the keyword “main” which designates the outmost block, i.e. the model itself (whatever the name of this block). It would be thus possible, although also quite awkward, to declare all connections from “C1” blocks. For instance, the connections declared in block “A.B1.C1” could be as sketched Figure 17.

```
block A
     …
     block B2
          block C1
               port h, v;
               connection [main.B1.C1.h, h];
          end
          block C2
               port h, v;
               connection [main.B2.C1.v, v];
               connection [main.B1.C2.h, h];
          end
     end
end
```

Figure 17. Absolute paths using the “main” construct.

3 Prototypes and Classes

This chapter introduces the notions of prototype, class, instance of class and inheritance.

3.1 Classes and Instances of Classes

3.1.1 Reusing modeling components via classes

S2ML blocks are prototype, in the sense of object and prototype oriented programming languages. A block has normally a unique occurrence. There are cases however where the system under study embeds several identical components. Consider for instance the system pictured Figure 18. This system contains two identical pumps “pump1” and “pump2”. It would be of course possible to duplicate the code of the two pumps, but this would be both error prone and confusing.

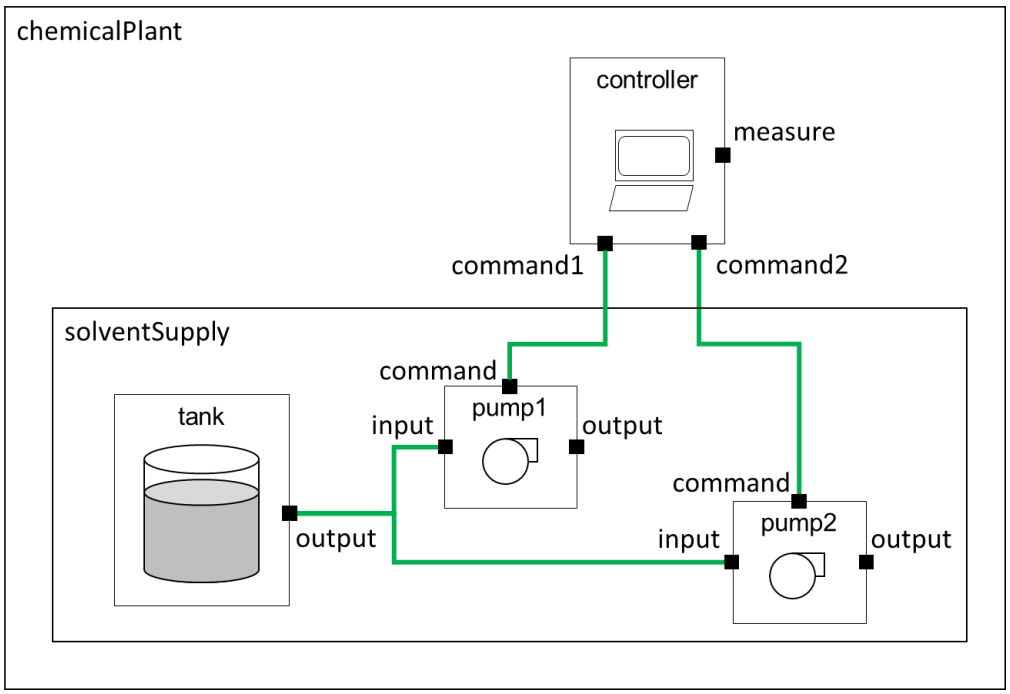

Figure 18. A system with identical components.

A first solution to address this problem is to create a class for pumps and then to instantiate this class twice in the model. A class is a separate on-the-shelf block (with possibly a full hierarchy of blocks underneath) that can be reused in models via the instantiation mechanism. Instantiating a class “C” with name “c” in a block “B” means creating a sub-block “c” of “B” whose content is the one of class “C”. The instantiation process is discussed in details Chapter 6. The code presented Figure 19 illustrates this process. First, the class “Pump” is declared. In this case, the class contains no sub-block, but only four ports and two connections. Then, the class is instantiated twice in the block “solventSupply”, respectively with names “pump1” and “pump2”.

```
class Pump
    port state, input, output, command;
    connection [state, command];
    connection [input, output, state];
end

block chemicalPlant
    …
    block solventSupply
        …
        Pump pump1, pump2;
        connection [tank.output, pump1.input];
        connection [tank.output, pump2.input];
        …
    end
    …
end
```

Figure 19. Class and instance of class.

After instantiation, the code of Figure 19 becomes as sketched Figure 20.

```
block chemicalPlant
    …
    block solventSupply
        …
        block pump1
            port state, input, output, command;
            connection [state, command];
            connection [input, output, state];
        end
        block pump2
            port state, input, output, command;
            connection [state, command];
            connection [input, output, state];
        end
        connection [tank.output, pump1.input];
        connection [tank.output, pump2.input];
        …
    end
    …
end
```

Figure 20. The code of Figure 19, once instantiated.

Classes are thus a very powerful mechanism to reuse modeling components. It is of course possible to instantiate a class into another class. It is however not possible to instantiate a class within itself, nor to have circular definitions, i.e. to instantiate the class “C1” into the class “C2”, the class “C2” into the class “C3”,… the class “Ck-1” into the class “Ck” and finally the class “Ck” into the class “C1”.

3.1.2 Graphical representation

The instantiation mechanism is hard to represent on a Process & Instrumentation Diagram like drawing (such as the one of Figure 18). It would just blur the message. UML and SysML provide

however a way to represent the instantiation like with structure breakdown type of diagram. The instantiation relation is represented by a dashed link terminated on class side with a triangle, as illustrated Figure 21. Introducing this kind of link may however make the diagrams rather complex (or necessitate representing instantiation relations separately, which leads quickly to diagram proliferation). Therefore, the best solution is probably the one suggested on Figure 21: write explicitly that the box represents a class instance and tells which class is instantiated (with drawing any link).

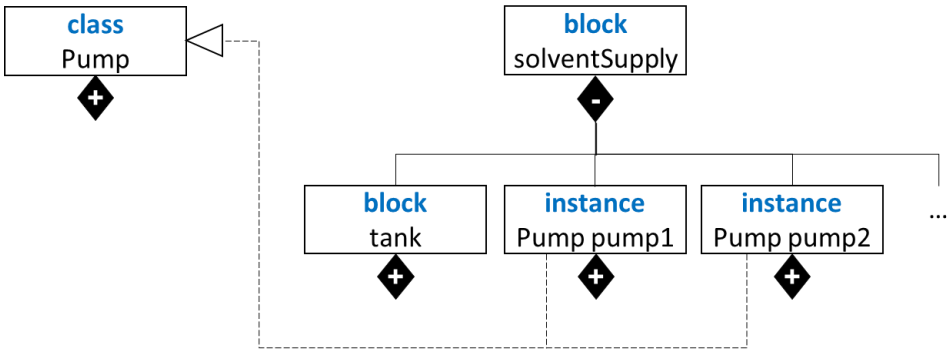

Figure 21. Representation of the class instantiation.

3.1.3 Instantiation with modification of attributes

It is possible to set up the value of attributes at instantiation time. For instance, we may want to specify that both pumps are initially open by setting their attribute “state” to “off” and to describe their maximum capacity by setting the attribute “maxFlw” of their input port to “0.1m3/s”. The corresponding code is as follows.

```
…
Pump pump1, pump2 (state="on", input.maxFlw="0.1m3s-1");
…
```

The attribute setting applies to both “pump1” and “pump2”. If the attribute settings of the two pumps were different, they should be declared separately.
If the attribute “status” is already set in the declaration of the class “Pump”, the above instantiation modifies its value (i.e. set it to “on”).
It is also possible to set attributes of connections, under the condition that they can be referred to, i.e. they are named.
Chapter 5 presents much more powerful S2ML constructs to modify the instance of a class (and more generally any modeling element).

