On the Formal Execution of UML and DSL Models

Julien DeAntoni, Frédéric Mallet, Charles André
Aoste Team-Project
Université de Nice Sophia Antipolis
INRIA Sophia Antipolis Méditerranée
Email: `firstname.name`*@sophia.inria.fr*

Abstract

Model-Driven Engineering intensively uses models and model transformations. Transformation tools ensure that the target model conforms to the target metamodel, so that it is syntactically correct. However, there is few assistance, or none at all, to guarantee that the semantics is preserved during the transformation. This is mainly due to the absence of an explicit semantics within the models. Models bring the syntax while the related (application-specific) analysis tools bring their own semantics.

We propose here a model-driven approach to describe a formal and explicit semantics as a separate model. This formal semantics can then be attached to different UML /DSL models and a UML /DSL model can be executed with different semantics.

1. Introduction

To deal with complex systems, designers have always proceeded by building models that abstract away details to focus on the relevant aspects. In the domain of Distributed Real Time and Embedded Systems (DRES), adequate abstractions should allow early validation/verification of the system. Consequently, the model, and more important its underlying semantics, is often specific and driven by the expected kind of analysis.

For about thirty years, computer sciences have used various kind of models to abstract systems and perform analyses [1]. These models were first described by Domain Specific Language (DSL). DSL define entities that are sound in the targeted domain. Considering DRES and their need for analysis, these entities are then augmented with a formal semantics [2], [3], [4], [5], [6], [7]. One major problem of these approach is the multiplication of the languages. Moreover, for each proposed DSL, the associated tools must be developped. Two kinds of tools are particularly important: the (graphical) editors, which allow the model to be specified in a convenient way; and the analysis tools that manipulate the DSL. There are two main ways of analyzing models. First, a new analysis environment is developed for the specific use of this model. Second, a transformation is realized from the model to the input language of an existing analysis tool.

The problems emerging from such an approach are twofold:

- It is difficult to benefit from all languages because their semantics is described in various languages and the creation of bridges between these semantics require a fine knowledge about both semantics.
- Developing, for a given DSL, (graphical) editors, analysis tools or transformation models to other existing tools is a time-consuming activity.

An (complementary) alternative to defining a DSL is to use the UML (Unified Modeling Language) [8], a general-purpose modeling language. UML is used in various domains and specifically well accepted in domains where structuration is a key issue for success while verification/validation is a second order concern. This is due to needs for general concepts that deliberately introduce variation points in the semantics, itself described by using natural language. In DRES, analysis is a key issue, which explains the little use of UML. To enable verification/validation of UML models, a common approach is to use the profiling mechanism. A profile allows UML to be specialized, through stereotypes, with domain specific concerns. It is then possible to describe the semantics of the stereotypes and, based on this semantics, to make transformations to the input languages of existing analysis tools. Being a UML profile, it benefits from the large set of rapidly improving UML graphical editors. However, the semantics of the profile entities is either provided in a natural language from which transformations to analysis tool languages are derived, or scattered between the transformation and the semantics of the targeted language. In the first case, it is obvious that the semantics is only informal while in the second case it is hard to guarantee semantic equivalence between two transformations of the same model into different analysis languages.

The sum-up of these two main approaches is given in figure 1.

In this paper we are discussing the possibility to describe explicitly the semantics of a model as a separate model, which encapsulates a specific Model of Computation and Communication (MoCC) This MoCC may then be applied

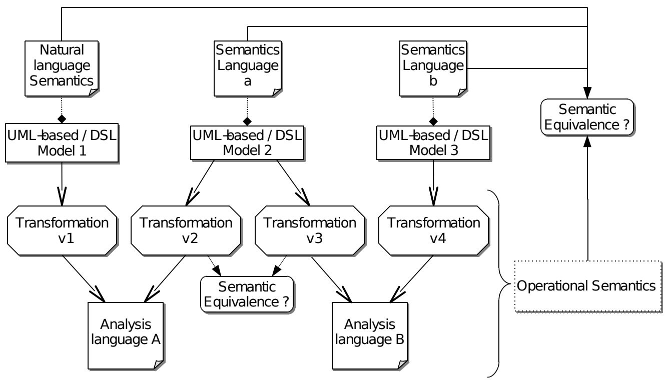

Figure 1. Current practice in using models for DRES

to one or more models. Consequently, on one hand it is possible to describe several semantics for a same model, and on the other hand, the formal semantics can be (tooled one for once(?)). Each developed tool or transformation models benefit from all models applying a MoCC described with our approach (?).

