Grant ANR-12-INSE-0011

ANR INS GEMOC

D3.1.2 - Language Composition Operator
Task T3.1

Version 1.0

Authors

Author	Partner	Role
Matias Vara Larsen	I3S/INRIA AOSTE	Lead author
Julien DeAntoni	I3S/INRIA AOSTE	Contributor
Benoit Combemale	IRISA/INRIA Triskell	Contributor
Frédéric Mallet	I3S/INRIA AOSTE	Contributor

Contents

Language Composition Operator

1. Introduction

1.1 Purpose

Current developments of complex systems involve the use of multiple Domain Specific Modeling Languages (DSMLs), which capture different system aspects, both structural and behavioral. To understand the global and emerging behavior of the system, it is mandatory to eventually coordinate the different models. Coordinating the models is a complex and tedious but primordial task. For now, the coordination of the models from different DSMLs is done by integrator experts. Instead of specifying the coordination for each new system (or new model in the system), we propose to leverage on their knowledge by providing them a meta language to specify how they apply some coordination patterns between DSMLs. By providing such a metalanguage, the knowledge can be shared, tuned, reviewed, etc. Also, the coordination between specific models can be automated. We named this language B-COoL for Behavioral Coordination Operator Language. It is validated in this deliverable by defining a coordination operator between two DSMLs, namely TFSM (Timed Finite State Machine) and fUML.

1.2 Perimeter

This document is the version v1 of the D3.1.2 deliverable. The document presents a language to specify composition operator. Here composition is a generic term and the chosen solution is to use coordination operator, *i.e.,* a specific kind of composition. Consequently the document introduces the Behavioral Coordination Operator Language (B-COoL). In the previous deliverable D3.1.1-v2, we presented a first sketch of the language. In this document, we go further by proposing to leverage the knowledge of integrator experts by providing them a dedicated meta language to specify how they apply coordination patterns between DSMLs. In addition, we present the first integration of B-COoL into the GEMOC STUDIO.

1.3 Summary

The present document is organized as follows. In Section 2.1, we introduce the currents problems of the coordination of model behaviors, and we show how these problems can be solved by expressing the coordination at the meta model level. In Section 2.2, we introduce the notion of language behavioral interface by relying on a language named Timed Finite State Machine (TFSM). This language is used latter in Section 2.3 to illustrate B-COoL , a meta-language dedicated to the specification of behavioral coordinators operators between languages. In Section 2.4, we use B-COoL to specify the synchronized product between TFSM language and fUML language.

2. Behavioral Coordination Operator Language

2.1 From Coordination of Model Behaviors to Coordination of Language Behaviors

In the development of complex software intensive systems, multiple stakeholders are usually working on a sub part of the system. Each stakeholder uses a language suitable to their problems, possibly making each sub-part specified in a different language. The system specification is then heterogeneous, *i.e.,* made up with models in various different languages. To obtain a global understanding of the system, the behaviors of models have to be coordinated. The coordination consists in integrating a number of different (executable)

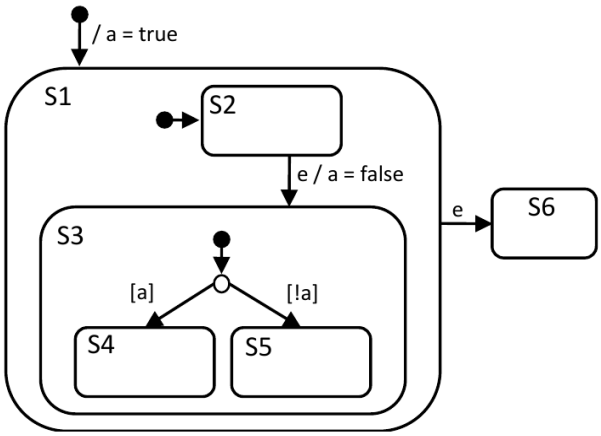

Figure 2.1: Hierarchical coordination of state machines

models. It is a means of obtaining the (desirable or unexpected) emerging behaviors resulting from the interaction of the models. In other words, the behavior of the whole system is the behaviors of the individual models plus the interactions between them. Therefore, specifying the coordination is a mandatory development step to verify and validate the system as a whole.