3.2 Inheritance

3.2.1 Description

As other object-oriented languages, S2ML implements inheritance. Inheritance is a “is-a” type of relation. In modeling languages, it is very close to composition, which is a “is-part-of” type of relation. Inheritance is like composition, but without prefixing the names of elements of the inherited class.
In S2ML, inheritance is implemented by means of the “extends” clause.
As an illustration, consider the two models of pump we considered so far and that are reproduced Figure 22, for the sake of the convenience.

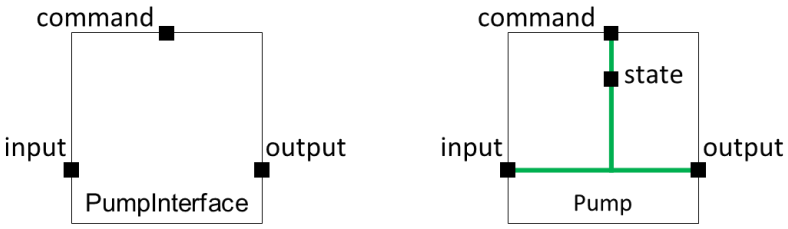

Figure 22. A pump and its interface.

The block (or class) “PumpInterface” specifies just the interface of pumps, while the block (or class) “Pump” proposes a particular implementation of pumps, with an additional port (“state”) and two connections. The code for these two classes is recalled Figure 23.
Now it is clear that the class “Pump” is a special case of the class “PumpInterface”. Therefore, it is possible to make the former derives from the latter, as illustrated Figure 24. This code is equivalent to the one of Figure 23, i.e. all ports, connections and possibly sub-blocks of the inherited class are copied into the inheriting class. In our case, only the three ports “input”, “output” and “command” are copied.

```
class PumpInterface
    port input, output, command;
end

class Pump
    port state, input, output, command;
    connection [state, command];
    connection [input, output, state];
end
```

Figure 23. The code for the class “PumpInterface” and “Pump”.

In S2ML, a block can also inherit a class, using exactly the same directive. This mechanism makes it possible to specialize a class for a particular context, without having to create and to maintain a dedicated class.

```
class PumpInterface
    port input, output, command;
end
```

```
class Pump
     extends PumpInterface;
     port state;
     connection [state, command];
     connection [input, output, state];
end
```

Figure 24. The code for the class "PumpInterface" and "Pump" using inheritance.

As for instantiation, inheritance mechanism can set the value of some attributes. For instance, we may set the attribute "maxFlw" of the port "input" to "0.2m3s-1" as follows.

```
class Pump
     extends PumpInterface (input.maxFlw="0.2m3s-1");
     …
end
```

3.2.2 Graphical representation

The UML/SysML representation for inheritance is a plain link terminated with an empty triangle on the inherited class side, as illustrated Figure 25.

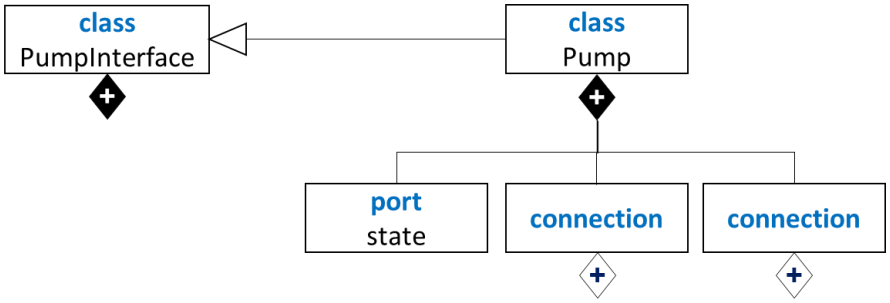

Figure 25. Graphical representation of inheritance.

In the case of a block inheriting from a class, the same graphical representation can be used. In the case where drawing transversal links overloads the graphics, it is probably possible to annotate the rectangle representing the block.

3.2.3 Multiple inheritance

Multiple inheritance is allowed in S2ML, although not recommended. A classical problem is with multiple inheritance is the conflicts raised by common ancestors. Assume for instance a situation as pictured Figure 26. Both classes "C2" and "C3" inherit from a class "C1". The class "C4" inherits both from classes "C2" and "C3".

The first issue is to avoid duplicating in class "C4" an object, say a port "p", declared in class "C1" (since "p" comes from two sources). The situation gets even trickier if both class "C2" and "C3" declare an object with the same name, here a sub-block "B" with possibly some common as well as different components. Similarly, there may be some conflicting attribute settings. Assume for

instance that the attribute “a” of the port “p” is set to “1” in class “C1”, to “2” in class “C2” and to “3” in class “C3”. What is the value of “a” in class “C4”?

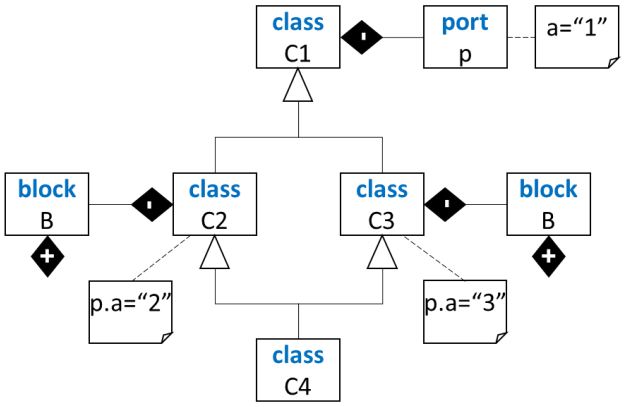

Figure 26. Problems raised by multiple inheritance.

The S2ML code corresponding to Figure 26 is given Figure 27.

```
class C1
    port p (a="1");
end

class C2
    extends C1(p.a="2");
    block B
        port q, r (b="2");
    end
end

class C3
    extends C1(p.a="3");
    block B
        port q, s (c="3");
    end
end

class C4
    extends C2;
    extends C3;
end
```

Figure 27. Code showing conflicts raised by multiple inheritance.

S2ML solves partially the above conflicts by considering that a model is an ordered sequence of declarations (see Chapter 5).

3.3 Cloning

3.3.1 Description

Consider again the system pictured Figure 18. It may be the case that creating a class for pumps is not worth (for instance because this type of pumps are specific to that system) or that pumps aggregate (see Chapter 4) some component of the system (for instance the power supply) so it is impossible to declare them in a separate model. In such situation, a possible solution (to avoid duplicating the code) consists in declaring the block representing one of the two pumps, then into cloning this block.
In S2ML, cloning is implemented by means of the “clones-as” directive.
This construct is illustrated by the (partial) code given Figure 28. A block “pump1” is declared, then a block “pump2” that clones the block “pump1”.

```
block chemicalPlant
    …
    block solventSupply
        …
        block pump1
            port input, output, command;
            …
        end
        clones pump1 as pump2;
        connection [tank.output, pump1.input];
        connection [tank.output, pump2.input];
        …
    end
    …
end
```

Figure 28. Partial code for the system pictured Figure 18.

3.3.2 Handling references to outside entities

Block cloning is thus similar to class instantiation. It is more powerful however, because the cloned block may refer to external entities, while a class cannot refer anything to outside itself.
Consider for instance, the (partial) model described Figure 29. The block “A.B.D” declares a connection that refers to its own port “s”, to the port “q” of the block “A.B.C” and finally to the port “p” of the main block (“A”). The block “D” is cloned into a block “F” within the block “A.E”. The connection is thus cloned. In the cloned connection, the reference to port “s” of block “E” causes no problem for it is purely internal. The reference “owner.owner.p” to port “p” of block “A” is already more problematic. To work correctly, it requires that the cloning block “A.E.F” stands at the same declaration depth as the cloned block “A.B.D”. The reference “owner.C.q” is even more problematic for it requires that a block “C” itself declaring a port “q” is declared under the parent of “F”.
This example illustrates both the power, but also the potential complexity of cloning.

```
block A
    port p;
    block B
        block C
            port q, r;
        end
        block D
            port s;
            connection [s, owner.C.q, owner.owner.p];
        end
    end
    block E
        block C
            port q, t;
        end
        clones main.B.D as F;
    end
end
```

Figure 29. Cloning and external references.

