Modeling the Behavioral Semantics of Heterogeneous Languages and their Coordination

Julien Deantoni

Modeling the Behavioral Semantics of Heterogeneous Languages and their Coordination

Julien Deantoni
University of Nice
I3S CNRS laboratory
INRIA AOSTE team
2004 routes des colles, 06902 Sophia Antpilois
Email: julien.deantoni@polytech.unice.fr

***Abstract*—In the software and system modeling community, research on domain-specific modeling languages (DSMLs) is focused on providing technologies for developing languages and tools that allow domain experts to develop system solutions efficiently. Unfortunately, the current lack of support for explicitly relating concepts expressed in different DSMLs makes it very difficult for software and system engineers to reason about information spread across models describing different system aspects. As a particular challenge, we present in this paper how we dealt with relationships between heterogeneous behavioral models to support their concurrent and coordinated execution. This was achieved by providing dedicated meta-language to define the behavioral semantics of DSMLs and their coordination. The approach made explicit a formal model of the control flow (MoCC); domain-specific actions (DSA) and a well-defined protocol between them (incl., mapping, feedback and callback) reified through explicit domain-specific events (DSE). The protocol is then used to infer a relevant behavioral language interface for specifying coordination patterns to be applied on conforming executable models. As a result, heterogeneous languages and their relationships can be developed in the GEMOC studio, which provides extensive support to run and debug heterogeneous models. This is outlined in the paper on the definition of the Marked Graph language and its coordination with a scenario language.**

I. Introduction

The development of complex software intensive systems involves interactions between different subsystems. These subsystems can be deployed on different kinds of resources (general-purpose processors, SoC, GPU), connected through a wide range of heterogeneous communication resources (buses, networks, meshes). To cope with this heterogeneity, the design of complex systems often relies on several Domain Specific Modeling Languages (DSMLs) that may pertain to different theoretical domains with different expected expressiveness and properties. As a result, several models conforming to different DSMLs are developed and the specification of the overall system becomes heterogeneous in terms of syntax but also in terms of behavioral semantics.

To understand the system globally and its emerging behavior, it is necessary to specify how the models are related to each others, in both a structural and a behavioral way. This problem is becoming more and more important with the globalization of modeling languages [1].

Whereas the MDE community provides some extensive support for the structural composition of models and languages (*e.g.*, [2], [3]), it provides few support for behavioral coordination. Such heterogeneous coordination has been studied both in the coordination languages and Architecture Description Language (ADL) communities [4], [5], [6], [7], [8], [9], [10]. In current coordination approaches the coordination is manually defined between particular models (or components). This is usually done by integrator experts that apply some coordination patterns according to their own skills and know-how.

The work done along the GEMOC ANR project[1], inspired by the approaches from different communities, we aimed at providing simulation and/or verification capabilities for systems specified with heterogeneous behavioral models. This was achieved in two main steps. The first step consisted in making explicit the behavioral semantics of a DSML and the second one, based on the explicit behavioral semantics consists in defining coordination patterns between heterogeneous languages.

Making the behavioral semantics explicit required to define appropriate meta languages. Exactly like the abstract syntax of a language which is defined with a meta language like BNF or eMOF can take advantages of some tooling based on these meta-languages (*e.g.*, parser/lexer), we proposed a meta language to specify the concurrency semantics of a DSL [11]. To specify the whole behavioral semantics of a DSL, we also proposed a specific modeling pattern [12], [13]. The next section of this paper presents these results.

To deal with the coordination of the models conforming to different languages, we proposed to use the events of the concurrency semantics as coordination points, defined in intention on the language level. Then, we expressed *coordination patterns* at the language level. The coordination patterns can be applied on heterogeneous models to automatically generate the coordination. Some existing approaches like Ptolemy [14] or Modhel'X [15] were already specifying some coordination patterns. In our approach, instead of hard coding the patterns in a tool, we proposed a meta language dedicated to the specification of behavioral coordination patterns: BCOoL (Behavioral

[1] http://gemoc.org/ins

Coordination Operator Language [16]). Section III presents our approach about the coordination of heterogeneous models.