The second section of this paper clearly sum-up the objectifs of the proposed approach and identify the underlying challenges. The third section describes the formal language used while the fourth and the fifth section shows how it is integrated in a model based approach. Finally, before to conclude, already developed tools are presented.

2. Objectives and challenges

To build *semantic models* and attach them to models and transformation models, we need to address five challenges described in this section. Each of the following sections proposes a solution for each of these five steps.

1) Provide a language general enough for the description of various MoCCs and ensure a formal operational semantics.
2) Describe the concepts of the formal language defined at step 1 in a metamodel whose instances are MoCCs. To do so, the proposed metamodel must allow a kind of genericity to represent the various possible MoCCs as libraries. Moreover, it must be possible to apply these MoCC libraries on several different kinds of models ranging from UML models to DSLs.
3) Augment the metamodel described in step 2 with the formal operational semantics dexcribed in step 1. This step makes the instances of this metamodel executable according to the formal semantics descrided in step 1.
4) Build tools that implement the previous steps in order to provide various outputs ranging from user feedback for models (exhaustive) simulation to specific analysis tools.
5) Select and apply an executable MoCC on a specific model to make it executable.

The global objectives resulting from these 5 steps are described in figure 2, which illustrates the different kinds of relations between the different models. Moreover, this figure also presents some desired tools like the simulation engine or the tools that realize the tranformations. The simulation can run models according to a specific MoCC and produce a trace model (that conforms to its metamodel). In the end of this section, we focus on transformations since various kinds of transformations are identified: MoCC transformations, semantic transformations and trace transformations. The MoCC transformations consider transformations between two MoCCs. To figure out the sense of these transformations, the reader can imaging two MoCCs `A` and `B` where `A` is a refinement[1] of `B`. One can also imagine a semantically sound transformation between two very different MoCCs. The trace transformations manipulate the result of a specific simulation. This way it is possible to provide user feedbacks as well as bridges to analysis tools for specific analyses. The semantic transformations create a semantic bridge between the formal semantics of the very expressive formal language used in the approach and more specific analysis tool languages, often dedicated to a specific kind of analysis. This way, a specific semantic transformation α, a UML-based or a DSL model and its explicit MoCC can be transformed, according to α, into an analysis tool specific representation to perform the analysis. This last transformation is named derived trasformation on figure 2.

By implementing each of these steps, we provide a MDK (Model Development Kit) where MoCCS relying on a formal semantics can be described and used for UML-based and DSL model simulations and analyses. Among the five presented steps, we believe that the three first ones are the most challenging and must be realized in a way that make the two last steps as simple as possible. We have developed a MDK named `TimeSquare`, which covers each of the previously presented steps. For each step, the rational and choices made in the implementation of this MDK are described in the next sections.

3. Formal language for MoCC description

The idea developed in this paper is to provide a language that allows the description of a specific operational semantics for the description of MoCCs, namely CCSL (Clock Constraint Specification Language). The operational semantics of CCSL is given as SOS [9] and allows the associated abstract machine to execute models whose MoCC description is described in CCSL.