3.3.3 Graphical representation

Because there is no difference in essence between block cloning and class instantiation, they should be graphically represented in the same way, i.e. with a dotted line terminated on the side of the cloned block by an empty triangle, as illustrated Figure 21 for classes.

3.4 Packages

3.4.1 Description

Classes are independent modeling entities. By default, all classes stand thus at the same hierarchical level, the top level. S2ML provides a mean to organize classes via the notion of package. Packages are containers for classes. A package may contain classes and other containers so to give a hierarchical (tree-like) organization. Classes inside packages (at any nested level) are simply referenced to using the dot notation. This principle is illustrated on the code given Figure 30.

```
package P
    package Q
        class C
            …
        end
    end
end
block B
    P.Q.C I;
end
```

Figure 30. S2ML code for packages.

The block “B” declares an instance “I” of the class “P.Q.C”.
Packages are on-the-shelf modeling entities. They cannot contain blocks (classes declared inside packages can of course declare internal blocks).
Two different packages can declare two different classes with the same name. In the example of Figure 30, a package “R” could declare a class “C”.
The “owner” and “main” constructs are not allowed for packages: to refer to a class, only simple paths, starting from the root package are allowed.

3.4.2 Graphical representation

Although probably not very useful in the S2ML context, it is possible to represent graphically packages, using the same graphical constructs as in UML, as illustrated Figure 31.

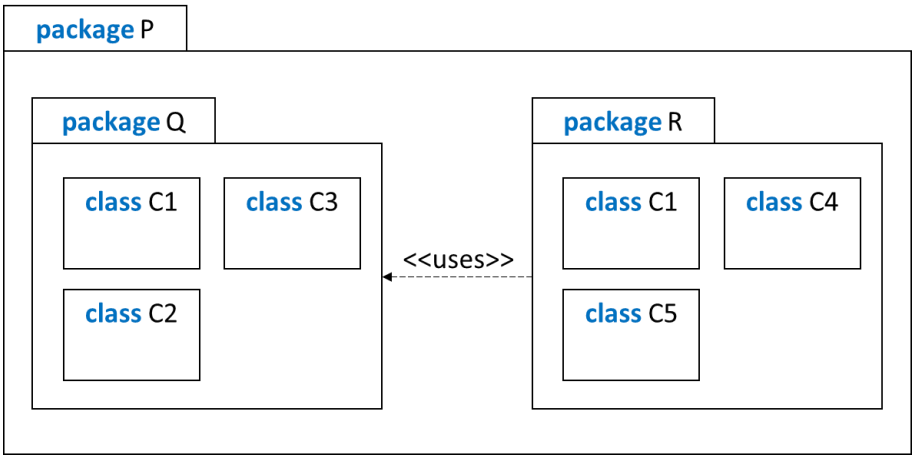

Figure 31. Package diagram.

4 Aggregation

This chapter presents S2ML construct to implement aggregation.

4.1 Rational

In modeling languages, composition describes a “is-part-of” relation. This relation assumes that a component cannot belong to two different super-components. There are cases however where the same component is used at several places, or more exactly to serve, different functions of the system. This kind of “uses” relations can be described by means of aggregation.
As an illustration, consider the chemical plant described Figure 32. The figure shows something like a physical breakdown structure of the system: it is made of three sub-systems: the solvent supply, the controller and the vessel. The solvent supply and the vessel are further decomposed: the solvent supply system is made of a tank and two pumps; the vessel is made of a reactor and two sensors.

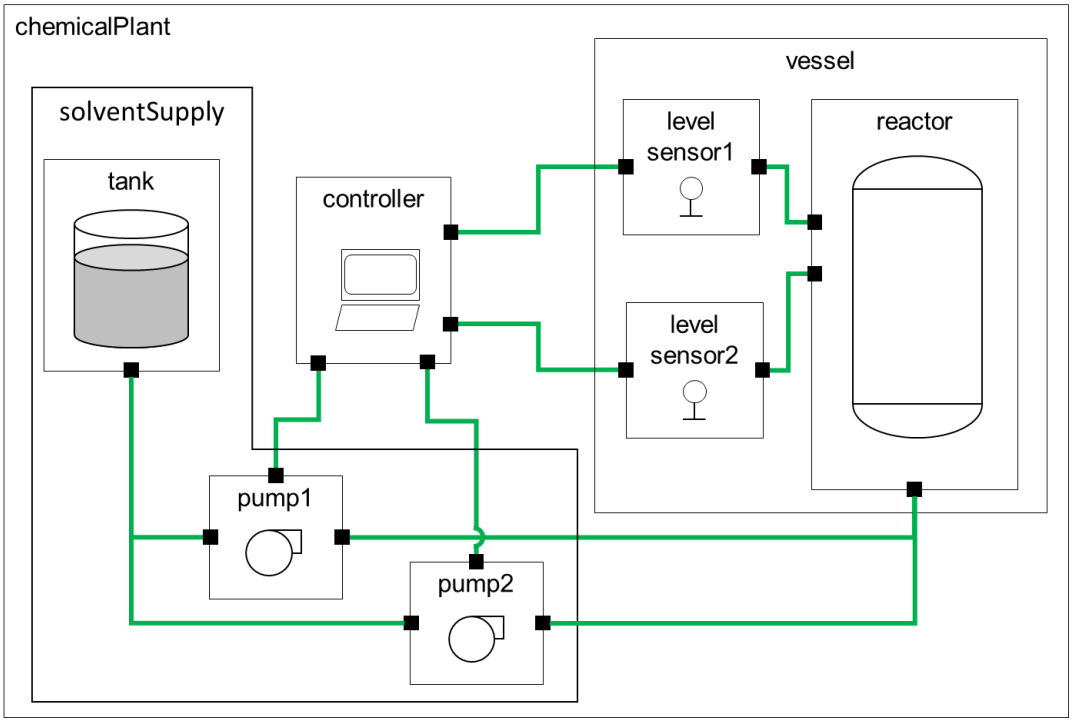

Figure 32. A chemical plant.

The above structural breakdown is fine, although it mixes physical description and ones that are more functional. The solvent supply system for instance is more a functional grouping than a physical grouping of components. However, this structural breakdown is probably not the only decomposition of the system that would be used. For instance, engineers in charge of the control of the chemical reaction would probably look at the system via functional chains sensor-calculator-actuator. In the above example, these functional chains could look like those of Figure 33.

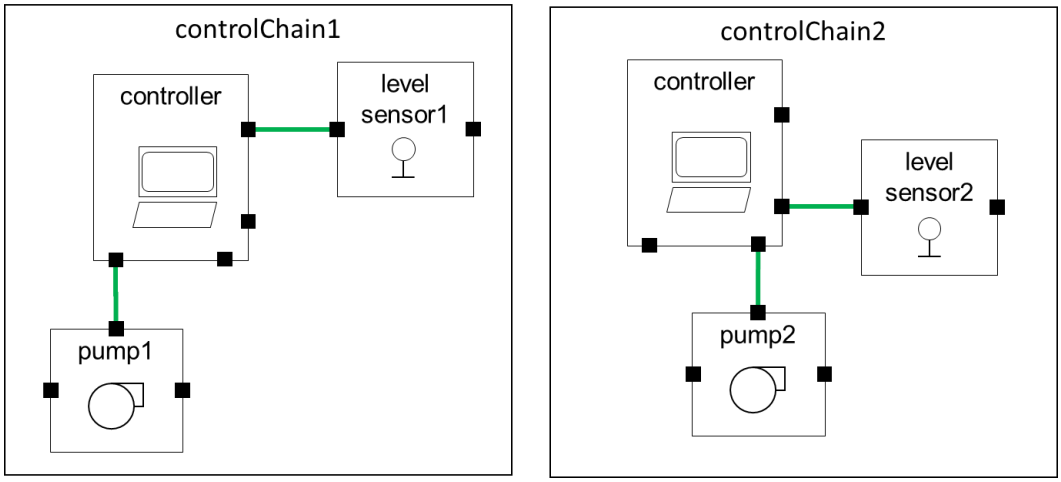

Figure 33. Functional chains for the control of the reaction.