Finally, all these meta-languages being formally defined, the individual models but also the coordinated models are amenable to verification and validation, either by simulation in the GEMOC studio (with model animation, back tracking, etc) or by construction of the scheduling state space (construction of all the specified execution paths, which takes into account the inter-leavings due to the concurrency). These possibilities are outlined on an heterogeneous example.

II. Modeling the language behavioral semantics

The specification, design and tooling of DSMLs leverage the rich state of the art in language theory. Several metamodeling environments support the specification of the syntax and the (static and dynamic) semantics of a DSML. These two elements of a DSML specify the domain-specific concepts, as well as the meanings of domain-specific actions that manipulate these concepts[2]. A significant limitation of current metamodeling environments comes from the implementation of the behavioral semantics, which usually intermingles the control flow and the data manipulation. Considering an operational semantics, both parts are typically embedded in the language used to implement the behavioral semantics (*e.g.*, if an operational semantics of the language is implemented in Java, the control flow is split in the different methods, together with the code that manipulates the data from the model).

There are three main drawbacks in these approaches: 1) both concurrency and timing aspects are kept implicit in general purpose code; 2) During the simulation of a model, the concurrency and timing aspects are strongly dependent on the environment used to execute the behavioral semantics (*e.g.*, the model simulations ran on a dual core and on an octocore can be different while the model is the same); 3) the non determinism resulting from different inter-leavings or synchronous actions is hidden to the designer, making difficult the understanding of concurrent and/or temporal aspects of the model.

Furthermore, having an implicit control flow also makes difficult the distinction of semantic variants in a model. For example, the fUML specification identifies several semantic variation points. As stated in the fUML specification, some semantic areas "are not explicitly constrained by the execution model: The semantics of time, the semantics of concurrency, and the semantics of inter-object communications mechanism" [17]. The lack of an explicit control flow, including concurrency, time and communication, prevents one from understanding the impact(s) of these variation points on the execution of a conforming model.

To avoid these issues, we proposed in [12] to reify the control flow as a metamodeling facility. We leveraged formalization work on concurrency and time from concurrency theory, specifically, theoretical work on tagged structures [18] and on heterogeneous composition of models of computation [19], [14]. Based on this, we defined a DSML as a 4-tuple whose units are defined in the remainder of the section. To illustrate our proposal, we use as a running example the Marked Graph language[3].

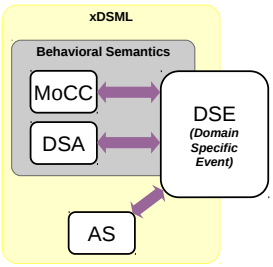

Fig. 1. The different units composing an executable DSML

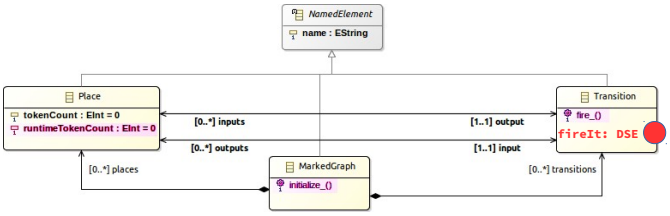

Fig. 2. the Marked Graph Abstract Syntax (with the execution state and the DSA highlighted)

a) AS: : The Abstract Syntax (AS) specifies the concepts of the language and their relationships. An instance of the AS is a model (see the simple Marked Graph metamodel on Figure 2).

b) DSA: : The Domain Specific Actions (DSA) represent both the execution state and the execution functions of a DSML. An instance of the DSA represents the state of a specific model during the execution and the functions to manipulate such a state. Note that there is no (visible) control flow in the execution functions. In the Marked Graph language, the number of token in a *Place* during the execution constitutes the execution state of the model (See the attribute highlighted in Figure 2). Also, to make a Marked Graph model evolving, two execution functions are required: one to *initialize* the model and one to *fire* a transition. Both the execution state and the execution functions are defined as aspects, woven to the abstract syntax by using Kermeta [4]. Listing 1 shows the implementation of the *fire* method, which moves a token from its input(s) to its output(s).