CCSL is a declarative language whose result is a CCSL *system*. A CCSL *system* is composed of *clocks* and *clock constraints* imposed on the *clocks*. A clock constraint can be either a *Relation* or an *Expression*. One can notice that

1. `A` is `B` with additionnal constraints

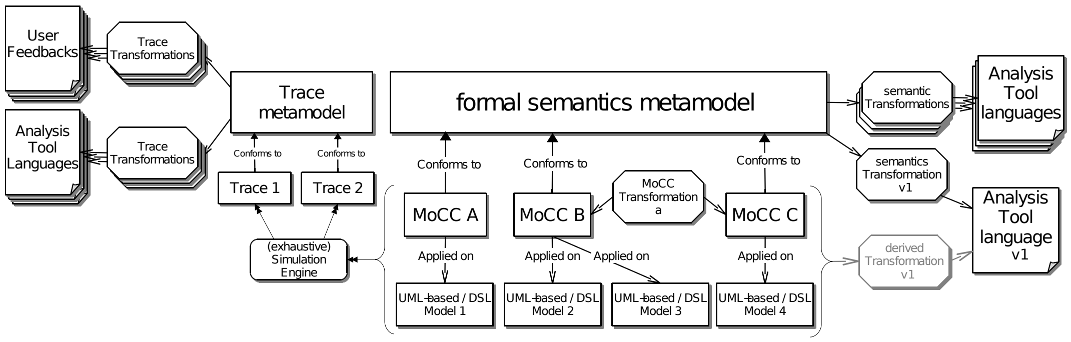

Figure 2. Proposed approach, centralized around a formal language for MoCC description

Clock is perhaps a misleading name since it is closer to an "activation conditions" rather that the physical device that measures time. However, this name comes from the strong connections between CCSL and the MARTE Time Model [10]. *clock*, CCSL *system* and *clock constraints* are detailed in the next sub-sections.

3.1. Clock and CCSL system

A *Clock* is an ordered set of instants ($\mathcal{I}$), where $\prec$ is a quasi-order relation on $\mathcal{I}$, named *strict precedence*.

A *discrete-time clock* c is a clock with a discrete set of instants $\mathcal{I}$. Since $\mathcal{I}$ is discrete, it can be indexed by natural numbers in a way that respects the ordering on $\mathcal{I}$. $c[k]$ denotes the k^{th} instant. Moreover, in the discrete case, each instant, but the first one, has a unique direct predecessor.

A set of *clocks* constrained by *clock constraints* defines a CCSL *system*. More formally, a CCSL system is a pair $\langle C, \preccurlyeq \rangle$ where C is a set of clocks and where $\preccurlyeq$ is a binary relation on $\bigcup_{c \in C} \mathcal{I}_c$, named *precedence*. $\preccurlyeq$ is reflexive and transitive. From $\preccurlyeq$ we derive four new instant relations: *Coincidence* ($\equiv$ is defined by $\preccurlyeq \cap \succcurlyeq$), *Strict precedence* ($\prec$ is defined by $\preccurlyeq \setminus \equiv$), *Independence* ($\parallel$ is defined by $\overline{\preccurlyeq \cup \succcurlyeq}$), and *Exclusion* ($\#$ is defined by $\prec \cup \succ$).

Instant relations are defined on pairs of instants. This is obviously not suitable for a time structure specification. Instead we have defined constraints on clocks: a clock constraint imposes many—usually infinitely many—instant constraints.

3.2. Clock constraints

A clock constraint can be either a clock relation or a clock expression. A clock relation refers two clock specifications that can be either a clock or a clock expression. It constraints the partial ordering of clock instants of the clock specifications it refers to. A clock expression is a mean to construct a new clock from existing clock specification(s) and optionally additional parameters. A more detailed view of these two kinds of constraints follows.

3.2.1. Clock relations. Clock relations can be divided into three categories: *synchronous*, *asynchronous*, *mixed*.

Synchronous clock constraints rely on *coincidence*. *Subsets* is such a constraint: each instant of the *subclock* must coincide with one instant of the *superclock*. Of course, the mapping must be order-preserving.

Asynchronous clock constraints are based on *precedence*. Clock a *strictly precedes* clock b if for all natural number k, the k^{th} instant of a precedes the k^{th} instant of b ($\forall k \in \mathbb{N}^{\star}, a[k] \prec b[k]$).

Mixed clock constraints combine both coincidence and precedence. For instance the *sampling* constraint: $c = a$ sampledOn b constraints c to tick synchronously with b whenever a tick of a precedes a tick of b.

