From Models of Structures to Structures of Models

Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy

From Models of Structures to Structures of Models

Michel Batteux
IRT SystemX
8, avenue de la Vauve
91120 Palaiseau, France
Email: michel.batteux@irt-systemx.fr

Tatiana Prosvirnova
Laboratoire de Genie Industriel
CentraleSupélec, Université Paris-Saclay
8-10, rue Joliot-Curie
91190 Gif-sur-Yvette, France
Email: tatiana.prosvirnova@centralesupelec.fr

Antoine Rauzy
MTP, Norwegian University of
Science and Technology
S. P. Andersens veg 5
7491 Trondheim, Norway
Email: antoine.rauzy@ntnu.no

***Abstract*—The complexity of industrial systems is steadily increasing. To face this complexity, the different engineering disciplines are designing models. These models are complex as they reflect the complexity of systems under study. Therefore, they need to be structured.**

In this article we study structural constructs of modeling languages used in systems engineering. We introduce for that purpose a small domain specific language, the so-called S2ML for System Structure Modeling Language. We show that a large class of actual modeling languages can be (re)constructed by plugging their underlying mathematical framework into S2ML.

***Index Terms*—Modeling Languages, Model Structure**

I. Introduction

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 [2]. As systems are complex, one cannot expect models of these systems to be simple. They are complex. Therefore, they need to be organized and structured. The main purpose of this article is to study how to structure models.

Designing models has much to do with designing computer programs. To model, one needs a modeling language; just as to program, one needs a programming language. According to [3]: "Algorithms + Data Structures = Programs". We revisited this sentence and applied it to models:

"Behaviors + Structures = Models"

Most of modeling languages are actually the combination of a mathematical framework, e.g. differential equations (as for Modelica [10]) or Mealy machines (as for Lustre [13]), 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 different reasons. On one hand, a model is always a specific point of view on the system. It aims at capturing a particular aspect or at evaluating a particular property of that system. Therefore, it abstracts away irrelevant parts of the system and is useful only because it does so. On the other hand, the organization of a model has its own logic that can be quite different from the architecture of the system. It is clear that 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. Our thesis is 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 and that knowledge can be capitalized throughout successive projects.

Constructs to structure models derive from those to structure computer programs. In this article we introduce a small modeling language, the so-called S2ML for System Structure Modeling Language, which provides constructs to structure models. The behavioral part is left unspecified, i.e. just represented by terms that are left un-interpreted. The particular syntax of the language has no much importance. What is truly important is to ensure a one-to-one correspondence between the concepts and the syntactic constructs of the language. A large class of actual modeling languages can be (re)constructed by plugging their behavioral constructs into S2ML, i.e. by interpreting these terms.

Our contribution is therefore multiple. First, it studies model structuring at a conceptual level, i.e. independently of any concrete modeling formalism. Second, it gives an overview of structural constructs of modeling languages used in systems engineering. Third, it introduces S2ML that can be seen as the skeleton for many particular modeling languages. Finally, it illustrates the syntax, semantics and pragmatics of the different constructs of the language rather than to give a formal and complete definition of these constructs. The reader interested in a detailed specification can refer to [5].

The remainder of this article is organized as follows. Section II presents an example used throughout the article to illustrate S2ML constructs. Section III introduces basic elements of S2ML: blocks, ports and connections, and shows how they can be interpreted by different system engineering modeling languages. Section IV gives an overview of structural constructs and discusses the reuse of modeling components via blocks, classes and the associated mechanisms. Section V concludes the article by carrying out a general discussion about model design process and the role of languages such as S2ML in this process.