[2]Examples of metamodeling environments include Microsoft's DSL tools http://www.microsoft.com/en-us/download/details.aspx?id=2379, Eclipse Modeling Framework (EMF) http://www.eclipse.org/modeling/emf/, Generic Modeling Environment (GME) http://www.isis.vanderbilt.edu/Projects/gme/, and MetaEdit+ http://www.metacase.com/mep/

[3]http://en.wikipedia.org/wiki/Marked_graph

[4]http://kermeta.org/

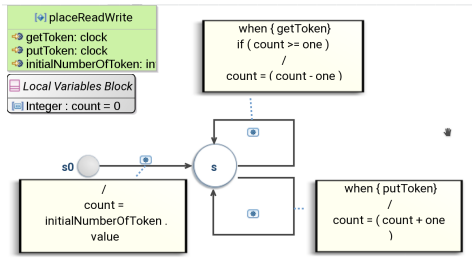

Fig. 3. An event constraint defined in MoCCML

```
@Aspect(className=Transition)
class TransitionAspect {
  def public void fire() {
    println("Transition "+_self.name+":fired")
    _self.inputs.forEach[
      runtimeTokenCount = runtimeTokenCount-1
    ]
    _self.outputs.forEach[
      runtimeTokenCount = runtimeTokenCount+1
        ]
  }
}
```

Listing 1. Definition of the *fire* DSA with Kermeta

c) MoCC: : The Model of Concurrency and Communication (MoCC) represents the control flow aspects in a language, including the concurrency, the synchronizations and the, possibly timed, causality relationships between the execution functions. An instance of a MoCC, Name *execution model*, is defined for a specific model that conforms to the DSML. The execution model specifies, by using in the CCSL (Clock Constrain Specification Language [20]) language, the correct partial orderings between the calls to the execution functions (more details can be found in [11]. For instance in the Marked Graph language, a place is initialized with a specific number of token and can be read if there is at least one token in it. This is encoded by the MoCCML automata presented in Figure 3. Then a transition can be fired if it can read from all its inputs.

d) DSE: : The Domain Specific Events (DSE) represent the link between the MoCC, the DSA and the AS to establish the whole behavioral semantic. It is a set of domain specific events, which are mapped with

- the execution functions from the DSA,
- the events constrained by the MoCC;
- the result of the execution functions (to prune correspondingly the concurrency model when the data flow is data dependent);
- the concepts from the AS to specify how the execution model must be generated for a specific model that conforms to the DSML

The DSE are the observable events from the user point of view, for instance when debugging step by step. The reader can refer to [13]) for more details on the DSE.

```
context Transition
 def: fireIt: Event = self.fire()

context Place
 inv readIfAtLeastOneToken:
  Relation MG_Place(self.output.fireIt,
   self.input.fireIt, self.tokenCout)
```

Listing 2. The *fireIt* DSE linked to an execution function and constrained by the MoCC.

Defining a language according to this 4-tuple results in a behavioral semantics in which the control flow is made explicit and cleanly associated to the data manipulation to form a whole operational semantics. For each of these units we proposed a meta-language and integrated it in the GEMOC studio[5]. Once these 4 units are defined in the GEMOC studio, it is possible to automatically generate a dedicated *modeling workbench* in which models conforming the developed language can be executed with extensive debugging support (*e.g.*, Breakpoint definition on model element, Multi-dimensional and efficient trace management, timeline, step backward, stimuli management). Additionally, due to the reification of the control flow, the model under simulation exhibits its internal concurrency and timing aspect in a platform independent manner and the designer can investigate different inter-leavings of the model evolution to better understand its model. To illustrate the marked graph definition, we augmented the language definition with a graphical syntax defined in Sirius[6] and a graphical animation defined with the Sirius animator[7]. On the screenshot Figure 4), you can see a simple marked graph model under simulation. Note that, in this model, the scheduling state space is infinite due to the possibility to fire the *acquire* transition infinitely many times. However, whens the state space is finite, the GEMOC studio, which is based on TimeSquare [21], offers the possibility to compute the state space of all the acceptable scheduling of the execution functions (typically in order to check temporal properties). Note that a step by step tutorial for the definition of the Marked Graph language is proposed online: http://gemoc.github.io/gemoc-studio/publish/tutorial_markedgraph/html_single/GuideTutorialMarkedGra

Based on these 4 units and the GEMOC studio, we recently won the first price of the model execution context [22].