The coordination between models can be modeled by a *coordination language* [1, 19, 3, 15]. These approaches allow the user to explicitly model the interactions between different models by providing a dedicated language. The developer thus defines a coordination model to specify how models interact with each other. Since the coordination is explicitly modeled, the global behavior of the system is explicit and amenable to reasoning. As a result, different interactions can be studied during the integration of the system. However, when the coordination is specified at the model level, such a coordination model only captures the resulting reasoning of the integrator for a particular problem. This leads to the redundant activity of writing coordination models for each new models.

To systematize the coordination of models, some approaches specify the coordination at the metamodel level. This is the case of *coordination frameworks* [14, 9] which implicitly define a set of rules between languages, thus the coordination between models is automatically generated. Similarly, there exist ad hoc solutions that propose to couple two particular languages together. For instance, the MASCOT [7] methodology specifies how Matlab and SDL can be integrated.

These approaches are interesting because they are able to express some interactions rules at the language level so that model coordination can be automatically achieved. In this sense, they capture the knowledge of how the coordination between models is done and they express this knowledge at the language level. Thus, the knowledge can be applied/used on any model conforming these languages. However, these approaches either hide (hard-coded) the specification in the tool or fix the language to be applied (fixed coordination pattern), thus limiting in one hand the verification and validation activities, and in the second hand, the definition of domain specific coordination without putting hand in the code of the associated framework.

At this point we have to highlight that there is non a golden coordination for a given set of models. The coordination varies from one domain to other thus varying the semantics. Different approaches could provide different coordination depending the domain. In [5], a very interesting example of a homogeneous hierarchical coordination is used (see figure 2.1) to highlight the possible variations in the semantics of the coordination depending on the tool used. The example corresponds with the hierarchical coordination of state machines where a state can refine into a new state machine. From figure 2.1, we suppose the chart is in state S2 and the value of *a* is true. If the event *e* occurs, the behavior exhibited by the overall system is different depending on the approach: the Stateflow semantics will terminate in state S6; the UML State Machine semantics will terminate in state S4, and the Rhapsody semantics will terminate in state S5. Since each approach interprets differently the hierarchical coordination, we obtain three different behaviors. This

difference in the resulting behavior is desired to fit different domain but if the variation is hidden in a tool and is not made explicit, it can lead to misunderstandings between different teams. Therefore, the developer has be able to specify variations in the coordination, but such variations must be explicit modeled to ensure a global understanding of the system.

In this paper, we propose to capture the knowledge of integrator expert, which is already understood implicitly, in a dedicated language. This language must express, based on explicit rules at the language level, how models written in these languages can be coordinated. In other words, the language must enable the integrator expert to define coordination patterns. Once specified in a dedicated language, this knowledge can be reused, shared and tuned by different integration experts. Also, such specification can be exploited by generative techniques to automatically generate an explicit coordination when specific models are used.

In our approach, we base on the language behavioral interface, which is introduced in the next section by using an example language named TFSM (Timed Finite State Machine). This language is also used in section 2.3 to introduce the first behavioral coordination language named B-COoL .

2.2 Language Behavioral Interface

Current coordination languages [18] aim to separate the computations concerns from the coordination concerns by treating the models as boxes with a clearly defined interface. The content of the interface varies from one approach to others. While in [10], the interface is just a list of methods provided by the model, in RAPIDE [19], the interface is built by events. Other approaches go further and also exhibit part of the internal concurrency. For instance, in [6], the interface contains services and events, but also properties that express requirements on the behavior of the components. The interface is aimed at easing the coordination of the models by exposing the needed information. In analogy, a behavioral coordination language should take advantage of a language behavioral interface. The language behavioral interface must provide the elements of semantics to ease the coordination of language behavior. In our approach, we base on the notion of language interface as proposed in [11] to specify the coordination of languages. In the following, we briefly sum up the approach by using an example language together with its behavioral interface.