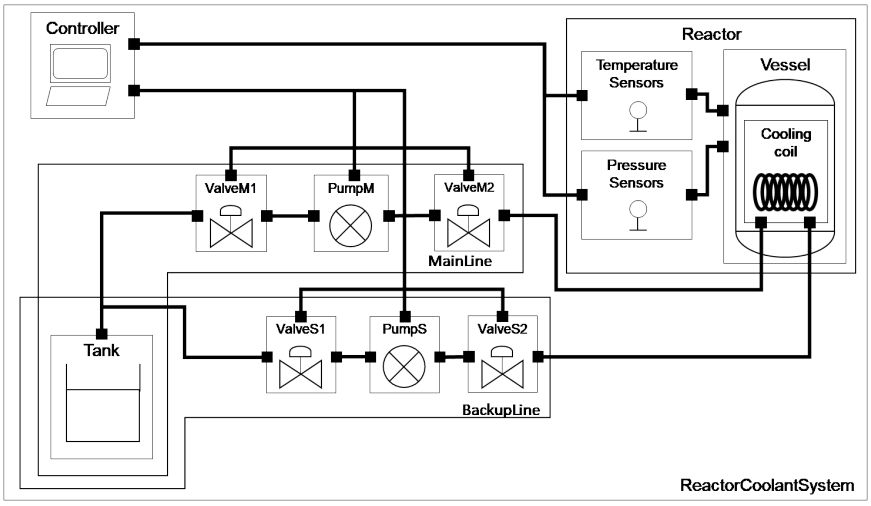

Fig. 1. A simplified reactor coolant system (partial view).

II. Illustrative Example

To illustrate S2ML concepts and notions presented throughout this article, we shall use an illustrative example inspired from safety engineering. The considered system is a simplified reactor cooling system. It consists of a reactor whose temperature and pressure must be maintained within acceptable limits thanks to a cooling system.

Fig. 1 gives a partial view of the physical architecture of this system. The main function of this system is to maintain the temperature of the reactor in a given range $[T_{min}, T_{max}]$. The temperature sensors measure the temperature of the reactor and provide the value to the controller.

The cooling water comes from a tank and is provided to the reactor via two different lines: a main one and a backup one, which can be used in case of failures. These two lines are composed of three actuators: a pump and two valves.

If the temperature gets lower than T_{min}, the controller commands the closing of the valve "ValveM1". If it gets higher than T_{max} then the controller commands the opening of this valve. If the valve "ValveM1" fails to open (or if the pump "PumpM" is failed), the temperature and the pressure in the reactor increase. Pressure sensors detect the excessive pressure and send the value to the controller which switches to the backup line and commands the opening of the valve "ValveS1".

The view is simplified and only partial because it does not show other classical elements of such systems, for instance, steam generator or pressurizer. Nevertheless, it is a good view of a classical system with strong safety requirements to take into account: the system should be designed so that no single failure could lead to the loss of the capability to cool the reactor.

III. Basic Elements: Blocks, Ports and Connections

The block is the basic S2ML modeling unit. A block has a name and may contain ports and connections (and several other modeling elements).

The three terms "block", "port" and "connection" are borrowed to the SysML terminology [9], 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 in programming languages, but also of modeling languages such as Modelica, Lustre or AltaRica. A connection is a relation between ports (we shall come back to this issue later).

A. Graphical Representation

In Fig. 1, ports are represented by black filled squares, connections by bold lines and blocks are delimited with light lines and are usually represented by rectangles. For instance, the rectangle "ValveM1" represents a block with three ports. Names of ports are not indicated on the figure so not to overload the graphics. Ports of the block "ValveM1" are connected to ports of the blocks "Tank", "Controller" and "PumpM". The represented model contains several other blocks, with some of them containing others (explained later c.f. Section IV-A).

In Fig. 1, all ports are represented on the border of blocks. This is by no means mandatory. Ports can be internal to a block. Fig. 2 shows for instance an internal view of the block "ValveM1". The port "state" is internal.

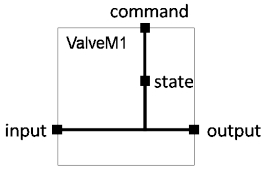

Fig. 2. Internal view of the block "ValveM1".

B. Textual Representation

Graphical representations, as those of Fig. 1 and Fig. 2, are excellent communication means. However, they quickly reach

their limit: as soon as the model ceases to be simple, they can only represent it partially. In other words, the model exists independently of its graphical representation. Moreover, there may be several graphical representations of the same concept, each having its own interest, depending on the context. That is the reason why S2ML is primarily a textual language. As previously pointed out, the particular syntax for this language is not important. What is really important is to ensure a one-to-one correspondence between the concepts and the syntactic constructs of the language.

The S2ML code for the block "ValveM1" is described in Fig. 3. It declares a block with four ports and two connections. The first connection is anonymous whereas the second is named. Blocks, ports and connections can carry information via lists of attributes. An attribute is a pair (name, value), where value is a string.

```
block valveM1
   port state (type="open/close", init="open");
   port input, output, command;
   connection [input, state, output];
   connection action[command, state];
end
```

Fig. 3. S2ML code for the block "ValveM1".