At this point it is possible to model the behavioral semantics of different languages but it is impossible to execute coordinated heterogeneous models. The next section explains how we took advantage of this behavioral semantics structure to provide a meta-language for the definition of coordination patterns (BCOoL). We also developed an heterogeneous execution engine allowing to tame heterogeneous execution in the GEMOC studio.

[5]a language workbench to federate development on the globalization of modeling languages: http://www.gemoc.org/studio

[6]http://www.eclipse.org/sirius

[7]http://www.eclipse.org/sirius/lab.html

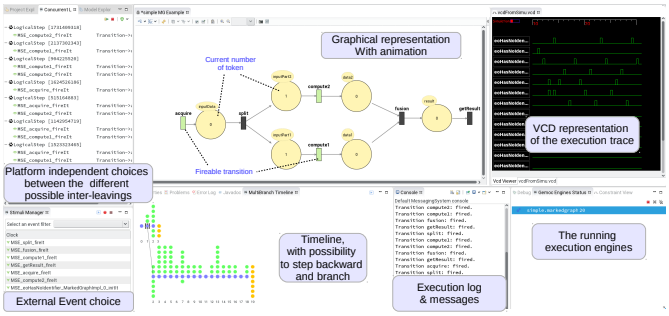

Fig. 4. Screenshot of a marked graph model under step by step simulation in the GEMOC studio

III. Modeling Heterogeneous Model Coordinations

The development of complex systems usually requires skills from different domains (*e.g.*, security, cost, consumption, control). Each domain typically uses its own modeling language (*e.g.*, SysML, AADL, LSC, state machines) and consequently to specify a single system, several models conforming to several DSMLs are developed. A DSML captures the concepts but also the behavioral semantics of a domain. In this context, to provide simulation and verification of the whole system, it is mandatory to coordinate the behavior of these heterogeneous models [23].

For several years, coordination languages proposed dedicated languages to explicitly specify the coordination between (possibly heterogeneous) models [24], [9], [25]. In these approaches, a *system designer* manually develops one or more coordination models, which specify how the different models interact with each others. Conjointly with coordination languages, the software Architecture field used ADLs (Architecture Description Languages) to specify the coordination of (possibly heterogeneous) models [26], [27], [28], [8], [7]. ADLs are usually component based languages in which the models are encapsulated in components with well defined interfaces. The coordination is then specified in some *glue*, defined inside the connectors between these interfaces. Both ADLs and coordination languages usually hide the complexity of model behaviors by treating models as black boxes encapsulated within the boundary of an interface. A model behavioral interface gives a partial representation of the model behavior therefore easing the coordination of behavioral models.

The evolution of both ADLs and coordination languages promoted the use of reusable coordination entities to ease the coordination activity. ADLs proposed connector types that can be instantiated several time in a design [26], [29] and coordination languages proposed reusable subroutines in modules (*e.g.*, *manners* in Manifold [25]). Such reusable entities are of great help for the system designer, avoiding writing several times the same glue/coordination. However, the growing number of involved models, their size and their behavioral heterogeneity make the specification of the coordination a tedious and error prone task. It requires a deep knowledge of all the DSMLs involved in the specification.

In the last years, some approaches like for instance Ptolemy [14] or Modhel'x [30] have proposed to automate the coordination of models by specifying the coordination between DSMLs. More precisely, these approaches have identified specific patterns of coordination between models, and then, they specified them on the languages to be independent of the models. We refer to such approaches as *Coordination Pattern Approaches*. Once a coordination pattern between a set of languages is specified, any models conforming to such languages can be automatically coordinated.

