A Behavioral Coordination Operator Language (BCOoL)

Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, Frédéric Mallet

A Behavioral Coordination Operator Language (BCOoL)

Matias Ezequiel Vara Larsen*, Julien DeAntoni*, Benoit Combemale† and Frédéric Mallet*
*Université Nice-Sophia Antipolis, I3S, INRIA
†INRIA and University of Rennes 1

***Abstract*—The design of complex systems involves various, possibly heterogeneous, structural and behavioral models. In model-driven engineering, the coordination of behavioral models to produce a single integrated model is necessary to provide support for validation and verification. Indeed, it allows system designers to understand and validate the global and emerging behavior of the system. However, the manual coordination of models is tedious and error-prone, and current approaches to automate the coordination are bound to a fixed set of coordination patterns. In this paper, we propose a Behavioral Coordination Operator Language (B-COoL) to reify coordination patterns between specific domains by using coordination operators between the Domain-Specific Modeling Languages used in these domains. Those operators are then used to automate the coordination of models conforming to these languages. We illustrate the use of B-COoL with the definition of coordination operators between timed finite state machines and activity diagrams.**

***Index Terms*—Heterogeneous Modeling, Coordination Languages, DSMLs**

I. Introduction

The development of complex software intensive systems involves interactions between different subsystems. For instance, embedded and cyber-physical systems require the interaction of multiple computing resources (general-purpose processors, DSP, GPU), and various digital or analog devices (sensors, actuators) connected through a wide range of heterogeneous communication resources (buses, networks, meshes). 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*.

To understand the system and its emerging behavior globally, it is necessary to specify how models and languages are related to each other, 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]), in this work, we rather focus on the coordination [4] of behavioral languages to provide simulation and/or verification capabilities for the whole system specification. In current coordination approaches [4]–[7], the coordination is manually defined between particular models. This is usually done by integrator experts that apply some coordination patterns according to their own skills and know-how.

In this paper, we propose to leverage the integrator expert's skills into a dedicated language named B-COoL that allows for capturing coordination patterns for a given set of DSMLs. These patterns are captured at the language level, and then used to derive a coordination specification automatically for models conforming to the targeted DSMLs. The coordination at the language level relies on a so-called *language behavioral interface*. This interface exposes an abstraction of the language behavioral semantics in terms of Events. B-COoL helps understand and reason about the relationships between different languages used in the design of a system.

The paper is organized as follows. Section II presents the main issues in the coordination of behavioral models, and shows how they can be tackled by explicitly capturing coordination patterns at the language level. Section III defines the notion of language behavioral interface by using an example language named Timed Finite State Machine (TFSM). This language is used later in Section IV to illustrate B-COoL. In Section V, we validate the approach by using B-COoL to capture three coordination patterns between two languages: TFSM and fUML Activities. Section VI gives an overview and comparison to related work. Section VII concludes with a brief summary and a discussion of ongoing and future actions.

II. Coordination Patterns

Large and complex systems are often made of smaller "coordinated" behavioral models. There are several definitions of the notion of coordination in the literature [8]. Carriero et al. [4] define coordination as the process of building programs by gluing together active pieces. Differently, Eker et al. [9] use the word *composition* to refer to the interactions and communications between models while preserving the properties of each individual model. By relying on these definitions, in this paper, we adopt the wording of *coordination specification* as being the explicit modeling of the interactions amongst behavioral models to obtain the emerging system behavior. In this sense, the coordination specification must be executable to enable the evaluation of the emerging behavior of the whole system.

The coordination between models can be explicitly modeled by using a *coordination language* (*e.g.*, Linda [4], Esper [6]). An integrator can define one or more coordination specifications to specify how models interact. This results in a global behavior that is explicit and amenable for reasoning (for instance for Verification and Validation activities). However, if a

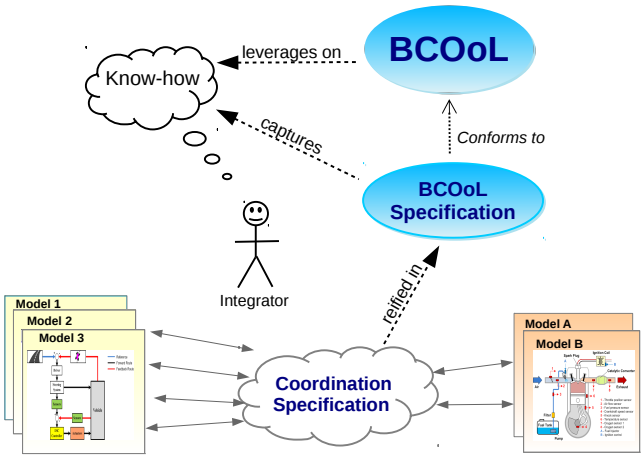

Fig. 1. Coordination at the language level with BCOoL

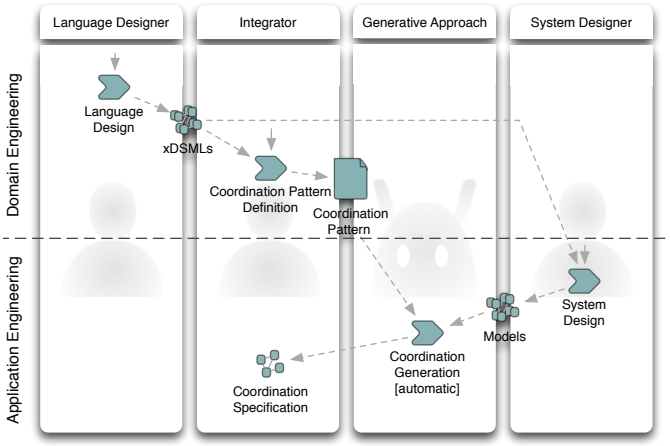

Fig. 2. The proposed workflow

similar problem occurs on different models, the integrator has to devise another coordination specification. When applied to models, a coordination specification only captures the solution for one single problem and does not propose a commonly applicable solution for other coordination problems.