In [11], the authors define an executable language as a 4-tuple $< AS, DSA, DSE, MoCC >$ where the AS is the Abstract Syntax of the language, the DSA defines both the data that represents the execution state of the model and the execution functions that modifies this execution state. The MOCC represents the, possibly timed, causalities and synchronization of the system by using some events and constraints between them. Then, the *Domain Specific Events* (DSE) link together the three other parts. A DSE is an event type, defined in the context of a metaclass of the AS that links an event from the MOCC with the call to an execution function from the DSA. We illustrate the AS and DSE concepts by using a simple automaton language named Timed Finite State Machine (TFSM); a state machine language with timed transitions (see Figure 2.2).

The syntax of the TFSM is described by its metamodel (see Figure 2.2). A *System* is composed of a set of *TFSMs*, a set of global *FSMEvents* and a set of global *FSMClocks*. Each *TFSM* is composed of *States* and each state can be the source of outgoing guarded *Transitions*. A guard can be specified by the reception of a *FSMEvent* (*EventGuard*) or by a duration relative to the entry time in the incoming state of the transition (*TemporalGuard*). When fired, a transition can generate a set of *FSMEvent* occurrences. The TFSM language defines the following DSE : the *entering* in (the *leaving* of) a state, the *firing* of a transition, the *occurs* of a FSMEvent and the *ticks* of a FSMClock. These events are defined in the context of a metaclass, *e.g., entering* and *leaving* are defined in the context of State and *firing* in the context of Transition (see Figure 2.2).

In our approach, the DSE act as the behavioral interface of the language. At the model level, the language behavioral interface enables to automatically extract the model behavioral interface for a model conforming to the language. When a metaclass is instantiated, the corresponding DSE are instantiated; *e.g.,* for each instances of the metaclass *State*, the DSE *entering* is instantiated. In the case of *TFSM0* (see Figure 2.2), since two States are instantiated (*idle* and *danger*), there are two instances of the DSE *entering*: *idle_entering* and *danger_entering*. The set of all instances of the DSE makes the model behavioral interface. More precisely, both the instances of DSE and an abstract representation of the constraints between them are exposed. The model behavioral interface is then a symbolic representation of an event structure [21].