Anyway, the knowledge about system integration is currently either implicitly held by the integrator for ADL and coordination language approaches or encoded within a framework for coordination pattern approaches. To capture explicitly this knowledge and thus leverage integrator know-how, we proposed BCOoL [16], a dedicated language to specify coordination patterns between languages (thus reifying the coordination specification at the language level, see Figure 5). A BCOoL specification captures a coordination pattern that defines, independently of any model, what and how elements from different models are coordinated. Once specified in BCOoL, integration experts can share this specification thus allowing the reuse and tuning of coordination patterns. Also, such a specification is exploited by generative techniques to generate an explicit coordination specification when specific models are used. To be able to specify the coordination between languages, a partial representation of the language behavioral semantics is mandatory. In our approach, the behavioral semantics is abstracted by using a behavioral language interface. The language behavioral interface we used is actually the set of DSE used to define the behavioral semantics of the language.

At the model level, the instances of the DSE are used as handles or control points in two complementary ways: to observe what happens inside the model, and to control what is allowed to happen or not. When required by the coordination, constraints are used to forbid or delay some event occurrences. Forbidding occurrences reduces what can be done by individual models. When several executions are allowed (nondeterminism), it gives some freedom to individual semantics for making their own choices. These events are consequently suitable to drive coordinated simulations without being intrusive in the models. In this context, the DSE are acting as a specification of the "coordination points" that will be available on models (*i.e.,* it is a specification, in intention, of the coordination points that will be available on the models).

Based on the DSE as language behavioral interface, coordination patterns are captured by *Operators*, which are conditional constraints on the DSE. An *Operator* uses, as formal parameters, some DSE from different language interfaces. It contains a *correspondence matching* and a *coordination rule*. The correspondence matching uses the DSE to specify what elements from the model behavioral interfaces (*i.e.,* what instances of DSE) must be coordinated. Then, a coordination rule specifies the, possibly timed, synchronization and causality relationships between the instances of DSE selected during

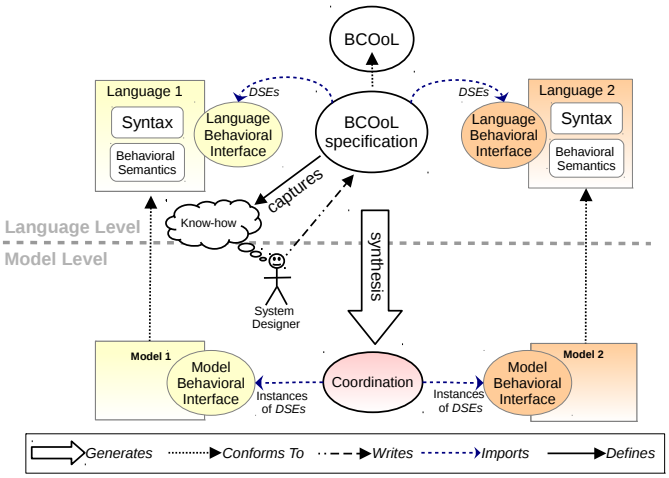

Fig. 5. Coordination pattern at the language level with BCOoL

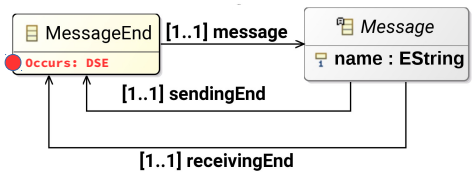

Fig. 6. excerpt of the scenario abstract syntax with one of the DSE.

the matching. To specify the coordination rule, we rely on MoCCML [11]. A *System designer* can then apply the BCOoL specification on some models to automatically generate an explicit coordination model. The generated coordination model is expressed in a CCSL specification, linked to the execution model of each coordinated models. It is then possible to simulate and run analysis on the scheduling space of the coordinated system.