3.2.2. Clock expressions. A clock expression has an internal clock that is coincident with the clock resulting from the expression evaluation whose semantics is given by SOS rules detailed in the next section.

There are three types of clock expressions: *terminating*, *non-terminating* and *conditional*. A terminating expression is a expression that has a clock whose set of instants ($\mathcal{I}$) is finite. *UpTo*, which takes two clock specifications c_1 and c_2 as parameters, is such an expression: it produces a clock coincident with c_1 until the next instant of c_2. After the next instant of c_2, the resulting clock is said to be *dead*; i.e. the clock will never tick again.

Conversely, a non-terminating expression has an *a priori* non-finite clock, i.e. a clock whose set of instants is infinite. However the death of a clock can be propagated so that nothing prevent a non-terminating expression from producing a finite clock depending on the clock specifications used as parameter in the expression. *union*, which takes two clock specifications c_1 and c_2 as parameters, is such an expression: it produces a clock whose an instant is add to $\mathcal{I}$ when an instant is add to $\mathcal{I}$ of c_1 or to $\mathcal{I}$ of c_2.

Finally, a conditional expression has a clock that is coincidenct with a clock specification c_1 or a clock specification c_2 depending on a boolean condition. c_1 and c_2 can be both terminating or not. The owned clock dies as soon as the boolean expression specifies a coincidence with a dead clock specification.

The notion of dead clock is specifically important to build the the *Concatenation* expression. *Concatenation* takes two clock specifications c_1 and c_2 as parameters. c_1 concatenated with c_2 means that the owned clock is coincident with c_1 as long as c_1 is not dead. When c_1 dies, the owned clock is then coincident with c_2. It is also important to notice that this expression can be recursive.

3.2.3. Semantics of clock constraints. A CCSL *system* is a dynamic system and its behavior is defined by an infinite sequence of *steps*. A step consists of simultaneous clock ticks. When a (discrete) clock ticks, its current (local) index is incremented by 1. We call *configuration* of a time structure $\langle C, \preccurlyeq \rangle$ a mapping $c : C \to \mathbb{N}$. For each discrete clock clk, $c(clk)$ is the current index of clock clk. This index denotes the *current instant* of clk.

For a set of clocks subject to a conjunction of clock constraints, the challenge is, "given a configuration, determine a step that meets all the constraints". There may be 0 (inconsistent constraints), 1 (deterministic) or several satisfying steps (non deterministic).

To address this challenge, we have endowed CCSL with a Structural Operational Semantics. We defined SOS rules for a *kernel* CCSL (less than twenty rules) [11]. For illustration purpose, consider the "strictly precedes" relationship $c_1 \boxed{\prec} c_2$.

$$\frac{\begin{array}{l} c_1, c \vdash \mathsf{b}_1 \\ c_2, c \vdash \mathsf{b}_2 \\ \mathsf{b} \triangleq (c(c_1) = c(c_2)) \end{array}}{c_1 \boxed{\prec} c_2, c \vdash \mathsf{b}_1 \wedge \mathsf{b}_2 \wedge (\mathsf{b} \Rightarrow \neg \mathsf{c}_2)} \quad \text{(strictly precedes)}$$

This rule reads that, for the given configuration c, constraint "c_1 strictly precedes c_2" implies the Boolean expression on the right-hand side. In this rule, c_k is a Boolean variable associated with clock c_k. $\mathsf{c}_k = \mathsf{true}$ means that c_k can tick. The Boolean expression refers to Boolean expressions (b_1, b_2) attached to the concerned clocks (c_1, c_2), and imposes additional logical constraints, specific to the precedence relation: $(\mathsf{b} \Rightarrow \neg \mathsf{c}_2)$ or equivalently $(\neg \mathsf{b} \vee \mathsf{c}_2)$.

Clock constraints not defined in the kernel CCSL can be specified by composing primitive clock constraints.