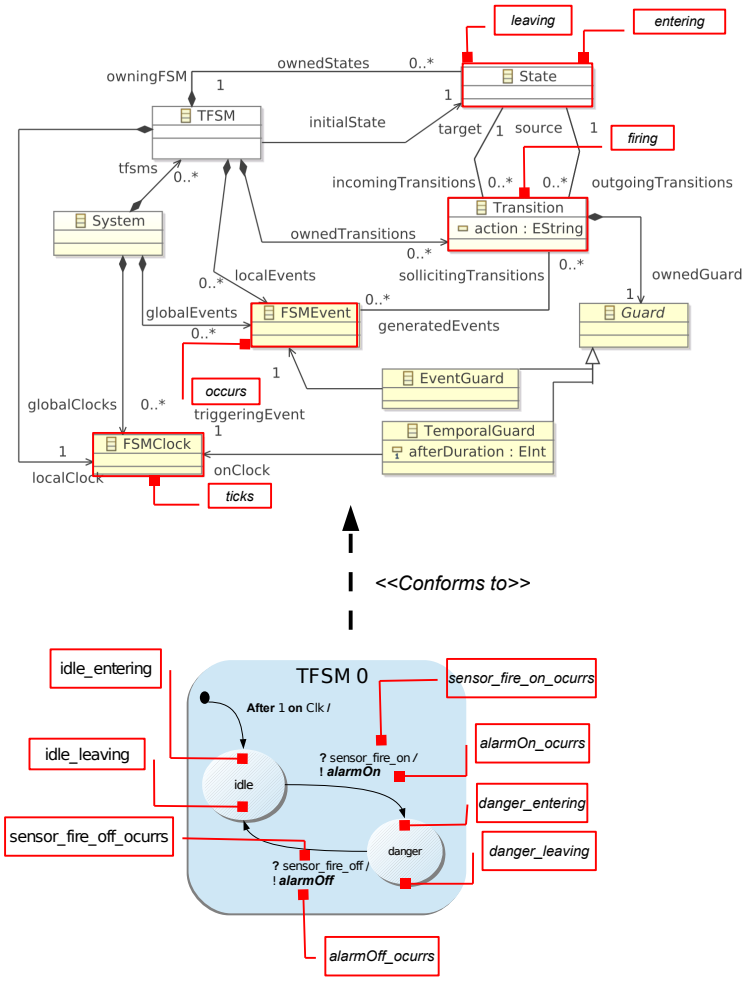

Figure 2.2: The TFSM metamodel and its behavioral interface, and a model behavioral interface conforms to such language behavioral interface

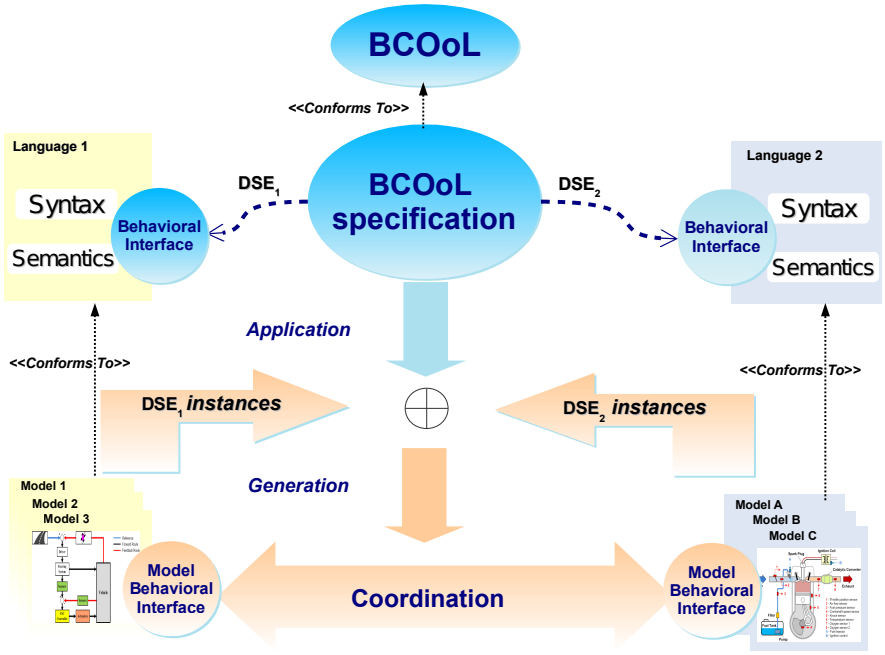

Figure 2.3: Behavioral Composition Language

These events make explicit and accessible the concurrency and time aspects of the model and can be used to coordinate models in different languages as illustrated in [20].

To sum-up, the approach proposed in [11] makes explicit the language behavioral interface in terms of the *Domain Specific Events* (DSE) of the language. This enables to automatically extract the behavioral interface for a model conforming to the language. In the next section, we introduce B-COoL that expresses the coordination of language behaviors by relying on the notion of DSE as behavioral interface of languages.

2.3 BCOoL

2.3.1 Overview

B-COoL is a dedicated language that is aimed at capturing the knowledge of an integrator expert. To do so, B-COoL allows the developer to specify at the language level how to coordinate models conforming to such languages. While other solutions propose to use a new unified language by composing the syntax of languages, B-COoL proposes to specify relationships between the behavioral semantics of different languages that applies between models to automatically generate the coordination (See Figure 2.3).

Inspired by structural composition approaches [16, 17, 13, 8], B-COoL enables the user to specify *matching rules* by comparing elements from the language behavioral interface (*i.e.,* the DSE) together with the specification of a *coordination rule*. A coordination rule specifies some, possibly timed, synchronization and causality. This specification applied on specific models to create sets of *matched* DSE *instances* on which the coordination specification is applied.