These functional chains are clearly made of components that are described in the structural breakdown. In some sense, they only "use" these components. Moreover, the two chains share the calculator. It is probably the case also that the connections sensor-calculator and calculator-pump are better described (edited) via the functional chains than in the structural breakdown. Some even argue that, when the system is complex enough, it is never edited as a whole, but only through functional chains.

Therefore, we need to be able to describe hierarchical breakdown with branches sharing components. This is the very purpose of the notion of aggregation. In modeling languages, aggregation can be seen as a "uses" relation. The functional chain "controlChain1" uses the component "controller", which is declared elsewhere.

4.2 S2ML encoding

S2ML provides the directive "embeds-as" to implement aggregation. The use of this directive is illustrated by the code of Figure 34. This code declares first two classes, one for the pumps, and one for the sensors. Then comes the core of the model, i.e. the block "chemicalPlant". Note that the classes "Pump" and "Sensor" could have been declared after the block "chemicalPlant" as well. The order of declarations for top level elements does not matter.

The block "chemicalPlant" declares five sub-blocks "vessel", "solventSupply", "controller", "controlChain1" and "controlChain2". Here the order of declarations does matter. It is not possible to refer to an element that has not been yet declared. The declarations of blocks "vessel" and "controller" do not deserve many comments.

The declaration of the block "solventSupply" shows a first use of aggregation. The block aggregates the port "input" from the block "chemicalPlant.vessel.reactor". This is only possible because this port has been declared before being referred to by the "embeds-as" directive. The port is embedded as "output", which means that its local name is output. It is then possible to use this local name to declare connections. It would have been of course possible to declare a port "output" in the block "solventSupply" and to connect it subsequently (either at "solventSupply" level or at "chemicalPlant" level with the output of the pumps on the one hand and with the input of the reactor on the other

hand. However, such port and connections would not correspond to any physical entity. It would thus overload the model with irrelevant constructs.

```
class Pump
    port input, output, command;
end

class Sensor
    port input, output;
end

block chemicalPlant
    block vessel
        Sensor sensor1, sensor2;
        block reactor
            port input, level1, level2;
        end
        connection [reactor.level1, sensor1.input];
        connection [reactor.level2, sensor2.input];
    end
    block solventSupply
        block tank
            port output;
        end
        Pump pump1, pump2;
        embeds main.vessel.reactor.input as output;
        connection [tank.output, pump1.input];
        connection [tank.output, pump2.input];
        connection [pump1.output, output];
        connection [pump2.output, output];
    end
    block controller
        port measure1, measure2, command1, command2;
    end
    block controlChain1
        embeds main.vessel.sensor1 as sensor;
        embeds main.controller as controller;
        embeds main.solventSupply.pump1 as actuator;
        connection [sensor.output, controller.measure1];
        connection [controller.commmand1, actuator.command];
    end
    block controlChain2
        embeds main.vessel.sensor2 as sensor;
        embeds main.controller as controller;
        embeds main.solventSupply.pump2 as actuator;
        connection [sensor.output, controller.measure2];
        connection [controller.commmand2, actuator.command];
    end
end
```

Figure 34. The S2ML code for the chemical plant of Figure 32 and Figure 33.

Blocks “controlChain1” and “controlChain2” aggregate previously declared blocks. For instance, the block “controlChain1” embeds the block “chemicalPlant.vessel.sensor1” and names it “sensor”. Note that embedded blocks are referred to by means of absolute paths. They could have been referred to by means of relative paths as well.

The above example makes clear that in modeling languages, aggregation is a “uses” relation. In the S2ML context, it can be seen just as local renaming of components defined elsewhere.

4.3 Graphical representation

Representing graphically aggregation while keeping a certain completeness is a challenge. Figure 9 showed a graphical representation of aggregated ports (referred to by a connection). Figure 35 extends this representation to blocks. As in UML/SysML aggregation is represented by a plain link ended with an empty diamond on the aggregating block side.

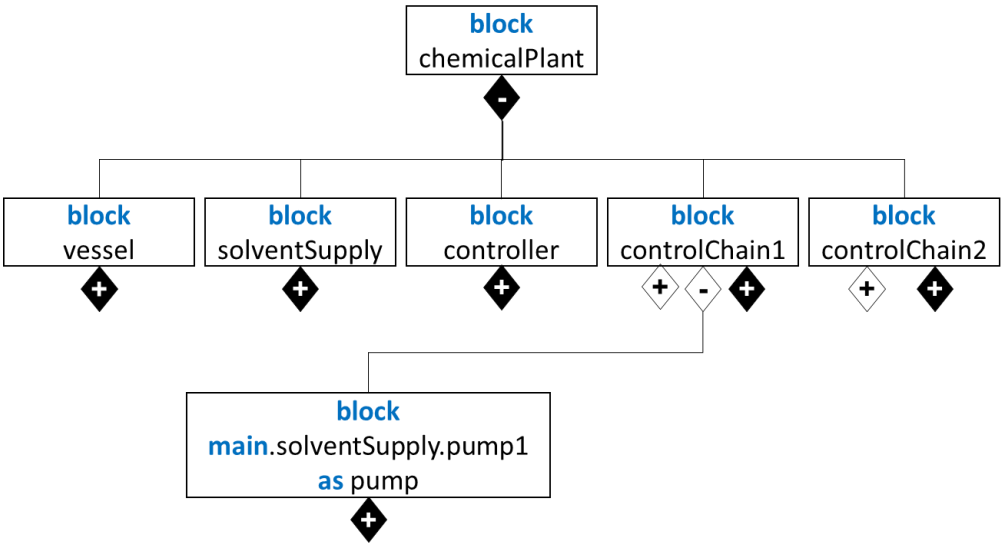

Figure 35. Graphical representation of aggregation.

The difficulty stands indeed in the relationship between the path given in the aggregated block and the block this path refers to. It is probably possible to use highlighting or some other graphical gadget to show this relationship.

5 Polymorphism

In programming languages, polymorphism denotes the capacity of a single interface to host entities of different types. S2ML implements a form of polymorphism, via re-declarations and the “deletes” directive that makes it possible to forget declarations. This chapter presents these constructs.

5.1 Handling re-declarations

S2ML considers a model as an ordered sequence of declarations. The declarations are considered in the order they are given in the text (here graphical representation is of no help).
If a named element is re-declared, then the new declaration is merged with the older one. To illustrate rules to handle re-declarations, we shall consider again (a part of) the chemical plant Figure 36.

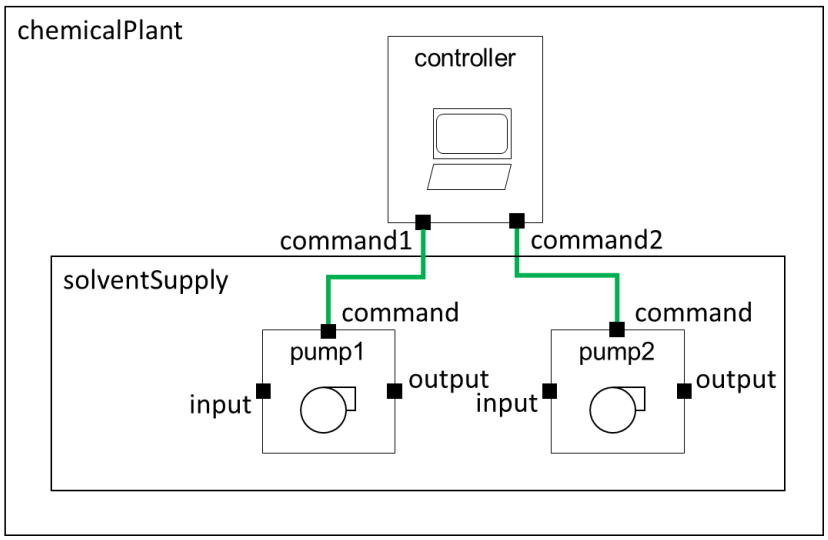

Figure 36. Part of the chemical plant.