A clock relation SOS rule is the same for all the CCSL system execution. However, expressions can be rewritten to change their internal state on given condition(s). Applying *rewriting rules* for all expressions yields a new set of clock constraints. For instance, considers the *UpTo* expression, which takes two clock specifications c_1 and c_2 as parameters (noted $\lightning$). The associated SOS rules for a given step is:

$$\frac{\begin{array}{c} c_1, c \vdash \mathsf{b}_1 \\ c_2, c \vdash \mathsf{b}_2 \end{array}}{c_0 \boxed{=} c_1 \lightning c_2, c \vdash \mathsf{b}_1 \wedge \mathsf{b}_2 \wedge \mathsf{ite}(\mathsf{c}_2, \neg \mathsf{c}_0, \mathsf{c}_0 = \mathsf{c}_1)} \quad \text{(UpTo)}$$

This specifies that if c_2 ticks, the owned clock (named c_0 in the SOS rule) can not tick unless the owned clock c_0 coincides with c_1. A rewriting rule must be given to specify that once c_2 ticks, c_0 must be dead at the end of the step. This is the meaning of the following rule:

$$\frac{c_2 \in F}{c_0 \boxed{=} c_1 \lightning c_2 \to c_0 \boxed{=} \mathbf{0}} \quad \text{(UpTo rewrite)}$$

In the UpTo rewrite SOS rule, we introduce $\mathbf{0}$ that is a special definition of a clock that never ticks. It specifies the semantics of a dead clock. Moreover, this SOS rule introduces $t_2 \in F$, which means that c_2 ticks at the current step, a clear explanation of that follows.

From a CCSL specification we derive a set of Boolean expressions. Let B be the conjunction of all these expressions. Starting with B, we determine the set of all possible (logical) solutions. From this set we deduce the set E of *Enabled Clocks*. A subset F (*Fired Clocks*) of E characterizes the new *step*. Not all subsets of E are correct solutions because a step must contain all or none of the clocks that have coincident instants. To derive F from E, the user chooses among different policies: minimal solution, maximal solution, random selection, and user's defined policies. The default policy is the random policy that selects one out of all the correct solutions.

Beyond activation conditions and constraints on them, we have to refer to the different modeling elements of a system that can be of interest in order to be able to describe various MoCCs. This is one of the purpose of the CCSL metamodel presented in the next section.

4. MetaModeling CCSL in an extendable way

To be fully integrated with UML-based and DSL models and to benefit from the MDE tools and facilities, it is important to be able to create CCSL model, which represents a CCSL *system*. CCSL provides a set of *kernel constraints* classified through *expressions* and *relations*. However, to be able to specify the possibly complex relations of a MoCC, it is important to be able to combined the *kernel constraint* to construct a library, which contains the MoCC relations. Finally, a new library may be built with relation(s) from one or more existing libraries; the metamodel must take this into account. This section overviews the proposed metamodel and the mechanism involved in the construction of libraries for MoCC.

Figure 3 shows a simplified metamodel of CCSL: a CCSL system is a set of clocks and constraints. A constraint can be a Clock relation, which constraints together a right and

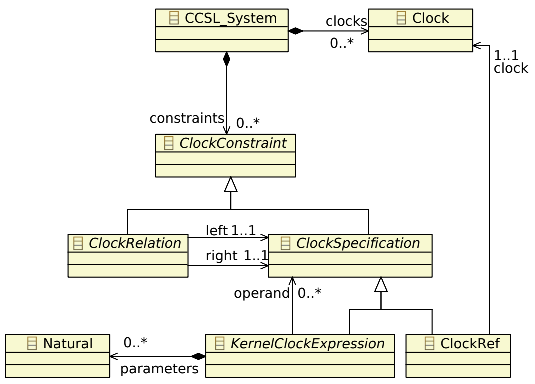

Figure 3. Simplified classical CCSL metamodel

a left clock specification. A Clock specification is either a clock expression or a simple reference to a clock. Finally, a clock expression can have parameters, simply represented by natural for short.