To illustrate the approach, we use B-COoL to specify the synchronized product operator of the example language (see section 2.2). This operator is well known in the literature [4] but, for now, only expressed in natural language; it relies on the concepts defined in the semantics of TFSM, *i.e.,* the event occurrences, and describes what are the specific event occurrences that must be synchronized by a rendez-vous. The specification can be applied between models to automatically synchronize theirs behaviors. Note that this first behavioral coordination is homogeneous (*i.e.,* it involves a single language). An example of heterogeneous coordination (*i.e.,* that involves several languages) is provided in section 2.4.

In the following, we first present the abstract syntax, and then, the execution semantics of B-COoL .

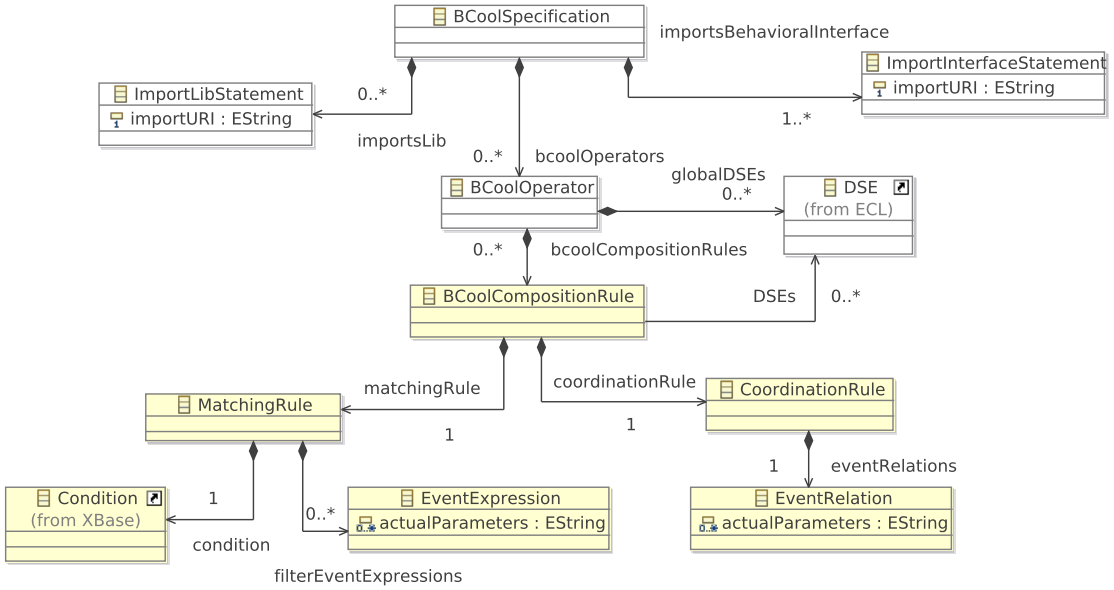

Figure 2.4: B-COoL abstract syntax

In these sections, the synchronized product operator is used as running example. Then, we finish with a tooling section which shows the current implementation of B-COoL by using the Gemoc studio.

2.3.2 Abstract Syntax of B-COoL

The abstract syntax of B-COoL is described by its metamodel (see Figure 2.4). The main elements of the *BCoolSpecification* are the language behavioral interfaces (*importsInterfaceStatements*) and the operators (*BCoolOperator*) (see Figure 2.4). The interfaces provide the DSE needed to express the coordination. These interfaces must be imported, and since several languages can be involved, B-COoL supports the importing of several behavioral interfaces. For instance, to specify the synchronized product of TFSM, its behavioral interface must be imported. The behavioral interface will provide the DSE needed to specify the coordination (see Figure 2.2).

In B-COoL , the specification is split up in operators (*BCoolOperator*). An operator defines how a set of the imported interfaces is coordinated. In the running example, one operator must be defined which specifies how the behavioral interfaces of TFSM are coordinated. An operator specifies the coordination by defining composition rules (*BCoolCompositionRule*). While operators are defined between interfaces, composition rules are defined between DSE . Composition rules select and coordinate instances of DSE . For the running example, we have to select and coordinate instances of the FSMEvents. To do so, we need to define a composition rule which compares and coordinates all the instances of the corresponding *occurs* DSE .