To illustrate BCOoL, we define in the remainder of this section a simple coordination pattern between the Marked Graph language defined in section II and a simple scenario language. In a nutshell, we want to control the input and output transitions of a Marked Graph model by using a scenario. To achieve this goal, we specified a BCOoL operator. The operator specifies that the sending of a message, when its name is the one of a Transition, is synchronous with the firing of the transition.

Before to detail the definition of these operators, we give a bird view of the scenario language, defined according to the methodology defined in section II (see Figure 6). Among many others, there exist two important concepts in the scenario language: *Message* and *MessageEnd*. A message references its sending and receiving message ends and a message end references the associated message. In the concept of MessageEnd, a DSE named *occurs* is defined.

Listing 3. Synchronization between the MarkedGraph and the Scenario languages

```
1 ImportInterface "markedGraph.ecl" as MG
2 ImportInterface "scenario.ecl" as SD
3
4 Operator SynchronousSendFire(dse_send : SD::occurs,
      dse_fire : MG::fireIt)
5   //CorrespondenceMatching
6     when (dse_send.message.name = dse_fire.name)
7           and
8         (dse_send.message.sendingEnd = dse_send);
9   CoordinationRule: RendezVous(dse_send, dse_fire)
10 end operator
```

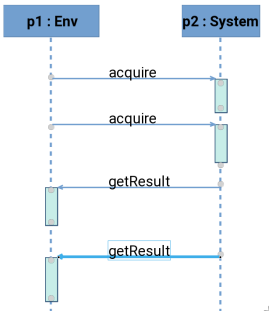

Fig. 7. A simple scenario model

The Listing 3 is a BCOoL specification, which defines the coordination pattern needed to generate the coordination between any marked graph and scenario models. The specification starts by importing the interface of each languages (lines 1 and 2). A first operator named *SynchronousSendFire* is defined from lines 4 to 10. It specifies two formal parameters: the *occurs* DSE, defined in the context of a *MessageEnd* and the *fireIt* DSE defined in the context of a *Transition*. When applied on specific models, all the couple of DSE instances corresponding to these types will be used as actual parameters of the operator. In lines 5 to 8, a matching condition defines the couple of DSE instances that will actually be coordinated. To do so, an OCL condition is defined to specify how two elements are known to match. Here the specification is "when the message referenced from the message end has the same name than the transition"(line 6) and "the message end has the role of a *sendingEnd*" (line 8). When this condition is true, then the DSE instances will be coordinated according to the coordination rule of the operator. In this example, the coordination is a rendez-vous (line 9). Of course, more complex matching conditions or coordination rules can be defined.

To illustrate the coordination on specific models, we applied the coordination pattern on the marked graph model depicted on Figure 4 and the scenario model of Figure 7. In this example, once the coordination generated, the sendings of the *acquire* message are forced to coincide with the firings of the *acquire* transition and the receivings of the *getResult* messages are forced to coincide with the firings of the *getResult* transition. In the behavioral semantics given to the scenario language, the scenario can restart once finished. this sequence can then run indefinitely. In Figure 8, the GEMOC studio is executing the two coordinated models. There are three execution engines: the engine of the marked graph model, the engine of the scenario model and the heterogeneous engine that coordinates the two other engines. On the coordination

points, the possibility of each individual model is then restricted by the heterogeneous engine. The designer still has access to the animation and debugging possibilities for all the models. Additionally, a direct effect of this coordination is to bound the state space of the system because (1) the marked graph model can now fire the transition *acquire* only once and needs to wait for the firing of the transition *getResult* to start again. Figure 8 show the computation of the scheduling state space produced in the GEMOC studio. This state space is a coordinated subset of the state space of the models used in the system. It can be used to search for a specific schedule but also to detect deadlocks or to check temporal properties on the scheduling space.