Using such a metamodel (of course more complex to be able to represent all the CCSL kernel) allows models to be defined. However, it is not possible to create reusable composition of clock constraints. What is expected is to define the equivalent of classical programming language functions, that can be defined and then called with the appropriate parameters when necessary. To do so, we propose a metamodel allowing the construction of library as the one presented in figure 4. This metamodel is a simplication of the one actually used. However, it is sufficient to understand the main principle. A library is a set of definition. A definition can be considered as a function definition in a classical programming language. A definition contains entities that can be either concrete or abstract. An abstract entity can be considered as a parameter in the prototype and in the definition of a function in a classical programming language. Following the same comparison, a concrete entity can be considered as an existing object in a classical object oriented programming language. A concrete relation is a concrete entity that is defined by a clock relation definition. It can be considered as a specific call to the function definition that is the clock relation definition. There are two possible uses of a concrete relation. It can be used in a definition to add some definition internal constraints or in a CCSL system to specify the application of a specific library on a Model to represent a MoCC. When used in a CCSL system, a concrete relation must bind with the variables of its definition some concrete elements in the same idea that a function call associates effective parameters to the formal parameters of the function definition. A binding is then used to link together a variable and a concrete entity. When a concrete relation is used in a definition, the bindings can either link the variables of their containing concrete relations to a concrete entity or to variables of the definition where it is used. For this reason, a binding links together a variable with either another variable

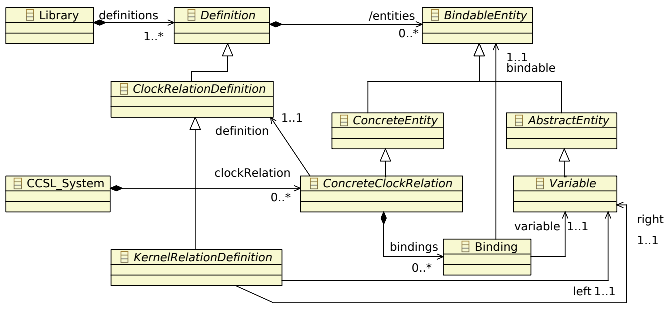

Figure 4. Simplified extendable CCSL metamodel

or a concrete entity depending on the case.

Finally, to be able to apply a specific CCSL system on a UML-based / DSL model, a *clock* owns an event that refers to an *EObject*, i.e. that can refer to every object of every eclipse based model. This way, applying a MoCC on a specific model consists in setting the reference between the event of a Clock and a UML-based / DSL model entity.

5. Augmenting the CCSL metamodel with the CCSL semantics

In order to avoid loss of information, it was important to stay in the modeling world so that no “opaque” transformations are necessary. To do so, we added the semantic directly at the metamodel level by using KerMeta [12]. KerMeta is a metamodelling language compliant with eMOF. One of the key features of KerMeta is a static composition operator, which allows extending an existing metamodel with new elements (such as properties, operations, constraints or classes). This operator allows defining various aspects of a metamodel such as structure, constraints, semantics or transformations in separated units and integrating them automatically. The composition is done statically and the composed model is typed-checked to ensure the safe integration of all units. This mechanism allows the operational semantic to be specified without any modifications on the Ecore metamodel. We specified the operation semantic of CCSL in KerMeta and weaved the operations that specify how the CCSL concepts are executed according to the SoS rules previously presented. Moreover, because a binding mechanism was used, we used KerMeta to navigate through the model and to resolve, at run-time, the binding used by expresiosn and relations of user defined Libraries.

By implementing the SOS rules and rewriting rules as operations of the metamodel entities, we obtained a CCSL metamodel, whose conforming models can be interpreted by the KerMeta framework.

6. Existing tooling and facilities

TIMESQUARE2 is the software environment we have developed to support the modeling approch presented in section 2.

TIMESQUARE2 has four main features: 1) modeling of user defined libraries, 2) modeling of CCSL system and applying it to a specific model, 3) generation of a solution, 4) displaying and exploring waveforms, animating UML-based model and storing the result inside the model.