In B-COoL , a composition rule is made by a *matchingRule* and a *coordinationRule*; a matching rule allows the user to identify pairs of matching instances of DSE , whereas the *coordinationRule* specifies how the selected instances must be coordinated. For instance, for the running example, the matching rule will specify which instances of DSE *occurs* must be selected, whereas the coordination rule will specify how such instances must coordinated.

To perform the matching, a matching rule contains a *condition* and optionally, a *filterEventExpression*. The condition allows the user to specify a query on the context of DSE . Thus, instances of DSE can be selected by querying its context. The condition is an OCL expression that is made up with an OCL query over a context of DSE and a comparison operator. For our running example, the instances of FSMEvents to be coordinated have the same name. Thus, we can use the attribute name defined in the context of

FSMEvents to select pairs of instance of DSE *occurs*.

Optionally, a matching rule allows the user to specify a query on the occurrences of the selected instances of DSE . To do so, a matching rule can contain several *filterEventExpressions*. An filterEventExpression is an *EventExpression* where the actual parameters can be the instances of DSE resulting from the matching condition. Such an EventExpression aims to select only some occurrences of such instances. For instance, the user may select only one occurrence every three for a matched DSE .

A *coordinationRule* specifies how the selected instances of DSE must be coordinated. Each coordination rule specifies an *EventRelation* where the actual parameters can be the instances of DSE resulting from the matching and/or the filtered occurrence of instances of DSE . This results in a coordination of some occurrences of instances of DSE . In B-COoL , the definition of the event expressions and event relations are in a dedicated library which must be imported (*ImportedLibStatement*). This is further explained in the following section. For the synchronized product, the selected events must be strongly synchronized. We express this coordination by using a "rendez-vous" relation. As a result, the selected instances of DSE occurs simultaneously. The definition of this relation must be imported.

An operator can also contain *globalDSEs* defined by the user that can be used as global events of synchronization. To do so, once defined in an operator, these new event can be used in any coordination rule defined into the operator.

2.3.2.1 B-COoL library

The event expressions and event relations used in a B-COoL specification are defined separately in a *B-COoL library*, which must be imported by the specification (Figure 2.4). The aim of the library is to enable the user to define coordination patterns suitable to its domain so that they can be used in a B-COoL specification.

A library is a set of declarations together with their formal parameters. The declarations are the entities used in a B-COoL specification. A library also contains some definitions, which give the actual behavior of the declarations it refers to. To specify the definitions, we reused the definition part of CCSL [2], a formal language dedicated to event constraints. It allows us to take benefits from the formal and tooled aspect of this language.

There is a difference between an expression and a relation. Generally speaking, event expressions create a new event from its parameters (*e.g.,* the *Union*, or the *Intersection* of its parameters). It 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 of to define some filter used in the matching rules. Differently, relations are constraints that specify the correct evolution of the events given as formal parameters. For instance they can define a *Rendezvous* synchronization between its parameters. Lots of other relations, more or less complex can be defined (*e.g., Causality*, *FIFO* or a specific protocol).

B-COoL comes with a set of predefined useful declarations but we use the library mechanism to let the integrator experts define specific constraints depending of their specific problems and domain.

Since we are based on the formal language CCSL to define the relations and expressions, the application of the operators results in a CCSL specification as used for coordination in [20]. This allowed us to simulate and animate the resulting coordination by using the framework associated to CCSL: TimeSquare [12].

2.3.3 Execution semantic

The B-COoL specification is applied on models conform to the coordinated language. The number of input models corresponds with the number of imported interfaces. The application of a B-COoL specification results in the generation of the coordination between the input models.

A B-COoL operator declares the interfaces on which it applies the composition rules. For each of these interfaces, the behavioral model interface is extracted from the input model. This results in N model behavioral interfaces, *i.e.,* in N sets of event set named *DSEinstanceSets* where N is the number of interfaces the operator declares.