To capture once and for all the know-how of the integrator, some approaches capture coordination patterns by specifying the coordination at the language level. This is the case for coordination frameworks like Ptolemy and ModHel'X, which rely on hierarchical coordination patterns between heterogeneous languages. This is also the case for solutions like MASCOT [10], which provide ad-hoc coordination patterns (*i.e.,* between Matlab and SDL). Once the coordination pattern between a set of languages is captured, the models conforming to such languages can be coordinated automatically. Such approaches successfully capture the know-how of the integrator, however, they do so by embedding the coordination pattern inside a tool. As a result, the integrator cannot change the coordination specification without altering the core of the tool. However, for complex systems, the integrator may need to capture several coordination patterns and potentially combine them. This is highlighted in [11] where authors use Ptolemy to capture three hierarchical coordination patterns between a finite state machine with Data flows, a Discrete event model and a Synchronous/Reactive model. Since each coordination pattern is applicable in different situations, authors argue that each one is useful. Thus, the integrator must be able to capture different coordination patterns since there is not necessarily a single *valid* one. However, current coordination frameworks can only support such a variation by modifying the framework itself. Additionally, the coordination model is mixed with the functional model, which makes it very tricky to modify one without risking altering the other.

During the integration activity, the integrator must be able to capture different coordination patterns and their semantics must be explicit rather than being hidden inside a tool. To illustrate this, Gabor et al. [12] show the semantic variations of Statecharts presented in different tools. Since each approach interprets differently the semantics of Statecharts, we obtain different behaviors depending on the approach. However, if the semantics is hidden inside the tool and is not made explicit, this can lead to misunderstandings and errors. Moreover, validation and verification activities are limited in current coordination frameworks since the coordination is encoded by using a general purpose language.

The knowledge about system integration is currently either implicitly held by the integrator or encoded within a framework. To capture explicitly this knowledge and thus leverage integrator know-how, we propose B-COoL, a dedicated language to capture coordination patterns between languages, thus reifying the coordination specification at the language level (see Figure 1).

A B-COoL specification captures a coordination pattern (see Figure 2) that defines what and how elements from different models are coordinated. Once specified in B-COoL, integration experts can share this specification thus allowing the reuse and tuning of coordination patterns. Also, such a specification can be 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 semantics is abstracted by using a behavioral language interface. This notion of behavioral language interface is further discussed in the next section and is illustrated with a language named TFSM (Timed Finite State Machine). This language is also used in Section IV to introduce B-COoL.

III. Language Behavioral Interface

Some coordination languages deal with 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. However, it is not uniquely defined and may vary depending on approaches. For instance, in *Opus* [13], the interface is a list of methods provided by the model. Other approaches

abstract away the non-relevant parts of the behavior of models as events [14] (also named signals in [15]). These approaches focus on events and how they are related to each other through causal, timed or synchronization relationships. Following the same idea, *control-driven* coordination languages rely on a model behavioral interface made of explicit events [5], [6], [16]. While in Rapide [5], the interface is only a set of events acceptable by the model, some other approaches go further and also exhibit a part of the internal concurrency. This is the case of [16] where authors propose an interface that contains services and events, but also properties that express requirements on the behavior of the components. Such requirements act as a contract and can be checked during the coordination to ensure a correct behavior. In these approaches, the model behavioral interface provides information to coordinate the behavior of a model. In particular, in event-driven coordination approaches events act as "coordination points" and exhibit what can be coordinated. This gives a support for control and timed coordination while remaining independent of the internal model implementation. Moreover, event-driven coordinations are non intrusive; *i.e.,* models can be coordinated without any change to their implementation, thus ensuring a complete separation between the coordination and the computational concerns. Several causal representations from the concurrency theory are used to capture event-based behavioral interface. A causal representation captures the concurrency, dependency and conflict relationships among actions in a particular program. For instance, an event structure [14] is a partial order of events, which specifies the, possibly timed, causality relations as well as conflict relations (*i.e.,* exclusion relations) between actions of a concurrent system. This fundamental model is powerful because it totally abstracts data and program structure to focus on the partial ordering of actions. It specifies, *in extension* and *in order*, the set of actions that can be observed during the program execution. An event structure can also be specified *in intention* to represent the set of observable event structures during an execution (see *e.g.*, [17] or [18]).

In our approach, to capture the specification of coordination patterns between languages, we require a behavioral interface, but at the language level. A language behavioral interface must abstract the behavioral semantics of a language, thus providing only the information required to coordinate it, *i.e.,* a partial representation of concurrency and time-related aspects. Furthermore, to avoid altering the coordinated language semantics, the specification of coordination patterns between languages should be non intrusive, *i.e.,* it should keep separated the coordination and the computation concerns. In [19] elements of event structures are reified at the language level to propose a behavioral interface based on sets of *event types* and *contraints*. Event types (named DSE for Domain Specific Event) are defined in the context of a metaclass of the abstract syntax (AS), and abstract the relevant semantic actions. Jointly with the DSE, related constraints give a symbolic (intentional) representation of an event structure. With such an interface, the concurrency and time-related aspects of the language behavioral semantics are explicitly exposed and the coordination is event-driven and non intrusive.

Then, for each model conforming to the language, the model behavioral interface is a specification, in intention, of an event structure whose events (named MSE for Model Specific Event) are instances of the DSE defined in the language interface. While DSE are attached to a metaclass, MSE are linked to one of its instances. The causality and conflict relations of the event structure are a model-specific unfolding of the constraints specified in the language behavioral interface. Just like event structures were initially introduced to unfold the execution of Petri nets, we use them here to unfold the execution of models.

We propose to use DSE as "coordination points" to drive the execution of languages. These events 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. All this put together makes the DSE suitable to drive coordinated simulations without being intrusive in the models. Coordination patterns are captured as constraints at the language level on the DSE.

To illustrate the approach, we introduce a simple state-based language named Timed Finite State Machine (TFSM) and its behavioral interface; a state machine language augmented with timed transitions (see Figure 3). The metamodel describes the abstract syntax of the TFSM language (see Figure 3). A *System* is composed of *TFSMs*, global *FSMEvents* and global *FSMClocks*. Each *TFSM* is composed of *States*. Each state can be the source of outgoing guarded *Transitions*. A guard can be specified either by the reception of an *FSMEvent* (*EventGuard*) or by a duration relative to the entry time in the source state of the transition (*TemporalGuard*). When fired, transitions generate a set of simultaneous *FSMEvent* occurrences.