C. Interpretation

In S2ML, ports and connections are interpreted by themselves, i.e. a connection has no other meaning than "there is a relation between these ports". In other words, S2ML is a purely syntactic language. A language that would use S2ML as a structuring paradigm would redefine the syntax of ports and connections and more importantly would assign them a semantics. For instance, a S2ML version of Modelica would define connections as differential equations. A S2ML version of Petri nets (see e.g. [17] for an introduction) would define ports as places and connections as transitions. A S2ML version of AltaRica [20] would define ports as variables and events, and connections as transitions and assertions. As an illustration, Table I summarizes how different modeling languages can interprete blocks, ports and connections.

IV. Models of Structures

In this section we give an overview of stuctural constructs of modeling languages used in systems engineering and show how these constructs can be represented in S2ML.

A. Composition

The simpliest structuring relation is the composition: a system composes a component means that the component "is-part-of" the system.

Fig. 1 shows a hierarchical description: the block "ReactorCoolantSystem" is composed of the blocks "Reactor", "Controller", "MainLine" and "BackupLine". The block "MainLine" contains the blocks "Tank", "ValveM1", "PumpM" and "ValveM2".

Many modeling languages implement composition. For example, it is the case of SysML [9], AADL [8], Modelica [10] and all the versions of AltaRica [1], [20].

In S2ML, to compose a block in another block, it is sufficient to declare the former inside the latter.

The declaration of the block "ReactorCoolantSystem" could be as presented Fig. 4: the block "ReactorCoolantSystem" composes the blocks as well as the connections.

```
block ReactorCoolantSystem
   block Reactor
      // body of the block Reactor
   end
   block Controller
      // body of the block Controller
   end
   block MainLine
      // body of the block MainLine
   end
   block BackupLine
      // body of the block BackupLine
   end
   connection [Reactor.TemperatureSensors.output,
        Controller.input1];
   connection [MainLine.ValveM2.output,
        Reactor.Vessel.CoolingCoil.input1];
   connection [BackupLine.ValveS2.output,
        Reactor.Vessel.CoolingCoil.input2];
   connection [Controller.command1,
        MainLine.ValveM1.command];
   connection [Controller.command1, MainLine.PumpM.command];
   connection [Controller.command1,
        MainLine.ValveM2.command];
   // Same connections between the controller and
        components of the BackupLine
end
```

Fig. 4. S2ML code for composition.

The port "p" of block "B" is referred to as "B.p" as in most of the object-oriented languages (it is prefixed by the name of its parent).

In S2ML, it is possible to refer to a port (and more generally any modeling element) of a block nested at any hierarchical level, therefore "crossing the wall" of composing blocks. As connections represent any relations, there is actually no reason to limit the reference mechanism as if ports were physical interfaces. Note finally that each block is a name space: the blocks "ValveM1" and "PumpM" can both declare a port "input".

B. Component reuse via classes

S2ML blocks are prototypes, in the sense of object and prototype oriented programming languages ([4], [18]). A block has normally a unique occurrence. There are cases however where the system under study embeds several identical components. In our illustrative example, pumps "PumpM" and "PumpS" may be identical. It would be of course possible to duplicate the code of the two pumps, but this would be both error prone and confusing. A first solution to address this problem is to create a class for pumps and to instantiate it 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. The code of the reactor coolant system could be as depicted in Fig. 5. This code is equivalent to the one that would declare two identical blocks for pumps "PumpM" and "PumpS".

TABLE I
BASIC STRUCTURAL CONSTRUCTS

Concept	Blocks	Ports	Connections
AADL	Components (system, abstract, thread, device, etc.)	Features, etc.	Connections
SysML	Blocks	Ports	Connections
Modelica	Classes, Models, Blocks	Constants, Variables, Parameters	Equations
Lustre	Functional blocks	Variables	Instructions
AltaRica	Nodes	Variables, Events	Transitions, Synchronizations, Assertions
AltaRica 3.0	Blocks, Classes	Variables, Parameters, Observers, Events	Transitions, Assertions
Petri Nets	-	Places	Transitions

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

block ReactorCoolantSystem
   block MainLine
      block ValveM1
         // body of the block valveM1
      end
      block ValveM2
         // body of the block valveM2
      end
      Pump PumpM;
      connection [ValveM1.output, PumpM.input];
      connection [PumpM.output, ValveM2.input];
      // remainder of the block MainLine
   end
   // remainder of the block ReactorCoolantSystem
end
```

Fig. 5. S2ML code for class declaration and instantiation.

C. Component reuse via cloning

In addition to the class/instance mechanism to reuse components, S2ML provides another way to automatically duplicate code (a part of a model). It is called cloning. For instance, the code for the block "Reactor" could be as follows.