5.1.1 Type mismatch

If a named object is re-declared with a different type, then an error is raised. E.g. in the code of Figure 37, the name “command” is first associated with a port, then a connection. A typing error is thus raised.

5.1.2 Attribute lists

If a named object is re-declared, then the attribute list of the new declaration is merged with the old one:

- Attributes that are in the old list but not in the new one are kept unchanged.
- Attributes that are in the new list are kept unchanged.

E.g. the codes of (a) and (b) of Figure 38 are equivalent.

```
block chemicalPlant
     block controller
          port command1, command2;
     end
     block solventSupply
          block pump1
               port input, output, command;
               connection command[command, main.controller.command];
                          // type mismatch the identifier command
                          // is already used for a port
          end
     end
end
```

Figure 37. Example of type mismatch in re-declaration.

```
block pump2
     port input, output ;
     port command(type="signal", channel="1");
     port command(location="right", channel ="2");
          // re-declaration of the port command
end
```

(a) with redeclaration

```
block pump2
     port input, output ;
     port command(type="signal", channel="2", location="right");
end
```

(b) without redeclaration

Figure 38. Re-declarations of attributes.

5.1.3 Ports

If a port is re-declared, then the only effect is to merge the old and new attribute lists.
Note that it is possible to (re-)declare a port nested in a block, as illustrated Figure 39. On this figure, the two codes are equivalent.

There is a restriction to the above rule however: it is not possible to re-declare a composed port as aggregated. Such a re-declaration would delete actually the reference to the composed port. The code of Figure 40 for instance would raise an error.

It is possible to re-declare an aggregated port as another port. As an illustration, consider the code of Figure 41. There are three re-declaration in this code.
There is a first re-declaration, just after the “embeds-as” directive in block “pump1”. This re-direction adds the attribute “channel”, with the value “1”, to the port “source” which is actually the port “chemicalPlant.controller.command1”.

```
block solventSupply
     block pump2
          port command(type="signal", channel="1");
     end
     port pump2.command(channel="2", location="right");
```

```
end
```

(a) with re-declaration

```
block solventSupply
    block pump2
        port command(type="signal", channel="2", location="right");
    end
end
```

(b) without re-declaration

Figure 39. Re-declaration of ports.

```
block chemicalPlant
    block controller
        port command1, command2;
    end
    block solventSupply
        block pump1
            port input, output, command;
        end
    end
    embeds controller.command1 as solventSupply.pump1.command;
          // Error: this would delete the access to composed
          // port solventSupply.pump1.command
end
```

Figure 40. Error: re-declaration of a composed port into an aggregated one.

```
block chemicalPlant
    block controller
        port command1, command2;
    end
    block solventSupply
        block pump1
            port input, output, command;
            embeds main.controller.command1 as source;
            port source(channel="1");
        end
        clones pump1 as pump2;
        embeds main.controller.command2 as pump2.source;
        port pump2.source(channel="2");
        connection [pump1.command, pump1.source];
        connection [pump2.command, pump2.source];
    end
end
```

Figure 41. Re-declaration of an aggregated port.

There is a second re-declaration just after the directive that clones the block “pump1” into “pump2”. This re-declaration changes, in the block “pump2” the reference “source” to “chemicalPlant.controller.command2”.

Finally, there is a third re-direction right after, which sets the attribute “channel” of the port “pump2.source”, i.e. the port “chemicalPlant.controller.command2” to “2”.
The code of Figure 41 is thus equivalent to the one of Figure 42.

```
block chemicalPlant
    block controller
        port command1(channel="1"), command2(channel="2");
    end
    block solventSupply
        block pump1
            port input, output, command;
            embeds main.controller.command1 as source;
        end
        block pump2
            port input, output, command;
            embeds main.controller.command2 as source;
        end
        connection [pump1.command, pump1.source];
        connection [pump2.command, pump2.source];
    end
end
```

Figure 42. A code equivalent to the one of Figure 41, without re-declaration.

5.1.4 Anonymous connections

Anonymous connections cannot be re-declared. Therefore, if two identical anonymous connections are declared one after the other, they are both kept. E.g.

```
block B
    port p, q;
    connection [p, q];
    connection [p, q]; // no merging: this is another connection
end
```

This rule explains why multiple inheritance should be handled with much care. Assume for instance, that a class “D” inherits two classes “B” and “C” in order. First, the code of “B” is copied in the class “D”. Then, the one of “C”. Assume moreover that these two classes inherit themselves from a common class “A”. Assume finally that this class “A” declares an anonymous connection. Then this connection will be duplicated in class “D” because it is declared first in “B”, then in “C”.

Anonymous connections can cause even more subtle problems, due to side effects. Consider the code of Figure 43, which is apparently very similar to the one of Figure 41. The only difference stands in the declaration of the anonymous connection, which is now internalized in the block describing the pump.
Although natural, this declaration is the cause of a hidden problem.

```
block chemicalPlant
    block controller
        port command1, command2;
```

```
    end
    block solventSupply
        block pump1
            port input, output, command;
            embeds main.controller.command1 as source;
            port source(channel="1");
            connection [command, source];
        end
        clones pump1 as pump2;
        embeds main.controller.command2 as pump2.source;
        port pump2.source(channel="2");
            // problem: the connection is still referring to
            // chemicalPlant.controller.command1
    end
end
```

Figure 43. An incorrect model of the chemical plant.

The block “pump1” is cloned into the block “pump2”. The block “pump2” thus aggregates a port named “source” and referring to the port “chemicalPlant.controller.command1”, as well as an anonymous connection between the internal port “command” and the aggregated port “source”.
The re-declaration of the port “source” changes the port it refers to, which is now “chemicalPlant.controller.command2”. However, the anonymous connection is left unchanged, therefore still referring to “chemicalPlant.controller.command1”. Worse, since it is anonymous, there is now way to refer to that connection, and therefore no way to fix the problem.
It is tempting to change S2ML semantics to avoid this problem. However, such a change would probably raise more problems than it would solve.

5.1.5 Named connections

If a named connection is re-declared, then:

- Attribute lists of the old and the new declarations are merged.
- Lists of ports of the connection become the one of the new declaration.

As an illustration, we can correct the code of Figure 43 as proposed Figure 44. Now the connection is named (“linkToController”). After the block “pump1” is cloned as “pump2”, the port “source” is re-declared and its attribute “channel” changed to “2”. Eventually, the connection “linkToController” is re-declared so to ensure that the second pump is connected to the second command port of the controller. The code of Figure 44 is thus equivalent to the one of Figure 42.

5.1.6 Blocks

If a block is re-declared, then:

- Attribute lists of the old and the new declarations are merged.
- Declarations of the new block are treated in sequence. If they declare new objects, these objects are just added to the block. Otherwise, rules for re-declarations are applied.

It is not possible to re-declare a composed block as aggregated for this would lose the reference to the old block.

```
block chemicalPlant
    block controller
        port command1, command2;
```

```
    end
    block solventSupply
        block pump1
            port input, output, command;
            embeds main.controller.command1 as source;
            port source(channel="1");
            connection linkToController[command, source];
        end
        clones pump1 as pump2;
        embeds main.controller.command2 as pump2.source;
        port pump2.source(channel="2");
        connection pump2.linkToController
                    [pump2.command, pump2.source];
    end
end
```

Figure 44. Corrected code for the chemical plant.

The code of Figure 44 could thus be rewritten as illustrated Figure 45.

```
block chemicalPlant
    block controller
        port command1, command2;
    end
    block solventSupply
        block pump1
            port input, output, command;
            embeds main.controller.command1 as source;
            port source(channel="1");
            connection linkToController[command, source];
        end
        clones pump1 as pump2;
        block pump2
            embeds main.controller.command2 as source;
            port source(channel="2");
            connection linkToController[command, source];
        end
    end
end
```

Figure 45. Re-declaration of a composed block.