The TFSM language defines the following DSE: *entering* and *leaving* a state, *firing* a transition, the occurrences (*occurs*) of a FSMEvent and the *ticks* of a FSMClock (see at the top of Figure 3). These DSE are part of the language behavioral interface of TFSM. DSE are defined by using a specific language named ECL (standing for Event Constraint Language [20]) which is an extension of OCL [21] with events. ECL takes benefits from the OCL query language and its possibility to augment an abstract syntax with additional attributes (without any side effects). Consequently by using ECL, it is possible to augment AS metaclasses and add DSE. A partial ECL specification of TFSM is shown in Listing 1 where the DSE *entering* and *leaving* are defined in the context of State (Listing 1: line 6) while *occurs* is defined in the context of FSMEvent (Listing 1: line 4). When a metaclass is instantiated, the corresponding DSE are instantiated; *e.g.*, for each instance of the metaclass *State*, DSE *entering* is instantiated. Each instance of DSE is a MSE. In the case of TFSM, since two States are instantiated (*S1* and *S2*), there are two MSE of *entering*: *S1_entering* and *S2_entering*. All MSE are part of

the model behavioral interface.

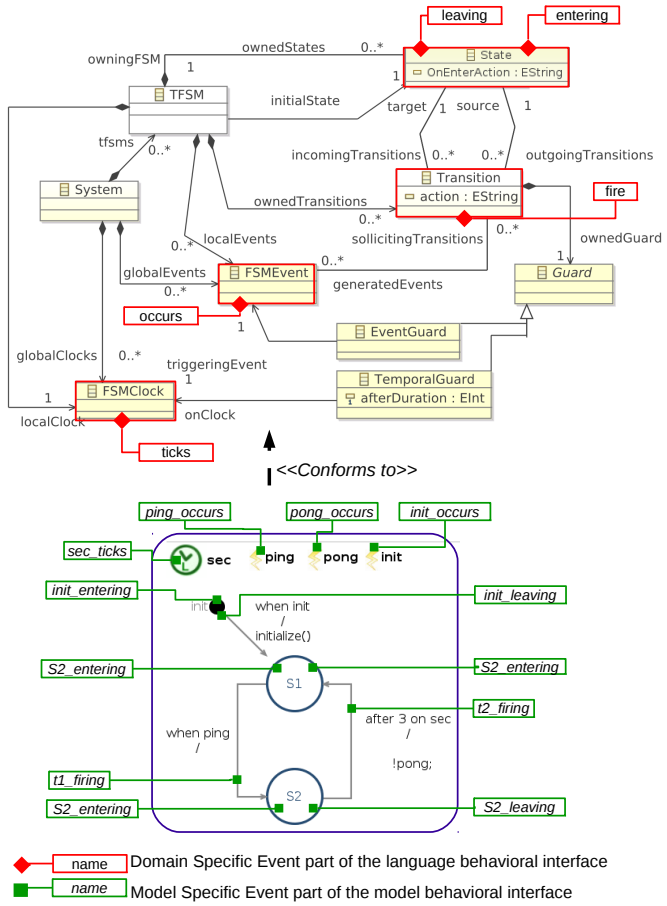

Fig. 3. (At the top) The TFSM metamodel with its language behavioral interface. (At the bottom) a TFSM model with its model behavioral interface

Listing 1. Partial ECL specification of TFSM

```
1 package tfsm
2 context FSMClock
3   def: ticks : Event = self
4 context FSMEvent
5   def: occurs : Event = self
6 context State
7   def : entering : Event = self
8   def : leaving : Event = self
```

In the following section, the TFSM language together with its behavioral interface is used to introduce our language B-COOL

IV. B-COOL

A. Overview

B-COOL is a dedicated (meta)language to explicitly capture the knowledge about system integration. With B-COOL, an integrator can explicitly capture coordination patterns at the language level. Specific *operators* are provided to build the coordination patterns and specify how the DSE of different language behavioral interfaces are combined and interact. From the B-COOL specification, we generate an executable and formal coordination model by instantiating all the constraints on each and every instance of DSE. Therefore, the generated coordination model implements the coordination patterns defined at the language level.

The design of B-COOL is inspired by current structural composition languages (*e.g.*, [2], [3]). These approaches rely on the *matching* and *merging* phases of syntactic model elements. A matching rule specifies what elements from different models are selected. A merging rule specifies how the selected model elements are composed. In these approaches the specification is at the language level, but the application is between models. Similarly, a B-COOL operator relies on a *correspondence matching* and a *coordination rule*. The correspondence matching identifies what elements from the behavioral interfaces (*i.e.,* what instances of DSE) must be selected. The merging phase is replaced by a coordination rule. While in the structural case the merging operates on the syntax, the coordination rule operates on elements of the semantics (*i.e.,* instances of DSE). Thus, coordination rules specify the, possibly timed, synchronizations and causality relationships between the instances of DSE selected during the matching.

We illustrate the use of B-COOL through a (simple) running example: *i.e.,* building the synchronized product of TFSM. This is a very classical "coordination" operation on automata with frequent references in the literature [22]. The goal here is to show that we can build this operator and use it off-the-shelf when needed. It is informally defined as follows: When coordinating two state machines, all events belonging to both state machines must be synchronized using a "rendez-vous". All the other events, belonging to only one state machine, can occur freely at any time. The synchronized product is defined for any state machine, at the language (metamodel) level. When applied it concerns two specific state machines, at the instance (model) level. Note that this first behavioral coordination pattern is homogeneous (*i.e.,* it involves a single language). Examples of heterogeneous coordination patterns (*i.e.,* that involve several languages) are provided in Section V.

In the following, we first present the abstract syntax, and then, the execution semantics of B-COOL. We finish this section by showing the language workbench of B-COOL which is implemented as part of the Gemoc studio. We use the studio on the running example, and we generate the coordination model between two particular TFSMs. We then show how the generated coordination model can be executed.

B. Abstract Syntax of B-COOL

The main element of B-COOL (see Figure 4) is a *BCoolSpecification* that contains language behavioral interfaces (*importsInterfaceStatements*) and *Operators*. The specification must import at least two language behavioral interfaces. Interfaces provide the *DSE* needed for the coordination. The imported DSE serve as parameters for the operators. Then, an operator specifies what instances of these DSE are selected and how they are coordinated (the *DSEs* reference). For instance, to build the synchronized product of TFSM, we need to synchronize FSMEvents. This is done by coordinating the instances of DSE *occurs* (see Figure 3). First, the language

behavioral interface of TFSM is imported. Then, an operator is defined with *occurs* as a parameter.