IV. Conclusion

In this paper, we outlined some of the development realized in the GEMOC project. In a first period, we focused on the modeling of the behavioral semantics of languages by providing the appropriate meta-languages. The modeling of the behavioral semantics emphasis on the possibility to make the control flow explicit and formal. It is then possible to execute/debug/explore the behavior of the model, specially on the concurrent and temporal aspects, in a platform independent manner. In a second period, we used the model of the behavioral semantics as a language behavioral interface. Based on this interface, we developed the BCOoL language to define coordination patterns between heterogeneous languages. Based on the BCOoL specification the coordination of model is automated and an heterogeneous execution engine can be used to coordinate the execution of heterogeneous models.

Currently, we investigate how the explicit modeling of language behavioral semantics and their coordination can be used as a basis to generate appropriate master algorithm on a co-simulation bus. More generally we investigate how co-modeling can be used for co-simulating.

Acknowledgment

The ideas and development depicted in this paper were born from the support of the ANR INS Project GEMOC (ANR-12-INSE-0011), from the numerous discussions with the GEMOC partners[8] and from the discussions with the AOSTE INRIA Sophia Antipolis méditerranée team members[9].

References

[1] B. Combemale, J. Deantoni, B. Baudry, R. France, J.-M. Jézéquel, and J. Gray, "Globalizing Modeling Languages," *Computer*, 2014.

[2] F. Fleurey, B. Baudry, R. France, and S. Ghosh, "A generic approach for automatic model composition," in *AOM Workshop at Models*, 2007.

[3] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Merging models with the epsilon merging language (EML)," in *ACM/IEEE Models/UML*, 2006.

[4] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, 1992.

[5] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Software Engineering*, 1995.

[6] L. Barroca, J. Fiadeiro, M. Jackson, R. Laney, and B. Nuseibeh, "Problem frames: A case for coordination," in *Coordination*, 2004.

[7] D. Garlan and M. Shaw, "An introduction to software architecture," Pittsburgh, PA, USA, Tech. Rep., 1994.

[8] N. Medvidovic and R. N. Taylor, "A framework for classifying and comparing architecture description languages," in *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC '97/FSE-5. New York, NY, USA: Springer-Verlag New York, Inc., 1997, pp. 60–76. [Online]. Available: http://dx.doi.org/10.1145/267895.267903

[9] Esper, "Espertech," 2009.

[10] M. Vara Larsen and A. Goknil, "Railroad Crossing Heterogeneous Model," in *GEMOC workshop*, 2013.

[11] J. Deantoni, P. Issa Diallo, C. Teodorov, J. Champeau, and B. Combemale, "Towards a Meta-Language for the Concurrency Concern in DSLs," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Grenoble, France, Mar. 2015. [Online]. Available: https://hal.inria.fr/hal-01087442

[12] B. Combemale, J. Deantoni, M. Vara Larsen, F. Mallet, O. Barais, B. Baudry, and R. France, "Reifying Concurrency for Executable Metamodeling," in *SLE*, 2013.

[13] F. Latombe, X. Crégut, B. Combemale, J. Deantoni, and M. Pantel, "Weaving Concurrency in eXecutable Domain-Specific Modeling Languages," in *8th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. Pittsburg, United States: ACM, 2015. [Online]. Available: https://hal.inria.fr/hal-01185911

[14] B. Evans, A. Kamas, and E. A. Lee, "Design and simulation of heterogeneous systems using ptolemy," in *Proceedings of ARPA RASSP Conference*, 1994.

[15] F. Boulanger and C. Hardebolle, "Simulation of Multi-Formalism Models with ModHel'X," in *ICST*, 2008.

[16] M. E. Vara Larsen, J. Deantoni, B. Combemale, and F. Mallet, "A Behavioral Coordination Operator Language (BCOoL)," in *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, T. Lethbridge, J. Cabot, and A. Egyed, Eds., no. 18. Ottawa, Canada: ACM, Sep. 2015, p. 462, to be published in the proceedings of the Models 2015 conference. [Online]. Available: https://hal.inria.fr/hal-01182773

[17] *Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0*, Object Management Group, Inc., 2011.

[18] E. A. Lee and A. Sangiovanni-Vincentelli, "Comparing models of computation," in *IEEE/ACM TCAD*, 1997.