Re-declaring an aggregated block produces a side effect since this block is modified through its reference. This construction is illustrated Figure 46. The block “pump1” aggregates the block “chemicalPlant.controller” with a local name “controller”. Then the aggregated block “controller” is re-declared first to set the value of the attribute “channel” of its port “command1”, second to create a link to the port “command” of the block “pump1”.

```
block chemicalPlant
    block controller
        port command1, command2;
```

```
    end
    block solventSupply
        block pump1
            port input, output, command;
            embeds main.controller as controller;
            block controller
                port command1(channel="1");
                connection linkToPump[owner.command, command1];
            end
        end
    end
end
```

Figure 46. Re-declaration of an aggregated port.

The main interest of block re-declaration stands in sub-typing instances of classes. Consider for instance a very generic model of pumping system with two identical pumps working in parallel. We can define a class for generic pumps and a class for generic pumping systems. Then, we can define a particular pumping system, derived from the generic one, by specializing pumps. This process is illustrated Figure 47.

```
class GenericPump
    port input, output;
end

class GenericPumpingSystem
    port input, output;
    GenericPump pump1, pump2;
    connection c1[input, pump1.input];
    connection c2[input, pump2.input];
    connection c3[pump1.output, output];
    connection c4[pump1.output, output];
end

class MyPump
    extends GenericPump;
    port state, command;
    connection flowDescription[state, input, output];
    connection commandAction[command, state];
end

block MyPumpingSystem
    extends GenericPumpingSystem;
    MyPump pump1, pump2;
end
```

Figure 47. Sub-typing as re-declaration.

5.2 Deleting declarations

Re-declarations as presented in the previous section mainly add new elements to old ones. There are two exceptions however: attributes may change of value and connections may change of lists of ports. The “deletes” clause makes it possible to delete declarations, pure and simple. This clause is given a path p as an argument. All elements that can be referred to with a path in the form p.q are deleted (assuming q is a, possibly empty, path that does not contain any directives “main” and “owner”). This deletion may have a snowball effect: all references (including via aggregation) to the deleted elements are themselves deleted. If a deleted element was referred to in a connection, this connection is deleted too. In a word, the deleted elements cease to exist in the model. It is then possible to replace them by fresh ones.
As an illustration, consider the code Figure 48.

```
class Pump
    port input, output;
    port command(type="manual");
    connection [input, output, command];
end

block chemicalPlant
    block solventSupply
        block tank
            port output;
        end
        Pump pump;
        deletes pump.command.type;
        connection flowEquation[tank.output, pump.input];
    end
    block controller
        port source;
        embeds main.solventSupply.pump.command as target;
        connection action[source, target];
    end
    deletes solventSupply.pump;
end
```

Figure 48. Illustration of the “deletes” directive.

The first use of the “deletes” clause, after the declaration of the pump, just deletes the attribute “type” of the port “command” of the pump. This port has therefore no more attribute in the instance. The class is left unchanged, i.e. a subsequent instance of this class would have the attribute “type” of the port “command” set to “manual”.
The second use of the clause has a much more dramatic effect. It deletes the instance “pump” of “Pump” in the block “chemicalPlant.solventSupply” and all elements of this instance, i.e. the three ports “input”, “output” and “command” and the connection. All references to these elements are also deleted: the aggregated port “target” of the block “chemicalPlant.controller” and the references to this port. This in turn deletes the connection “action” of the block. The resulting code is pictured Figure 49.

```
class Pump
    port input, output;
    port command(type="manual");
    connection [input, output, command];
end

block chemicalPlant
    block controller
        port command;
    end
    block solventSupply
        block tank
            port output;
        end
    end
end
```

Figure 49. Code resulting of the application of the "deletes" clause in code of Figure 48.

Needless to say that the “deletes” clause is to handle with much care.

6 Semantics

6.1 Introduction

The semantics of a S2ML model is a flat list of declarations of blocks, ports and connections together with their attributes. This list may contain also declarations of aliases resulting of aggregations. As an illustration, consider the code Figure 50.

```
class Pump
    port input, output, command(type="manual");
    connection [input, output, command];
end

block chemicalPlant
    block solventSupply
        block tank
            port output;
        end
        Pump pump;
        connection flowEquation[tank.output, pump.input];
    end
    block controller
        port source;
        embeds main.solventSupply.pump.command as target;
        connection action[source, target];
    end
end
```

Figure 50. A sample for the code.

The semantics of this model (i.e. of the block “chemicalPlant”) is the list of declarations given Figure 51. In these declarations, all names have been resolved: there is no more local names or names involving directives “owner” or “main”, references to aggregate elements are replaced by absolute paths. Nevertheless, aliases are kept because they can be used later on.
The S2ML semantics sees a model as a sequence of instructions. At any step of the “execution” of the model, the flat list contains a number of declarations. Instructions of the model add, modify or delete declarations in this flat list. The best way to define formally the semantics of a S2ML model is thus probably the Plotkin’s Structural Operational Semantics (SOS) style.

6.2 Path resolution

The first step in design (SOS) rules for S2ML is to define how paths, possibly containing “owner” and “main” directives are resolved.

Figure 52 gives the rules for name resolution. Above the bar stands the path to resolve in a given context, i.e. for a given prefix path. The context stands before the semi-colon, the path to resolve after the semi-colon.

```
block chemicalPlant
block chemicalPlant.solventSupply
block chemicalPlant.solventSupply.tank
port chemicalPlant.solventSupply.output
block chemicalPlant.solventSupply.pump
port chemicalPlant.solventSupply.pump.input
port chemicalPlant.solventSupply.pump.output
port chemicalPlant.solventSupply.pump.command(type="manual")
connection [
             chemicalPlant.solventSupply.pump.input,
             chemicalPlant.solventSupply.pump.output,
             chemicalPlant.solventSupply.pump.command]
connection chemicalPlant.solventSupply.flowEquation[
             chemicalPlant.solventSupply.tank.output,
             chemicalPlant.solventSupply.pump.input]
block chemicalPlant.controller
port chemicalPlant.controller.source
embeds chemicalPlant.solventSupply.pump.command as
      chemicalPlant.controller.target
connection chemicalPlant.controller.action[
             chemicalPlant.controller.source,
             chemicalPlant.solventSupply.pump.command]
```

Figure 51. The semantics of the block "chemicalPlant" of Figure 50.

Below the bar stands the context and the path obtained after application of the rule. Identifiers are denoted by "Id". Paths are denoted by lower case Greek letters.

$$\text{P1: } \frac{\sigma\ ;}{\sigma\ ;\ \sigma} \qquad \text{P2: } \frac{\sigma\ ;\ Id.\pi}{\sigma.Id\ ;\ \pi}$$

$$\text{P3: } \frac{\sigma.Id\ ;\ \mathbf{owner}.\pi}{\sigma\ ;\ \pi} \qquad \text{P3': } \frac{;\ \mathbf{owner}.\pi}{ERROR}$$

$$\text{P4: } \frac{Id.\sigma\ ;\ \mathbf{main}.\pi}{Id\ ;\ \pi} \qquad \text{P4': } \frac{;\ \mathbf{main}.\pi}{ERROR}$$

Figure 52. Rules for name resolution.

We denote $[\![\sigma,\pi]\!]$ the path obtained by resolving the path π in the context σ applying rules P1-P4'. Once resolved, a name may be changed for another one if it is the local name of an aggregated element. In the sequel, we shall denote flat list with upper case Greek letters. Since the order of declarations within the flat list does not really matter, we shall write for instance: "{block a.b} $\cup\ \Gamma$" for a list declaring the block "a.b" and a set Γ of other declarations.

Figure 53 shows rules for de-aliasing. The proposition after the bar "|" is a condition for the rule to be applicable.

$$\text{E1: } \frac{\{\textbf{embeds } \rho \textbf{ as } \sigma\} \cup \Gamma \ ; \ \sigma}{\rho} \qquad \text{E2: } \frac{\Gamma \ ; \ \sigma \ | \ \neg\exists \textbf{ embeds } \rho \textbf{ as } \sigma \ \in \Gamma}{\sigma}$$

Figure 53. Rules for de-aliasing.