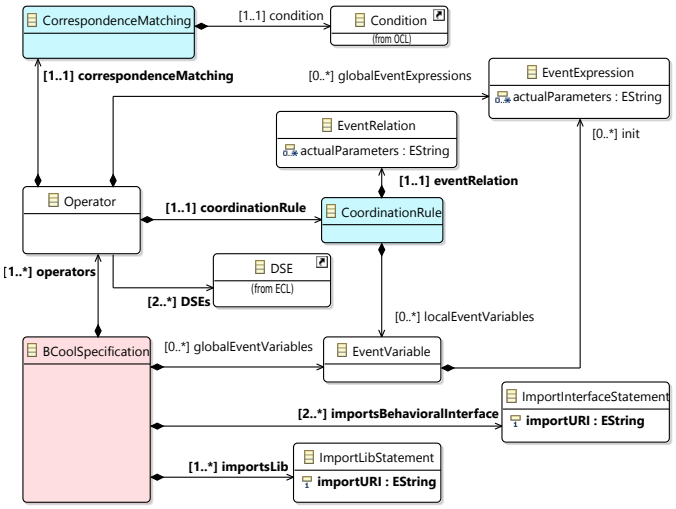

Fig. 4. Simplified view of B-COoL abstract syntax

Each operator contains both a *correspondenceMatching* and a *coordinationRule*. The former relies on a Boolean *Condition* defined as an OCL expression. It acts as a precondition for the coordination rule, *i.e.,* it is a predicate that defines when the coordination rule must be applied to the given parameters. To specify the predicate, it is possible to navigate through the context of the DSE and query a specific element used within the Boolean expression. For instance, for the synchronized product, the condition selects an instance of DSE *occurs* by looking at its attribute *name*.

The *coordinationRule* specifies how the selected instances of DSE must be coordinated. To do so, the user must define some *EventVariables* (*localEventVariables*) and an *EventRelation*.

An event variable can be either defined locally within the operator or globally for the whole specification (*globalEventVariables*). These variables either define global events used across different operators, or create a new event from the selected instances of DSE and possibly from attributes of the input models. The definition of these events is made by using an *EventExpression*. An event expression returns a new event from a given parameter. For instance, this can be used to select only some occurrences of a DSE instance, thus allowing the implementation of filters. An event expression can also be used to join in a single event the occurrences of different events (union). When used in the coordination rule, the resulting events can be used as parameters of event relations, constraining by transitivity (some of) the occurrences of DSE instances.

How the selected events are coordinated is determined by event relations that restrict the occurrences of the events on which it is applied. The actual parameters of the event relation can be some instances of DSE and/or some *EventVariables*. For instance, the synchronized product specifies a strong synchronization. Thus, the coordination rule uses a "rendezvous" relation between the selected instances of DSE *occurs*. As a result, all the occurrences of these events are force to happen simultaneously.

In B-COoL, the definition of event expressions and relations is made in dedicated libraries, which must be imported. This is further explained in the following subsection.

C. B-COoL library

Libraries gather some predefined event expressions and relations, which must be imported by the specification (*ImportedLibStatement* in Figure 4). Libraries can be organized by modeling domains to gather all the relevant operators.

A library is a set of declarations together with their formal parameters. A library also contains some definitions, which give the actual behavior of the declarations. Declarations are referenced in a B-COoL specification. Generally speaking, event expressions create a new event from their parameters (*e.g.*, building the *Union*, or the *Intersection* of its parameters). They can be used to filter some occurrences of existing events. Such constraints are used in B-COoL either to provide global events used in different operators or to define some filters used in the coordination rules. Relations, however, constrain the evolution of the events given as formal parameters. For instance, a relation can define a *Rendezvous* synchronization on its parameters. Lots of other relations, more or less complex can be defined (*e.g.*, *Causality*, *FIFO* or ad-hoc relations for specific protocols).

Currently, B-COoL includes a library, named *facilities.bcoollib*, that provides all the declarations used in all the following examples. The integrator however can extend the current library by defining new specific constraints depending on its problems and domain. The definition part of B-COoL is common to the one of CCSL [17], a formal language dedicated to event constraints. As a result, when building B-COoL operators a CCSL specification is produced [7]. We can then use CCSL tool (TimeSquare [23]) to analyze and execute the generated coordination specification. This is further discussed in Section IV-E. We could also use another language to build the semantics of operators and then take benefit from other analysis tools.

D. Execution semantics

In this section we give a rough description of the execution semantics of B-COoL. *i.e.,* how a B-COoL specification is used to obtain a coordination model. The detailed semantics is available in [24].

Let Ev be the (finite) set of event type names (representing the DSE). Considering a language L, A behavioral interface i_L is a subset of event type names, $i_L \subset Ev$. A B-COoL specification imports N disjoint language interfaces and a set of operators $\mathcal{O}p$, with $N \geq 2$. Each operator from $\mathcal{O}p$ has a set of formal parameters $\mathcal{P}$, where each parameter is defined by a name and its type (*i.e.,* an event type). Each operator also has a correspondence matching condition (denoted CMC) and a correspondence rule (denoted CR). A B-COoL specification is applied to a set of input models denoted $\mathcal{M}_\mathcal{I}$, with $|\mathcal{M}_\mathcal{I}| =$

N. From an operational point of view, the first step consists in producing the model behavioral interface of each input model (see example in Figure 3). It results in a set of model interfaces denoted $\mathcal{I}_{\mathcal{M}_\mathcal{I}}$, of size N. An interface is a set of events, each of which is typed by an event type. Each operator op in $\mathcal{O}p$ is processed individually and several times with different actual parameters, which depend on the model interfaces in $\mathcal{I}_{\mathcal{M}_\mathcal{I}}$. The set of actual parameters to be used is obtained by a *restricted* Cartesian product of all the model interfaces in $\mathcal{I}_{\mathcal{M}_\mathcal{I}}$. The restriction consists in two steps: First, a new set of model interface (denoted $\mathcal{I}'_{\mathcal{M}_\mathcal{I}}$) is created. For each parameter p in $\mathcal{P}$, a new model interface $\mathcal{I}^p_{\mathcal{M}_\mathcal{I}}$ is created and all the events in $\mathcal{I}_{\mathcal{M}_\mathcal{I}}$ that have the same type than p are collected in $\mathcal{I}^p_{\mathcal{M}_\mathcal{I}}$. Then, $\mathcal{I}^p_{\mathcal{M}_\mathcal{I}}$ is added to $\mathcal{I}'_{\mathcal{M}_\mathcal{I}}$. Second, a classical Cartesian product is applied on $\mathcal{I}'_{\mathcal{M}_\mathcal{I}}$. It results in a set containing the list of actual parameters to be used with the operator, *i.e.,* each set in the result of the Cartesian product represents the actual parameters of the operator. For each set $actualParams$ in the result of the Cartesian product, if $actualParams$ satisfies the correspondence matching condition (CMC), then the coordination rule (CR) is instantiated with the values in $actualParams$. The instantiation is made in two steps. First, the local events, if any, are created in the targeted coordination language according to the expression used to initialize it. The expression can use any event in $actualParams$ and possibly some constants (*e.g.*, some Integer constants). The local events are added to $actualParams$ so that they can be used in the next. The second step is the application of the relation. It results in the creation of the corresponding relation in the targeted coordination language. The actual parameters of the coordination rule are then the ones from $actualParams$ or some constants, like for the expressions.