[19] A. Jantsch, *Modeling Embedded Systems and SoCs*. Morgan Kaufmann Publishers Inc., 2004.

[20] C. André, "Syntax and Semantics of the Clock Constraint Specification Language (CCSL)," Tech. Rep., 2009.

[21] J. DeAntoni and F. Mallet, "Timesquare: treat your models with logical time," in *Objects, Models, Components, Patterns*, 2012.

[22] B. Combemale, J. Deantoni, O. Barais, A. Blouin, E. Bousse, C. Brun, T. Degueule, and D. Vojtisek, "A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio," in *8th Transformation Tool Contest*. l'Aquila, Italy: CEUR, 2015. [Online]. Available: https://hal.inria.fr/hal-01152342

[23] G. A. Papadopoulos and F. Arbab, "Coordination models and languages," CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 1998.

[24] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 97–107, Feb. 1992. [Online]. Available: http://doi.acm.org/10.1145/129630.129635

[25] F. Arbab, I. Herman, and P. Spilling, "An overview of manifold and its implementation," *Concurrency: Pract. Exper.*, vol. 5, no. 1, pp. 23–70, Feb. 1993. [Online]. Available: http://dx.doi.org/10.1002/cpe.4330050103

[26] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *Software Engineering, IEEE Transactions on*, vol. 21, no. 4, pp. 314–335, Apr 1995.

[27] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 3, pp. 213–249, Jul. 1997. [Online]. Available: http://doi.acm.org/10.1145/258077.258078

[28] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using

[8] http://gemoc.org/ins-partners/

[9] https://team.inria.fr/aoste/team-members/

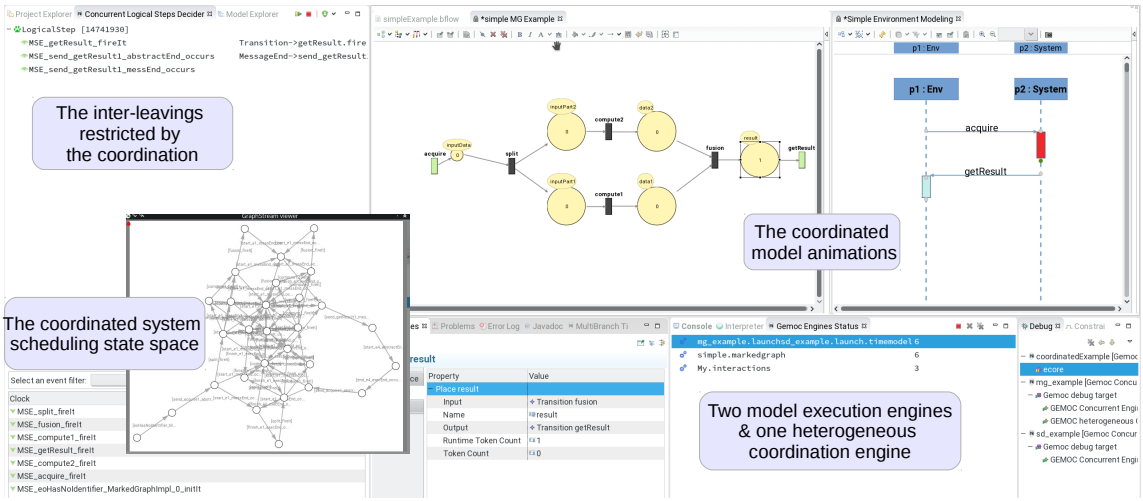

Fig. 8. Debugging of two coordinated models with their scheduling state space

rapide," *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, pp. 336–355, Apr. 1995. [Online]. Available: http://dx.doi.org/10.1109/32.385971

[29] F. Arbab and F. Arbab, "A channel-based coordination model for component composition," in *Mathematical Structures in Computer Science*. University Press, 2002, pp. 329–366.

[30] F. Boulanger and C. Hardebolle, "Simulation of Multi-Formalism Models with ModHel'X," in *Proceedings of ICST'08*. IEEE Comp. Soc., 2008, pp. 318–327.