```
block Reactor
   block TemperatureSensors
      // body of the TemperatureSensors
   end
   clones TemperatureSensors as PressureSensors;
   // remainder of the body of Reactor
end
```

In the above code, the block "Reactor" defines a block named "TemperatureSensors" and clones it to get the block "PressureSensors". In this example, both blocks are local to the model and thus do not require any genericity.

The cloning directive is extremely powerful in conjunction with aggregation (discussed later). The S2ML specification document [5] provides a graphical representation for cloning.

D. Inheritance

In some cases it is necessary to modify or extend the characteristics of a modeling component/class without changing its nature. In these cases, composition is not really suitable because we would like to be able to substitute the modified/extended component for any occurrence of the original one.

It can be achieved using the inheritance mechanism. Inheritance describes a "is-a" relation between modeling units.

As an illustration, consider the valves of our illustrative example. The actual implementation for these valves could derive from an abstract description of what is the interface of a modeling component for a valve. This inheritance mechanism makes it possible to build step wisely the model.

As any object-oriented language, S2ML provides a construct to implement the inheritance mechanism: the "extends" directive. The code to model the valves of our illustrative example could thus be as follows.

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

class Valve
   extends ValveInterface;
   port state;
   connection [state, command];
   connection [input, output, state];
end
```

In the context of modeling languages, inheritance can be seen as a particular type of composition, where no prefix is added to the composed/inherited class or block. S2ML allows multiple inheritance, with all the usual warnings about it.

Object-oriented languages, such as AADL, SysML, Modelica and the new version of AltaRica, AltaRica 3.0, implement the inheritance mechanism.

E. Aggregation

We have seen that composition describes a "is-part-of" relation. This relation assumes that a component cannot belong to two different super-components (parents). There are cases however where the same component is used in several places or to contribute to different functions of the system. This kind of "uses" relations can be described by means of aggregation.

As an illustration, Fig. 1 shows a physical breakdown of the system: it is made of four sub-systems and some connections; these sub-systems are further decomposed, and so on. This structural breakdown is correct, although it mixes physical and functional descriptions. It is probably not the only decomposition of the system that would be used. The two lines for instance are more functional groupings than physical groupings. The first is the main and the second is the backup. Usually for such parts, failures occur on actuators (pumps or valves). The tank can be assumed as a passive component. Thus, only one tank exists and is shared by the two lines. Therefore, we need to be able to describe hierarchical

breakdowns with branches sharing components. This is the very purpose of the notion of aggregation. The S2ML construct for aggregation is the "embeds-as" directive.

The code describing the "Reactor Coolant System" could be as follows.

```
block ReactorCoolantSystem
   block Tank
   // body of the block Tank
   end
   block MainLine
      embeds main.Tank as tank;
      // remainder of the block MainLine
   end
   block BackupLine
      embeds main.Tank as tank;
      // remainder of the block BackupLine
   end
   // remainder of the block ReactorCoolantSystem
end
```

In the code above, the block "ReactorCoolantSystem" defines a block "Tank", which is then aggregated by the blocks "MainLine" and "BackupLine".

The aggregation mechanism is extremely powerful. It makes it possible to embed into the same model different views, including functional and physical views. It can also be used to share universal objects (e.g. physical constants) in a clean way.

Very few modeling languages implement aggregation. For instance, it is the case of SysML, where aggregation relation between blocks is represented as a white filled diamond. The new version of AltaRica, AltaRica 3.0, also implements aggregation by means of the "embeds-as" directive.

F. Blocks/Prototypes versus Classes

As we have seen earlier, S2ML provides two types of containers: blocks, which are prototypes in the sense of prototype-oriented programming, and classes (see e.g. [4] and [18] for discussions about these paradigms). Classes are well suited for "on-the-shelf", generic, stabilized knowledge. Prototypes are well suited for the "on-going" work.

The C-K theory of Hatchuel and Weill (see e.g. [14]) formalizes this idea in the context of engineering design. Their dialectic applies to the model engineering as well: the model (made of blocks/prototypes) is the space C of concepts, i.e. a sandbox in which new knowledge is maturing. Classes are the space K of stabilized knowledge.

The class/instance mechanism is without any doubt interesting to define "on-the-shelf", re-useable modeling components such as those for valves or pumps in our illustrative example. The fantastic richness of modeling languages such as Modelica or Matlab/Simulink stands for a great part in the wide choices of dedicated libraries.