E. Language Workbench

B-COoL is a set of plugins for Eclipse[1] as part of the GEMOC studio[2]; which integrates technologies based on Eclipse Modeling Framework (EMF) adequate for the specification of executable domain specific modeling languages. B-COoL is itself based on the EMF and its abstract syntax has been developed using Ecore (*i.e.,* the meta-language associated with EMF). The textual concrete syntax of B-COoL has been developed by using Xtextthus providing advanced editing facilities. To introduce the concrete syntax, we describe the running example in B-COoL (see Listing 2).

The specification begins by importing twice the ECL specification of the TFSM language (Listing 2: lines 3 and 4). In B-COoL, the number of language behavioral interfaces must correspond to the number of accepted models. Since the operator is specified between two models both conforming to the TFSM language, the ECL specification is imported twice and named *tfsmA* and *tfsmB*. Then, we define a new operator named *SyncProduct* (Listing 2: line 6) to select and coordinate instances of DSE *occurs*. Pairs of instances of these events are selected by comparing the attribute name defined in the context of FSMEvent. In Listing 2: line 7, the instances of DSE mapped as *dse1* and *dse2* are queried to get attribute name. Then, the attributes are used as operands for a boolean condition. When the two instances of DSE occurs have the same name, the pairs are selected and the coordination rule is applied. The selected instances are synchronized with a *Rendezvous* relation (Listing 2: line 8). This results in forcing a simultaneous occurrence of the two events.

We use the specification presented in Listing 2 to generate the coordination specification between two particular TFSM models: *TFSMA* and *TFSMB* (Figure 5). The workbench is then used to execute this coordination specification. Figure 5 illustrates the partial timing output of the execution of the whole example. The workbench also offers the possibility to obtain by exploration quantitative results on the scheduling state-space. A video presenting the whole flow (compilation, execution, diagram animation, state-space exploration) can be found on the companion the webpage[3].

To validate our approach, we present in the following section the development of some B-COoL coordination operators. We then use these operators to generate the coordination model for a video surveillance system. The workbench is finally used to execute and validate the result.

Listing 2. B-COoL specification of synchronized product between TFSM models

```
1 BCOoLSpec ProductTfsmAndTfsm
2 ImportLib "facilities.bcoollib"
3 ImportInterface "TFSM.ecl" as tfsmA
4 ImportInterface "TFSM.ecl" as tfsmB
5
6 Operator SyncProduct(dse1:tfsmA::occurs,dse2:tfsmB::occurs)
7     CorrespondenceMatching: when(dse1.name = dse2.name)
8     CoordinationRule: RendezVous(dse1, dse2)
9 end operator
```

V. Validation of the Approach

In this section, we validate the use of B-COoL through the specification of four coordination operators. Each operator captures a given coordination pattern between two languages: TFSM (the language presented in Section III) and the fUML language [25]. The operators are used to coordinate the model of a video surveillance system. By using this example, we illustrate the benefits of our approach for the definition of coordination operators (*i.e.,* understandability and adaptability), the generation of the coordination specification (*i.e.,* usability) and the studies of the coordinated system (*i.e.,* analysis capabilities). We finish this section by using these criteria to compare our approach with coordination languages (like the one presented in Section II) and with ad-hoc approaches like the one of [26] (best paper of MODELS 2014).

Before going into the example, we briefly present the language behavioral interface of the fUML language partially shown in Listing 3. For each *Activity* two DSE are defined: *startActivity* and *finishActivity*, to identify respectively the starting and finishing instants of the activity. Similarly, Two DSE are defined for each *Action*: *startAction* and *finishAction*,

[1]http://www.eclipse.org

[2]http://gemoc.org/studio/

[3]http://timesquare.inria.fr/BCOoL

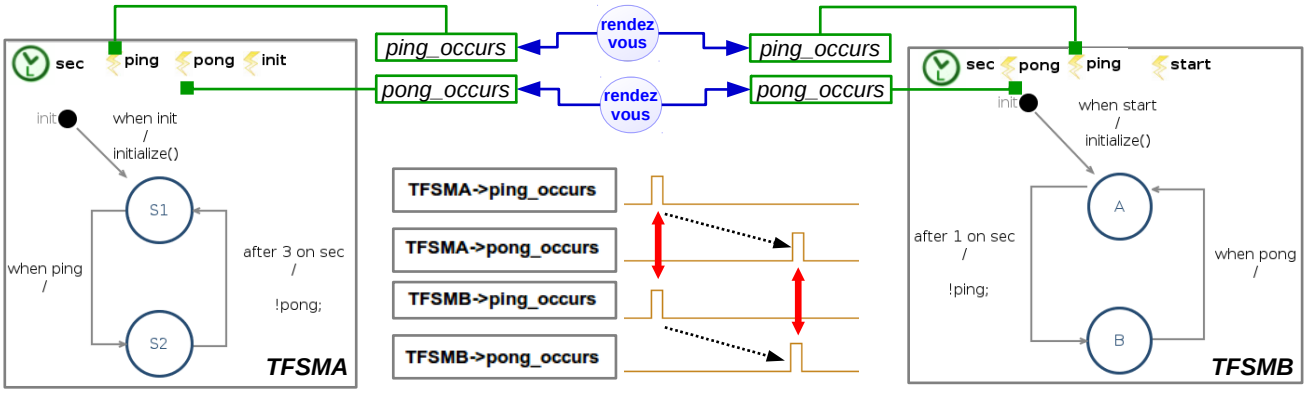

Fig. 5. Applying the synchronized product operator to two TFSMs and execution of the resulting coordination specification

to identify the starting and the finishing of an Action. These DSE, part of the fUML language behavioral interface, are used throughout this section to specify the coordination operators.