TIMESQUARE2 provides a basic environment for model specification (both libraries and CCSL systems) in eclipse. This environment also allows a simple way to apply a specific CCSL model on a specific UML-based / DSL model that are described in eclipse. Based on the KerMeta framework, it is possible to execute the CCSL system in order to simulate the UML-based / DSL models. A very crude version of the exhaustive simulation is also possible. The simulation of a CCSL system allows the generation of traces, given as waveforms written in VCD format. VCD (Value Change Dump) [13] is an IEEE standard textual format for dumpfiles used by EDA (Electronic Design Automation) logic simulation tools.

Waveforms can be displayed with any VCD viewer. TIMESQUARE2 has its own viewer enriched with interactive constraint highlighting and access facilities.

For UML-based models graphically modeled with papyrus[2], it is also possible to animate the model. It is then possible to interactively navigate in the steps to see the state of the model entities for which CCSL specifications are applied on. Moreover, the state of model elements are store directly inside the UML model.

It is important to notice that TIMESQUARE2 uses a common trace model for all outputs. This trace model keep the trace between the model entities, the CCSL specification and the resulting model element states. Moreover, it keeps additional information like the internal state of the CCSL system, which can be usefull for better user feedback. Finally, the result of the (exhaustive) simulation can be injected in the CADP [14] model checker in order to be analysed.

7. Conclusion

In this paper we presented ongoing work for better manipulation of models with different operational semantics. To do so, we explicitly specify a formal operational semantics of a model as a separated model. It is then possible to explore the impact of different operational semantics on a same model. By providing a based language for MoCC description, it also provides facilities to take benefits of different tools developed for specific models. Amongst the several perspectives for this work, we can cite: providing better user feedback during model animation, facilitating the application of a MoCC on a specific model, providing an automatic selection of efficient analysis tool based on the MoCC description and the desired analysis, etc

2. http://www.papyrusuml.org

References

[1] F. DeRemer and H. Kron, "Programming-in-the large versus programming-in-the-small," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM Press, 1975, pp. 114–121.

[2] S. Faucou, A.-M. Déplanche, and Y. Trinquet, "An ADL centric approach for the formal design of real time systems," *In Architecture description language, IFIP*, pp. 67–82, 2004.

[3] project EAST-EEA, "Definition of language for automotive embedded electronic architecture," *Version 1.02*, 2004.

[4] S. Vestal, "Metah user's manual - version 1.27," 1998.

[5] P.H.Feiler, B. Lewis, and S. Vestal, "the sae avionic architecture description language (aadl) standard: A basis for model-based architecture driven embedded systems engineering," *RTAS Workshop on Model-Driven Embedded Systems*, 2003.

[6] R. Allen, "A formal approach to software architecture," Ph.D. dissertation, Carnegie Mellon, School of Computer Science, January 1997, issued as CMU Technical Report CMU-CS-97-144.

[7] J. et al., "The cotre project : rigorous software development for real-time systems in avionics," *IFAC/IFIP/IEEE Workshop on Real-Time Programming*, 2003.

[8] OMG-UML, "Final ftf report mof 2.0 core and uml 2.0 infrastructure finalization task force," *http://www.omg.org*, 2004.

[9] G. D. Plotkin, "A structural approach to operational semantics," 1981.

[10] The ProMARTE Consortium, *UML Profile for MARTE, beta 2*, Object Management Group, June 2008, OMG document number: ptc/08-06-08.

[11] C. André and F. Mallet, "Combining CCSL and Esterel to specify and verify time requirements," INRIA, Research Report RR-6839, 2009. [Online]. Available: http://hal.inria.fr/inria-00360528/en/

[12] P. Muller, F. Fleurey, and J. Jezequel, "Weaving Executability into Object-Oriented Meta-languages," *Model Driven Engineering Languages and Systems: 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005: Proceedings*, 2005.

[13] IEEE Standards Association, *IEEE Standard for Verilog Hardware Description Language*, Design Automation Standards Committee, 2005, IEEE Std 1364TM-2005.

[14] J. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, "CADP-A Protocol Validation and Verification Toolbox," *Proceedings of the 8th International Conference on Computer Aided Verification*, pp. 437–440, 1996.