However, when authoring a model (graphically or not), we may want to modify the code for the cooling coil while editing the entire model. With a pure object-oriented language, this is not possible. Classes are flat: one cannot modify a class via its instance, even if this instance is unique. We may call that the "a box in a box in a box" issue.

Moreover, what can be re-used depends on the level of abstraction of the model. On the one hand, models designed with simulation languages such as Modelica or Matlab/Simulink, stand at physical component level. For these models, re-use means mainly re-use of modeling components. On the other hand, models standing at system level, typically designed with modeling languages such as SysML or AltaRica [20] shows a rather different picture. At system level, each model is unique. Re-use can be however obtained by cloning and adjusting models. In other words, there are prototypical systems/models, in the sense of [15], but no generic system model from which other system/model could be derived just by setting some parameters. What can be re-used is thus modeling patterns (in the sense of design patterns [11]) rather than modeling components.

The above discussion explains why S2ML provides both prototypes and classes. The comparison between classes and blocks is summarized in Table II.

G. Unfolding and Flattening

Any S2ML model is semantically equivalent to an unfolded one, i.e. a model made of a hierarchy of nested/aggregated blocks, connections and ports. In the unfolded model all the instantiated/inherited classes and "clones" directives are transformed into blocks and all the references/paths to model elements are resolved according to the rewriting rules, the so-called unfolding rules.

Any hierarchical S2ML model is also semantically equivalent to a flat one, i.e. a model made of a unique block with ports and connections. As in languages such as Modelica, Lustre or AltaRica, this flat model is obtained by applying recursively rewriting rules, the so-called flattening rules. These rules "remove the walls" of containers (blocks and instances of classes). In the S2ML specification document, they are formally defined in a Structural Operational Semantics style (see e.g. [19]).

H. Summary

As we have seen in this section, prototypes, classes, composition ("is-part-of" relation), inheritance ("is-a" relation) and aggregation ("uses" relation) are the fundamental concepts of model structuring. They originate from mechanisms to structure programs.

Table III presents an overview of structural constructs of some modeling languages used in systems engineering.

V. Conclusion

In this article, we introduced the domain specific language S2ML. S2ML provides a minimal, yet extremely powerful, set of concepts to describe functional and physical breakdowns of systems on the one hand, and to structure models on the other hand. The design of S2ML obeys the rule: "concepts should come first". S2ML is an object-oriented language that embeds some prototype-oriented features as well. The authors strongly believe that prototype orientation is the most suitable structuring paradigm for models at system level of abstraction. As a future work, it would be interesting to see how to embed S2ML to the Eclipse Modeling Framework [23].

TABLE II
BLOCKS/PROTOTYPES VERSUS CLASSES

Concept	*Class*	*Block*
Definition	Generic component	Component having a unique occurrence in the model
Reuse	Instantiation and inheritance	Cloning and modifying
Usage	Multiple instances "On-the-shelf" component stored in a library	Unique occurrence "Sandbox"
C-K theory	K-space	C-space
Operations	Composition Inheritance	Composition Inheritance Aggregation Cloning

TABLE III
STRUCTURAL CONSTRUCTS OF MODELING LANGUAGES: SUMMARY

Language	*Prototypes*	*Classes*	*Composition*	*Inheritance*	*Aggregation*	*Cloning*
AADL	No	Yes	Yes	Yes	No	No
SysML	Yes	No	Yes	Yes	Yes	No
Modelica	No	Yes	Yes	Yes	No	No
Lustre	No	Yes	Yes	No	No	No
AltaRica DataFlow	No	Yes	Yes	No	No	No
AltaRica 3.0	Yes	Yes	Yes	Yes	Yes	Yes
S2ML	Yes	Yes	Yes	Yes	Yes	Yes

Developments around S2ML can be useful in two different ways. First, they can contribute to increase the awareness of the systems engineering community on the topics of model structuring. They can inspire system modeling language designers and provide them with conceptual tools to ensure the convergence of the structuring part of modeling languages. Second, S2ML 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, possibly written into two different languages, cannot in general be compared directly. The idea is thus to abstract them into a pivot language, e.g. S2ML, and to compare their abstractions. The development of a full S2ML modeling and simulation environment, including libraries of comparators and abstractors for different modeling languages is, at the time we write these lines, under specification.