Listing 3. Partial ECL specification of Activity Diagram

```
1 package uml
2 context Activity
3   def: startActivity : Event = self
4   def: finishActivity: Event = self
5 context Action
6   def : startAction : Event = self
7   def : finishAction : Event = self
```

A. Definition of Coordination Operators between the TFSM and fUML language

In a B-COOL specification (see Listing 4: line 1), we define four coordination operators between the TFSM and fUML languages. The specification begins by importing (see Listing 4: line 3 and 4) the behavioral interfaces of each language, defined as an ECL specification.

The first operator, named *SyncProduct* (see Listing 4: line 6), differs from the running example because it applies to heterogeneous languages (namely fUML and TFSM). Whereas in the running example the occurrences of FSMEvents were synchronized with the occurrences of other FSMEvents, here they are synchronized with the start of fUML *Actions*, *i.e.,* by coordinating instances of DSE *occurs* and *startAction*. This operator only requires a slight modification of the specification presented in the running example (see Listing 2). The only modifications are the imported interfaces and the type of the DSE used in the operator (see Listing 4: line 6). The rest of the definition (*i.e.,* the correspondence matching and coordination rule) is unchanged (see Listing 4: line 7 and 8). This is the simple adaptation to go from the homogeneous case to the heterogeneous case. We want to highlight that the adaptation has been done only by identifying the new DSE to be constrained. This should also be the case for other languages.

For the second and third operators, we specify a hierarchical coordination pattern between the TFSM and fUML languages, unlike hierarchical coordination frameworks where the semantics is hidden, these operators explicitly specify how the hierarchical coordination is implemented. In our case, we chose the semantics in which entering a specific state of a TFSM model triggers the execution of a given fUML activity. When leaving a state, several semantic variation points may be chosen. The outgoing transitions from a state can be considered, for instance, as preemptive for the activity model (*i.e.,* firing a transition from a state to another pre-empts the internal activity). Alternatively, the transition can be considered as non-preemptive (*i.e.,* the states cannot be left before the associated activity finishes). In this paper, we chose non-preemptive transitions, preemptive ones are detailed on the companion webpage. Listing 5 presents the B-COOL specification that is organized around two operators: *StateEntering* and *StateLeaving*.

Listing 4. Heterogeneous synchronized product operator between the TFSM and fUML languages

```
1 BCOoLSpec TFSM-fUMLOperators
2 ImportLib "facilities.bcoollib"
3 ImportInterface "activitySemantics.ecl" as activity
4 ImportInterface "TFSM.ecl" as tfsm
5
6 Operator SyncProduct(dse1 : activity::startAction, dse2 :
      tfsm::occurs)
7   CorrespondenceMatching: when(dse1.name = dse2.name)
8   CoordinationRule: RendezVous(dse1, dse2)
9 end operator
```

The *StateEntering* operator coordinates the action of entering into a state with the start of an activity so that when a state is entered, the execution of the activity is started synchronously. Entering into a state is identified by the *entering* DSE defined in the context of State (see Figure 3). Instances of such DSE have to be coordinated with instances of the *startActivity* DSE. To identify pairs of such events, the correspondence matching selects events by comparing the *onEnterAction* defined in the states and the name of the activities (Listing 5: line 12). OnEnterAction is a method defined in the context of State (see Figure 3) that specifies the method invoked when a state is entered. In our case, we use this attribute to specify the name of the activity that the state represents. Then, since we have selected this semantics in which entering a state synchronously triggers the starting of an activity, the coordination rule specifies a rendez-vous relation between the selected pairs of DSE *entering* and *startActivity* (Listing 5: line 13).

The *StateLeaving* operator (Listing 5: line 16) coordinates the finishing of the activity with the leaving of the state. Leaving a state is identified by DSE *leaving* and finishing an activity is identified by DSE *finishActivity*. In this case, the

coordination rule has to express that leaving the state follows the termination of the activity. To do that, we use a causal event relation (Listing 5: line 19).

Listing 5. Hierarchical coordination operators between TFSM and fUML languages

```
10 Operator StateEntering(dse1 : activity::startActivity, dse2
       : tfsm::entering)
11   CorrespondenceMatching:
12    when(dse1.name = dse2.onEnterAction.name)
13   CoordinationRule: RendezVous(dse1, dse2)
14 end operator
15
16 Operator StateLeaving(dse1 : activity::finishActivity, dse2
       : tfsm::leaving)
17   CorrespondenceMatching:
18    when(dse1.name = dse2.onEnterAction.name)
19   CoordinationRule: Causality(dse1, dse2)
20 end operator
```

For the fourth operator, we deal with the temporal aspects of the model coordination. The operator specifies how the time in the TFSM elapses during the execution of the activities that specify the on-entry action of a state. This coordination is also hierarchical, but in this case, only considers the timing aspects. In the TFSM language, each state machine has a *localClock* used to measure the time (see Figure 3) while the fUML language is untimed. The local clock is a *FSMClock*, which defines a DSE named *ticks* whose occurrences represent a physical time increment. In the fUML language, the duration of activities can be represented as the time between the DSE *startActivity* and DSE *finishActivity* (Listing 3). To coordinate the time, it is necessary to specify the number of *ticks* of the local clock between the occurrence of the DSE *startActivity* and *finishActivity*. We propose an operator that enforces the execution of the "internal" activity to be atomic with respect to the time in the TFSM model. As a result, there is no occurrence of the DSE ticks of the corresponding local clock during the execution of the activity.

Listing 6 captures the corresponding coordination pattern by defining the operator named *NoTimeinRefinedActivity*. The operator selects instances of DSE startActivity and finishActivity by using their context. As a result, the pairs selected identify the starting and finishing of an activity. Then, we select the activities that represent a state (Listing 6: line 24). To do so, we use the onEnterAction defined in the context of State. Then, we use the selected instances of DSE entering to select instances of DSE ticks of the corresponding local clock (Listing 6: line 25). The coordination rule must specify how much time is consumed during the execution of an activity. First, we use the event expression *SampledBy* to create a local event named *sampled* which ticks always after the startActivity instance, and coincides with the occurrences of the instance of the corresponding DSE ticks (Listing 6: line 27). Second, we synchronize the event sampled with the finishing of the activity by using a causality relation (Listing 6: line 28). This results for instances of ticks to occur only after the activity has finished its execution.

The coordination rule presented earlier can be built by relying on a B-COoL library. In this case, we have to extend the library *facilities.bcoollib* and add a new event relation named *atomicActivity*. Then, we have to replace the event expressions and relations by the event relation atomicActivity with the corresponding parameters (*i.e.,* dse1, dse2, dse4). The use of the library to define domain specific relations has two major benefits. First, once defined in the library, event relations can be reused in various B-COoL specifications. Second, by defining a dedicated event relation, we improve the readability and modularity of the B-COoL specification.