By extension, we denote $[\![\Gamma, \sigma, \pi]\!]$ the path obtained by resolving the path π in the context σ and the current flat list Γ by applying rules P1-P4' and then applying rules E1-E2.

6.3 Attributes

We need formal rules to describe merging of list of attributes. Let M and N be two lists (sets) of attributes, we denote by M ⊗ N the merging of M and N. This operation is not commutative since if an attribute occurs in both M and N, it takes the value set in N. As previously, we shall use a set notation for lists of attributes.
Figure 54 shows rules for merging lists of attributes.

$$\text{A0: } \frac{M \otimes \emptyset}{M}$$

$$\text{A1: } \frac{(\{a = v\} \cup M) \otimes (\{a = w\} \cup N)}{(\{a = w\} \cup M) \otimes N} \qquad \text{A2: } \frac{M \otimes (\{a = w\} \cup N) \ | \neg\exists \ a = v \in M}{(\{a = w\} \cup M) \otimes N}$$

Figure 54. Rules for merging of lists of attributes.

6.4 Declarations

The rules that define the semantics of S2ML are similar to those to resolve names and compose lists of attributes. Above the bar stand: first, the current flat list, denoted by upper Greek letters; second, the current context, denoted as lower case letters; and third the sequence of declarations to be resolved. Under the bar stand the three same items after the application of the rule.
Figure 54 shows the rules for declarations.
Rules D1 and D2 are respectively for port declarations and port re-declarations. Rule D3 is for anonymous connection declarations. Rules D4 and D5 are respectively for named connection declarations and named connection re-declarations. Rules D6 and D7 are respectively for block declarations and block re-declarations. Note that first the declarations of block are evaluated in the context of the block (i.e. ρ), and then the remaining declarations (Θ) are evaluated in the current context (i.e. σ). Rules R8 and R9 are respectively for class instantiations and block re-declarations as class instantiations. They are similar to the previous ones.

$$\text{D1:}\ \frac{\Gamma\ ;\ \sigma\ ;\ \textbf{port}\ \pi\ N,\ \Theta\ \mid [\![\Gamma,\sigma,\pi]\!] = \rho,\ \neg\exists\ \textbf{port}\ \rho\ M \in \Gamma}{(\textbf{port}\ \rho\ N) \cup \Gamma\ ;\ \sigma\ ;\ \Theta}$$

$$\text{D2:}\ \frac{(\textbf{port}\ \rho\ M) \cup \Gamma\ ;\ \sigma\ ;\ \textbf{port}\ \pi\ N,\ \Theta\ \mid [\![\Gamma,\sigma,\pi]\!] = \rho}{(\textbf{port}\ \rho\ M \otimes N) \cup \Gamma\ ;\ \sigma\ ;\ \Theta}$$

$$\text{D3:}\ \frac{\Gamma\ ;\ \sigma\ ;\ \textbf{connection}\ [\pi_1 \cdots \pi_k]\ N,\ \Theta\ \mid [\![\Gamma,\sigma,\pi_1]\!] = \rho_1 \cdots [\![\Gamma,\sigma,\pi_k]\!] = \rho_k}{(\textbf{connection}\ [\rho_1 \cdots \rho_k]\ N) \cup \Gamma\ ;\ \sigma\ ;\ \Theta}$$

$$\text{D4:}\ \frac{\begin{array}{c}\Gamma\ ;\ \sigma\ ;\textbf{connection}\ \pi\ [\pi_1 \cdots \pi_k]\ N,\ \Theta \\ \mid [\![\Gamma,\sigma,\pi]\!] = \rho, [\![\Gamma,\sigma,\pi_1]\!] = \rho_1 \cdots [\![\Gamma,\sigma,\pi_k]\!] = \rho_k, \neg\exists\ (\textbf{connection}\ \rho\ P\ M) \in \Gamma\end{array}}{(\textbf{connection}\ \rho\ [\rho_1 \cdots \rho_k]\ N) \cup \Gamma\ ;\ \sigma\ ;\ \Theta}$$

$$\text{D5:}\ \frac{\begin{array}{c}(\textbf{connection}\ \rho\ P\ M) \cup \Gamma\ ;\ \sigma\ ;\textbf{connection}\ \pi\ [\pi_1 \cdots \pi_k]\ N,\ \Theta \\ \mid [\![\Gamma,\sigma,\pi]\!] = \rho, [\![\Gamma,\sigma,\pi_1]\!] = \rho_1 \cdots [\![\Gamma,\sigma,\pi_k]\!] = \rho_k\end{array}}{(\textbf{connection}\ \rho\ [\rho_1 \cdots \rho_k]\ M \otimes N) \cup \Gamma\ ;\ \sigma\ ;\ \Theta}$$

$$\text{D6:}\ \frac{\Gamma\ ;\ \sigma\ ;\ \textbf{block}\ \pi\ N\ B,\ \Theta\ \mid [\![\Gamma,\sigma,\pi]\!] = \rho,\ \neg\exists\ \textbf{block}\ \rho\ M \in \Gamma}{\Gamma'\ ;\ \sigma\ ;\ \Theta\ \mid \langle(\textbf{block}\ \rho\ N) \cup \Gamma\ ;\ \rho\ ;\ B\ \rangle = \langle\ \Gamma'\ ;\ \gamma\ ;\rangle}$$

$$\text{D7:}\ \frac{(\textbf{block}\ \rho\ M) \cup \Gamma\ ;\ \sigma\ ;\ \textbf{block}\ \pi\ N\ B,\ \Theta\ \mid [\![\Gamma,\sigma,\pi]\!] = \rho}{\Gamma'\ ;\ \sigma\ ;\ \Theta\ \mid \langle(\textbf{block}\ \rho\ M \otimes N) \cup \Gamma\ ;\ \rho\ ;\ B\ \rangle = \langle\ \Gamma'\ ;\ \gamma\ ;\rangle}$$

$$\text{D8:}\ \frac{\Gamma\ ;\ \sigma\ ;\ C\ \pi\ N,\ \Theta\ \mid [\![\Gamma,\sigma,\pi]\!] = \rho,\ \neg\exists\ \textbf{block}\ \rho\ M \in \Gamma,\ \textbf{class}\ C\ L\ B}{\Gamma'\ ;\ \sigma\ ;\ \Theta\ \mid \langle(\textbf{block}\ \rho\ L \otimes N) \cup \Gamma\ ;\ \rho\ ;\ B\ \rangle = \langle\ \Gamma'\ ;\ \gamma\ ;\rangle}$$

$$\text{D9:}\ \frac{(\textbf{block}\ \rho\ M) \cup \Gamma\ ;\ \sigma\ ;\ C\ \pi\ N,\ \Theta\ \mid [\![\Gamma,\sigma,\pi]\!] = \rho,\ \textbf{class}\ C\ L\ B}{\Gamma'\ ;\ \sigma\ ;\ \Theta\ \mid \langle(\textbf{block}\ \rho\ M \otimes L \otimes N) \cup \Gamma\ ;\ \rho\ ;\ B\ \rangle = \langle\ \Gamma'\ ;\ \gamma\ ;\rangle}$$

Figure 55. Rules for declarations.

Figure 56 shows the rules for directives.

Rule D10 is for inheritance. Rules D11 and D12 are respectively for cloning and cloning with re-declaration. Rules D13 and D14 are respectively for aggregations and aggregation re-definitions. Rule D15 is for deletions of attributes. Finally, rule D16 is for deletions of modeling elements.

Let Γ be a set of declarations and ρ be a path, $\Gamma \div \rho$ denotes the following set of declarations.

$$\Gamma \div \rho \quad = \quad \Gamma \setminus (\Delta_1 \cup \Delta_2 \cup \Delta_3 \cup \Delta_4)$$

where the Δi's are defined as follows.

$$\begin{array}{rcl}\Delta_1 & = & \{\textbf{port}\ \rho.\pi\ M \in \Gamma\} \\ \Delta_2 & = & \{\textbf{connection}\ \rho.\pi\ L\ M \in \Gamma\} \\ \Delta_3 & = & \{\textbf{block}\ \rho.\pi\ M \in \Gamma\} \\ \Delta_4 & = & \{\textbf{connection}\ \mu\ L\ M \in \Gamma;\ \rho.\pi \in L\}\end{array}$$

Δ_1, Δ_2, and Δ_3 are respectively the subsets of ports, connections and blocks of Γ such that ρ is a prefix of their path. Δ_4 is the subset of connections of Γ that refers to a port of Δ_1.