S2ML is already used as a structuring paradigm for modeling formalisms dedicated to safety analyses. The future version of the Open-PSA model exchange format (see e.g. [7] for the previous version) relies on S2ML to define the meta-model of classical modeling formalisms such as Fault Trees, Reliability Block Diagrams, Event Trees and Markov chains (see e.g. [21] for an introductory book). In addition, the high-level modeling language AltaRica 3.0 (see e.g. [20]) makes a full use of S2ML as a structuring paradigm while Stochastic Guarded Transition Systems [22] provide the underlying mathematical framework of the language. As of today, AltaRica 3.0 is probably the most advanced language to support Model-Based Safety Assessment, the declension of Model-Based System Engineering for Reliability, Availability, Maintainability and Safety (RAMS) studies. At the time we write these lines, both Open-PSA and AltaRica 3.0 are still under development. However, both projects are sufficiently mature to draw preliminary conclusions that show the pertinence of S2ML.

S2ML is a very powerful formalism to describe systems breakdowns as well as structures of models. Many widely used modeling languages could be re-casted so to adopt S2ML as a structuring paradigm. There is however an intrinsic limitation to S2ML expressive power: S2ML assumes that the architecture of the system (and/or of the model) does not change (significantly) throughout the mission time. Systems with evolving architecture, i.e. systems whose number of components and interactions between components change through the mission time, in other words systems of systems (see e.g. [16]), are much more difficult to model. Extending S2ML ideas and principles to this kind of systems would be an extremely important step forward.

REFERENCES

[1] A. Arnold, A. Griffault, G. Point and A. Rauzy. The AltaRica language and its semantics. Fundamenta Informaticae, 34:109–124, 2000.

[2] Benjamin S. Blanchard and Wolter J. Fabrycky. Systems Engineering and Analysis. Pearson, Upper Saddle River, NJ 07456, USA, 2008

[3] Niklaus Wirth. Algorithms + Data Structures = Programs. Prentice-Hall, Upper Saddle River, New Jersey 07458, USA, 1976

[4] Mauricio Abadi and Luca Cardelli. A Theory of Objects. Springer-Verlag, New-York, USA, 1998.

[5] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. System Structure Modeling Language (S2ML). AltaRica Association, 2015. archive hal 01234903, version 1.

[6] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. Communications of the ACM, 9(9):671–678, September 1966.

[7] Steven Epstein and Antoine Rauzy. Open-PSA Model Exchange format, version 2.0d, 2008. http://www.open-psa.org.

[8] P. Feiler, D. Gluch, and J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Carnegie Mellon University. 2006.

[9] Sanford Friedenthal, Alan Moore, and Rick Steiner. A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann. The MK/OMG Press, San Francisco, CA 94104, USA, 2011.

[10] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Press, Hoboken, NJ 07030–5774, USA, 2003.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley professional computing series. Addison-Wesley, Boston, MA 02116, USA, October 1994.

[12] Adele Goldberg and David Robson. Smalltalk 80: The Language. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[13] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language lustre. Proceedings of the IEEE, 79(9):1305–1320, 1991.

[14] Armand Hatchuel and Benoit Weill. C-k design theory: an advanced formulation. Research in Engineering Design, 19(4):181–192, 2009.

[15] George Lakoff. Women, Fire, and Dangerous Things: What Categories Reveal About the Mind. The University of Chicago Press, Chicago, Illinois, U.S.A, April 1990.

[16] Mark W. Maier. Architecting principles for systems-of-systems. Systems Engineering, 1(4):267–284, July 1998.

[17] Tadao Murata. Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE, 77(4):541–580, April 1989.

[18] James Noble, Antero Taivalsaari, and Ivan Moore. Prototype-Based Programming: Concepts, Languages and Applications. Springer-Verlag, Berlin and Heidelberg, Germany, 1999.

[19] Gordon D. Plotkin. The origins of structural operational semantics. Journal of Logic and Algebraic Programming, 6061:315, 2004.

[20] Tatiana Prosvirnova, Michel Batteux, Pierre-Antoine Brameret, Abraham Cherfi, Thomas Friedlhuber, Jean-Marc Roussel, and Antoine Rauzy. The AltaRica 3.0 project for model-based safety assessment. In Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS2013, pages 127–132, York, Great Britain, September 2013. International Federation of Automatic Control.

[21] Marvin Rausand and Arnljot Hyland. System Reliability Theory: Models, Statistical Methods, and Applications, 2nd Edition. Wiley-Blackwell, Hoboken, New Jersey, USA, January 2004.

[22] Antoine Rauzy. Guarded transition systems: a new states/events formalism for reliability studies. Journal of Risk and Reliability, 222(4):495–505, 2008.

[23] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Professional, Boston, MA 02116, USA, December 2008.