Listing 6. Timing coordination operator between TFSM and fUML language

```
21 Operator NoTimeinRefinedActivity(dse1 : activity::
       startActivity, dse2 : activity::finishActivity, dse3 :
        tfsm::entering, dse4 : tfsm::ticks)
22  CorrespondenceMatching:
23   when (dse1.name = dse2.name)
24   and  (dse1.name = dse3.onEnterAction.name)
25   and  (dse3.owningFSM.localClock = dse4)
26   CoordinationRule:
27    Local Event sampled = SampledBy(dse1, dse4);
28    Causality(dse2, sampled)
29 end operator
```

B. Use of Coordination Operators in a surveillance camera system

In this section, we develop the heterogeneous model of a surveillance camera system (see Figure 6). To model different aspects of the system, we use the TFSM and the fUML languages. Then, we use the operators developed in the previous section to generate the coordination specification.

The video surveillance system is composed of a camera and a battery control. The camera takes pictures by using either the *JPEG2000* or *JPG* algorithm and is powered by a battery. When the battery is low, the battery control makes the camera use the *JPG* algorithm, thus reducing the quality of the picture but also the energy consumption [27]. When the battery is high, the JPEG2000 algorithm is used instead. In Figure 6, the activity diagrams named *BatteryControl* represents the simple algorithm implemented in the battery control. At the bottom of Figure 6, the TFSM named *CameraControl* represents a partial view of the camera. When the TFSM model is in state *BatteryHigh*, the JPEG2000 algorithm is used (specified by the activity diagram on the right of Figure 6 named *doJPEG2000*). When in state *BatteryLow*, the encoding algorithm is replaced by a mere JPEG algorithm represented by an activity named *doJPEG* (The activity is not shown for lack of space). The transition from one state to another is done when either the *BatteryIsHigh* event or the *BatteryIsLow* event occurs, depending on the current state.

To coordinate the models, we have to specify a timing and hierarchical coordination between the states of the TFSM CameraControl and the activities doJPEG and doJPEG2000. In addition, we have to synchronize the activity BatteryControl and the TFSM CameraControl by coordinating the corresponding Action and FSMEvent. Applying the four operators on these simple models, we generate the expected coordination specification. The coordination generated by using our approach corresponds to eight CCSL relations.

C. Use of the coordination specification

In B-COoL, the generated coordination specification conforms to the CCSL language. Since we are using a formal

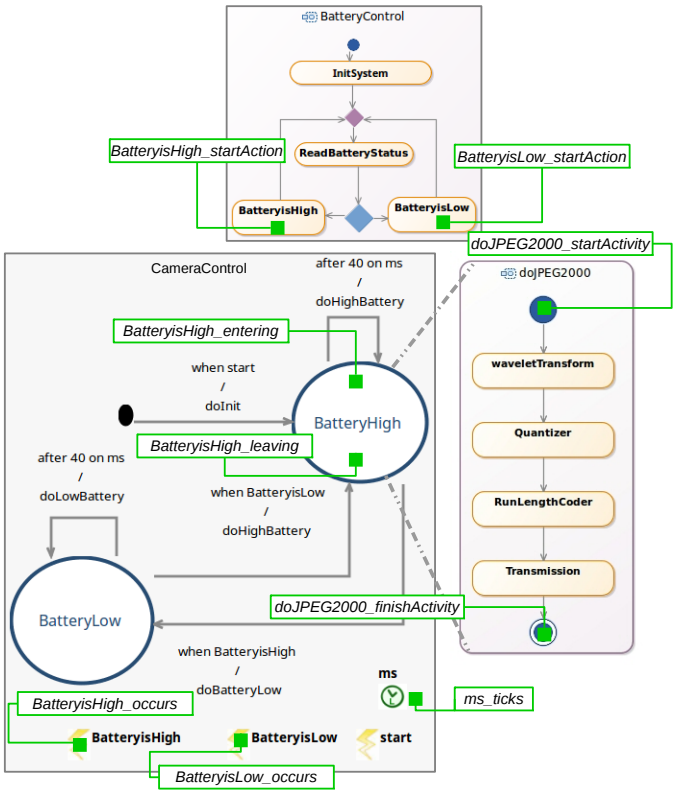

Fig. 6. Hierarchical model of a surveillance camera system and a partial representation of the behavioral interface

language, the integrator can execute and verify the coordination specification of the system. By using the language workbench presented in Section IV-E, the coordination specification generated for the surveillance camera system can be executed and analysed. More precisely, we are able to execute the coordination specification by using TimeSquare, and to explore the state space. For lack of space we do not show the timing output of the execution of the surveillance camera system, however, the models together with a procedure to execute and verify them can be found in the companion web site.

D. Comparison

In B-COoL, the definition of the coordination between languages is based on operators. In particular, coordination rules explicitly define the semantics of the resulting coordination. The reader can notice that variations of the semantics of the resulting coordination can be done by only modifying the coordination rules of the operators. In frameworks like Ptolemy, such a variation is only supported by modifying the framework itself. For instance, in Ptolemy, this means changing the current implementation of a *director* written in Java. The same problem appears in ad-hoc translational approaches [26], where the transformation needs to be changed. Since this state of the art approach is using general-purpose transformation frameworks, this work needs a good knowledge of coordinated languages as well as a good knowledge of the transformation language itself. This is beyond the expected skills of an integrator. In our approach, we are using a language dedicated to integrator experts thus easing the understanding and adaptation of the B-COoL specification.

The definition of domain specific coordination operators enables the automation of the coordination between models. For instance, in the case of the video surveillance system, the application of the operator generates eight CCSL relations. By manually coordinating the models (as proposed in [7] or when using a coordination language), this would require to specify each relation manually. The reader can notice that the number of relations increases with the number of model elements involved in the coordination. For instance, for a system with N cameras, the integrator would need to specify 8*N relations. Our proposition is to leverage this task for the integrator at the language level and then to generate all the required relations accordingly.

Regarding system execution and verification, both coordination languages and coordination frameworks allow to execute the coordinated system, however, the verification varies from one approach to another. Some coordination languages rely on a formal language thus providing verification. Differently, in Ptolemy, the main validation method is based on the simulation of the coordinated system [11]. In our approach, by relying on CCSL, we are able to provide execution and verification of the coordinated system.

VI. Related work

We categorize the related work into those approaches that manually coordinate models (*e.g.*, coordination languages) and those that capture a given coordination pattern between languages (*e.g.*, coordination frameworks) in order to generate the coordination between models.