D10: $$\frac{(\textbf{block}\ \sigma\ M) \cup\ \Gamma\ ;\ \sigma\ ;\ \textbf{extends}\ C,\ \Theta\ |\ \textbf{class}\ C\ N\ B}{\Gamma'\ ;\ \sigma\ ;\ \Theta\ \ |\langle(\textbf{block}\ \sigma\ M \otimes N) \cup\ \Gamma\ ;\ \sigma\ ;\ B\ \rangle = \langle\ \Gamma'\ ;\ \gamma\ ;\rangle}$$

D11: $$\frac{\Gamma\ ;\ \sigma\ ;\ \textbf{clones}\ \pi\ \textbf{as}\ \mu,\ \Theta\ |\ [\![\Gamma,\sigma,\pi]\!] = \rho,\ [\![\Gamma,\sigma,\mu]\!] = \theta,\ \exists\ \textbf{block}\ \rho\ M\ B,\ \neg\exists\ \textbf{block}\ \theta\ N \in\ \Gamma}{\Gamma'\ ;\ \sigma\ ;\ \Theta\ \ |\ \langle(\textbf{block}\ \rho\ M) \cup\ \Gamma\ ;\ \rho\ ;\ B\ \rangle = \langle\ \Gamma'\ ;\ \gamma\ ;\rangle}$$

D12: $$\frac{(\textbf{block}\ \theta\ N) \cup\ \Gamma\ ;\ \sigma\ ;\ \ \textbf{clones}\ \pi\ \textbf{as}\ \mu,\ \Theta\ |\ [\![\Gamma,\sigma,\pi]\!] = \rho,\ [\![\Gamma,\sigma,\mu]\!] = \theta,\ \exists\ \textbf{block}\ \rho\ M\ B}{\Gamma'\ ;\ \sigma\ ;\ \Theta\ \ |\langle(\textbf{block}\ \theta\ N \otimes M) \cup\ \Gamma\ ;\ \theta\ ;\ B\ \rangle = \langle\ \Gamma'\ ;\ \gamma\ ;\rangle}$$

D13: $$\frac{\Gamma\ ;\ \sigma\ ;\ \textbf{embeds}\ \pi\ \textbf{as}\ \mu,\ \Theta\ |\ [\![\Gamma,\sigma,\pi]\!] = \rho,\ [\![\Gamma,\sigma,\mu]\!] = \theta,\ \neg\exists(\textbf{embeds}\ \rho\ \textbf{as}\ \tau) \in \Gamma}{(\textbf{embeds}\ \rho\ \textbf{as}\ \theta) \cup\ \Gamma\ ;\ \sigma\ ;\ \Theta}$$

D14: $$\frac{(\textbf{embeds}\ \rho\ \textbf{as}\ \tau) \cup\ \Gamma\ ;\ \sigma\ ;\ \textbf{embeds}\ \pi\ \textbf{as}\ \mu,\ \Theta\ |\ [\![\Gamma,\sigma,\pi]\!] = \rho,\ [\![\Gamma,\sigma,\mu]\!] = \theta}{(\textbf{embeds}\ \rho\ \textbf{as}\ \theta) \cup\ \Gamma\ ;\ \sigma\ ;\ \Theta}$$

D15: $$\frac{(\textbf{d}\ \rho\ (a = v) \cup M) \cup \Gamma\ ;\ \sigma\ ;\ \textbf{deletes}\ \pi,\ \Theta\ |\ \textbf{d} \in \{\textbf{port}, \textbf{connection}, \textbf{block}\}, [\![\Gamma,\sigma,\pi]\!] = \rho}{(\textbf{d}\ \rho\ M) \cup \Gamma\ ;\ \sigma\ ;\ \Theta}$$

D16: $$\frac{\Gamma\ ;\ \sigma\ ;\ \textbf{deletes}\ \pi,\ \Theta\ |\ [\![\Gamma,\sigma,\pi]\!] = \rho}{\Gamma \div \rho\ ;\ \sigma\ ;\ \Theta}$$

Figure 56. Rules for directives.

Appendix A. S2ML Textual Grammar

A.1. Comments

```
Comment ::=
      PARAGRAPH_COMMENT | LINE_COMMENT

PARAGRAPH_COMMENT ::=
      "/*" .* "*/"

LINE_COMMENT ::=
      "//" .* \n
```

A.2. Identifiers and Paths

```
Identifier ::=
      REGULAR_IDENTIFIER
    | STRING_IDENTIFIER

REGULAR_IDENTIFIER ::=
      ( ALPHA | "_" ) ( DIGIT | ALPHA | "_" )*

STRING_IDENTIFIER ::=
      "'" ( REGULAR_CHAR | ESCAPE_SEQUENCE )* "'"

STRING ::=
      """ ( REGULAR_CHAR | ESCAPE_SEQUENCE )* """

ALPHA ::=
      [a-zA-Z]

DIGIT ::=
      [0-9]

REGULAR_CHAR ::=
      ALPHA | DIGIT | "!" | "#" | "$" | "%" | "&" | "(" | ")"
    | "*" | "+" | "," | "-" | "." | "/" | ":" | ";" |
    | "<" | ">" | "=" | "?" | "@" | "[" | "]" | "^"
    | "{" | "}" | "|" | "~"

ESCAPE_SEQUENCE ::=
      "\'" | "\"" | "\?" | "\\" | "\a" | "\b" | "\f" | "\n"
    | "\r" | "\t" | "\v"

Path ::=
      Identifier ( "." Path ) ?
    | "owner" "." Path
```

```
    | "main" "." Path
```

A.3. Blocks and Classes

```
Model ::=
    ( BlockDeclaration | ClassDeclaration | PackageDeclaration
    | IncludeDirective )*

BlockDeclaration ::=
    "block" Path AttributeList? BlockDeclarationClause* "end"

ClassDeclaration ::=
    "class" Path AttributeList? BlockDeclarationClause* "end"

BlockDeclarationClause ::=
      PortDeclaration
    | ConnectionDeclaration
    | BlockDeclaration
    | InstanceDeclaration
    | ExtendsClause
    | ClonesClause
    | EmbedsClause
    | DeletesClause

PortDeclaration ::=
    "port" Path ( "," Path )* AttributeList? ";"

ConnectionDeclaration ::=
    "connection" Connection ( "," Connection )* AttributeList? ";"

Connection ::=
    Path? "[" ( Path ( "," Path )* )? "]"

InstanceDeclaration ::=
    Path Path ( "," Path )* AttributeList? ";"

PackageDeclaration ::=
    "package" Path AttributeList? PackageDeclarationClause* "end"

PackageDeclarationClause ::=
    ClassDeclaration | PackageDeclaration

AttributeList ::=
      "(" Attribute ( "," Attribute )* ")"

Attribute ::=
      Identifier "=" STRING
```

A.4. Clauses and Directives

```
ExtendsClause ::=
    "extends" Path AttributeList? ";"

ClonesClause ::=
    "clones" Path "as" Path AttributeList? ";"

EmbedsClause
    "embeds" Path "as" Path AttributeList? ";"

DeletesClause ::=
    "deletes" Path ";"

IncludeDirective ::=
    "include" STRING ";"
```

Appendix B. XML Schema

It is sometimes preferable, in order to simplify parsing and transformation of models, to use a XML based representation rather than a textual grammar. The XML representation for S2ML follows Open-PSA principles:

- The declaration of a named element of type “xxx” is introduced by means of a XML tag “declare-xxx”.
- References to a named element of type “xxx” are introduced by means of a XML tag “xxx”.

A XML schema (XSD) is available for S2ML together with some of the examples of this specification, in both plain text format and XML format.