The use of a language to model explicitly the coordination between models is proposed by Carriero et al. [4] by using a dedicated language named *Linda*. Then, several coordination languages have been developed, *e.g.*, Esper [6], Rapide [5]. Other approaches additionally provide a syntax based on components to specify the coordination. For instance, in BIP [28], interactions between automata are based on connectors. Similarly, MetroII [29] relies on connectors and ports between heterogeneous components. In our approach, we are not proposing a new model of coordination. We aim to capture coordination patterns by specifying the coordination between languages instead of between models. Then, when a coordination pattern is instantiated, the coordination specification is generated. In particular, we focus on the coordination between models conforming to heterogeneous modeling languages. This problem has been recently addressed in [30] where authors propose a language named *CyPhy*. Similarly to our approach, the semantics of modeling languages (*e.g.*, SysML, AADL) is abstracted by using the notion of semantic interface. However, such an interface is used to transform models into *CyPhy* model integration space. Then, the coordination between models is manually specified in FORMULA. In this sense, CyPhy remains a coordination language that relies on a formal language to specify the coordination between models. In our approach, we also use a formal language to specify

the coordination between models (*i.e.,* CCSL), but such a coordination specification is generated. This work, however, gives a good incentive to use other formal languages, instead of CCSL, for the generated coordination specification.

In coordination frameworks and ad-hoc solutions, authors capture a given coordination pattern and generate the coordination specification between behavioral models. The frameworks Ptolemy [9] and ModHel'X [31] propose a hierarchical coordination pattern between heterogeneous models where the semantics of the models is described by a Model of Computation. Less systematic than a framework, ad-hoc solutions capture a given coordination pattern between a set of particular languages [10]. For instance, MASCOT [10] integrates Matlab and SDL. Despite the fact that these approaches manage to capture a given coordination pattern between languages, they encode the coordination pattern inside the tool thus resulting in a fixed relation between languages, and a limited tuning of the coordination. However in our approach, coordination patterns are captured by the integrator using a dedicated language.

In a recent work [32], authors allow the specification of different coordination patterns but between two fixed languages: *i.e.,* a DSML for physical models and a General Purpose Language. This approach relies on an *interface* and *composition filters*. The interface exposes an event model on the execution semantics of DSML models. Then, composition filters allow the user to filter events during the execution of the physical model, and execute certain behavior when an event matches. In this sense, both a composition filter and an operator in B-COoL are similar. However, while composition filters can be used to evaluate a variable, and then to execute a behavior, a correspondence matching cannot be used to evaluate a variable during the execution of the model, and then to execute a coordination rule. The development of such a feature remains future work.

VII. Conclusion and Future work

In this paper, we address the problem of the coordination of behavioral models by reifying coordination patterns at the language level. We present B-COoL, a dedicated (meta)language to capture coordination patterns between modeling languages and generate a formal coordination model for conforming models. Our workbench provides a support for simulation and analysis. Using B-COoL, the know-how of an integrator is made explicit, stored and shared in libraries and amenable to analysis. In future work, we plan to extend B-COoL to use data values to build more expressive coordination patterns.

Acknowledgment

This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011) and by the NSFC (grant 91418203).

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] Esper, "Espertech," 2009.

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

[8] G. A. Papadopoulos and F. Arbab, "Coordination models and languages," CWI, Tech. Rep., 1998.

[9] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity – the Ptolemy approach," *Proc. of the IEEE*, 2003.

[10] P. Bjureus and A. Jantsch, "Modeling of mixed control and dataflow systems in MASCOT," *VLSI Systems, IEEE Transactions on*, 2001.

[11] A. Girault, B. Lee, and E. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE TCAD*, 1999.

[12] D. Balasubramanian, C. S. Păsăreanu, M. W. Whalen, G. Karsai, and M. Lowry, "Polyglot: Modeling and analysis for multiple statechart formalisms," in *ISSTA*, 2011.

[13] B. Chapman, M. Haines, P. Mehrota, H. Zima, and J. Van Rosendale, "Opus: A coordination language for multidisciplinary applications," *Sci. Program.*, 1997.

[14] G. Winskel, "Event structures," in *Petri Nets: Applications and Relationships to Other Models of Concurrency*, 1987.

[15] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE TCAD*, 1998.

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

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

[18] A. Benveniste, B. Caillaud, L. Carloni, and A. Sangiovanni-Vincentelli, "Tag machines," in *ACM Emsoft*, 2005.

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

[20] J. Deantoni and F. Mallet, "ECL: the Event Constraint Language, an Extension of OCL with Events," INRIA, Research report, 2012.

[21] *UML Object Constraint Language (OCL) 2.0*, OMG, 2003.

[22] A. Arnold, "Transition systems and concurrent processes," *Mathematical problems in Computation theory*, 1987.

[23] J. Deantoni and F. Mallet, "TimeSquare: Treat your Models with Logical Time," in *TOOLS*, 2012.

[24] M. Vara Larsen, J. Deantoni, B. Combemale, and F. Mallet, "D3.1.2 - Language Composition Operator," Tech. Rep., 2014, http://gemoc.org/wp-content/uploads/2015/07/D3.1.2.pdf.

[25] *Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0*, OMG, 2011.

[26] M. Di Natale, F. Chirico, A. Sindico, and A. Sangiovanni-Vincentelli, "An MDA approach for the generation of communication adapters integrating SW and FW components from Simulink," in *ACM/IEEE Models*, 2014.

[27] M. Rhepp, H. Stgner, and A. Uhl, "Comparison of jpeg and jpeg 2000 in low-power confidential image transmission," in *SPC*, 2004.

[28] S. Bliudze and J. Sifakis, "The algebra of connectors: Structuring interaction in BIP," in *Emsoft*, 2007.

[29] A. Davare, D. Densmore, L. Guo, R. Passerone, A. L. Sangiovanni-Vincentelli, A. Simalatsar, and Q. Zhu, "metroII: A design environment for cyber-physical systems," *ACM TECS*, 2013.

[30] G. Simko, T. Levendovszky, S. Neema, E. Jackson, T. Bapty, J. Porter, and J. Sztipanovits, "Foundation for model integration: Semantic backplane," in *ASME IDETC/CIE*, 2012.

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

[32] A. Roo, H. Sözer, and M. Akşit, "Composing domain-specific physical models with general-purpose software modules in embedded control software," *Softw. Syst. Model.*, 2014.