A composition rule declares the type of event it expects (*i.e.,* the DSE on which the composition is specified). For each composition rule, each model behavioral interface in *DSEinstanceSets* is restricted so that

only the event typed by the DSE declared by the composition rule are kept.

Consequently, the input of a matching rule is m event sets (named *CompositionRuleSets*) where m is the number of DSE the composition rule declares. Note that in each of this set, the events are instances of the DSE declared by the composition rule.

For each m-tuple of events so that it contains an event picked from each sets in *CompositionRuleSets*, the matching-rule is applied. From all these m-tuples, the ones that fulfills the matching condition are kept, the others are ignored. Consequently, after the matching condition, there is a set of m-tuples named *matchedEventTuples*.

The occurrence filtering is working on one specific event of each tuple on *matchedEventTuples*. Its goal is to replace the original event in the tuple by a new event in which some of its occurrences have been removed/filtered. Consequently, some event filters are generated in the targeted coordination language. The result is a set of m-tuples with the same size than *matchedEventTuples*. It is named *filteredEventTuples*.

Finally, each tuple in *filteredEventTuples* the coordination is generated in the targeted coordination language. More precisely, for each tuple in *filteredEventTuples*, a call to the specified EventRelation is done, where its actual parameters are the events from the tuple.

2.3.4 Tooling

B-COOL is developed on top of the eclipse platform as a set of plugins. More precisely it is integrated to the GEMOC studio[1]. The GEMOC studio is the integration of various Eclipse Modeling Framework (EMF) based technologies adequate to 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 to EMF). The textual concrete syntax has been developed using Xtext[2], providing advanced edition and manipulation facilities.

The development of a new operator begins by creating a new B-COOL specification (step 1 in Figure 2.5). For instance the screenshot provided in Figure 2.6 shows the resulting B-COOL editor in which the specification of the synchronized product for TSFM is shown. The specification begins by importing the ECL specification of TFSM (Figure 2.6:lines 5 and 6), which is used as its behavioral interface. Since the behavioral coordination is homogeneous (it is defined between the same language) the interface is imported twice and named *TFSMLeft* and *TFSMRight*. To coordinate these interfaces, we define the operator *SyncProductOperator* (Figure 2.6: line 8). The operator has to select instance of DSE *occurs* and coordinate them. This is specified by the composition rule *MatchingandCoordinateSharedEvents* which compares pairs of the DSE *occurs* from both interfaces (Figure 2.6: line 10). To select pairs of instances of *occurs*, the matching rule compares the attributed *name* defined in the context of the DSE . In the coordination rule, the event relation must enforce an strong synchronization between the events. To do so, the relationship *Rendezvous* is used (Figure 2.6: line 12), thus resulting in a simultaneous occurrence of events. The B-COOL specification is thus used to generate the coordination between models.

The automatic generation of the coordination is made in two steps. The first step consists in the automatic generation of a transformation by using an higher order transformation written in acceleo[3]. The acceleo transformation translates the B-COOL specification into a QVTo transformation (step 2 in Figure 2.5).

The QVTo transformation takes in input any models conforming to the languages used in the composition operator and generates in output the coordination for these models (step 3 in Figure 2.5). Note that we reused, as a coordination language CCSL , an executable, formal and declarative language to specify synchronizations and timed causalities between events. This is convenient in our case since this language has been used in [11] to specify the execution model of a model. It is then easier to coordinate the models execution. We are currently investigating what other coordination languages may be used in B-COOL .

By using the TFSM language proposed as packaged example by the GEMOC studio, we built two simple TFSMs models (see Figure 2.8). In this example, when *tfsmSensorAlarm* detects the event *sensorOn*, it switches to *Danger* and fires the event *alarmOn*. This makes the *tfsmSensorLight* switches from *Green* to *Red*. When *tfsmSensorAlarm* detects the event *sensorOff*, it switches to *Idle* and *tfsmSensorLight* switches

[1] http://gemoc.org/studio/
[2] http://www.eclipse.org/Xtext/
[3] http://www.eclipse.org/acceleo/

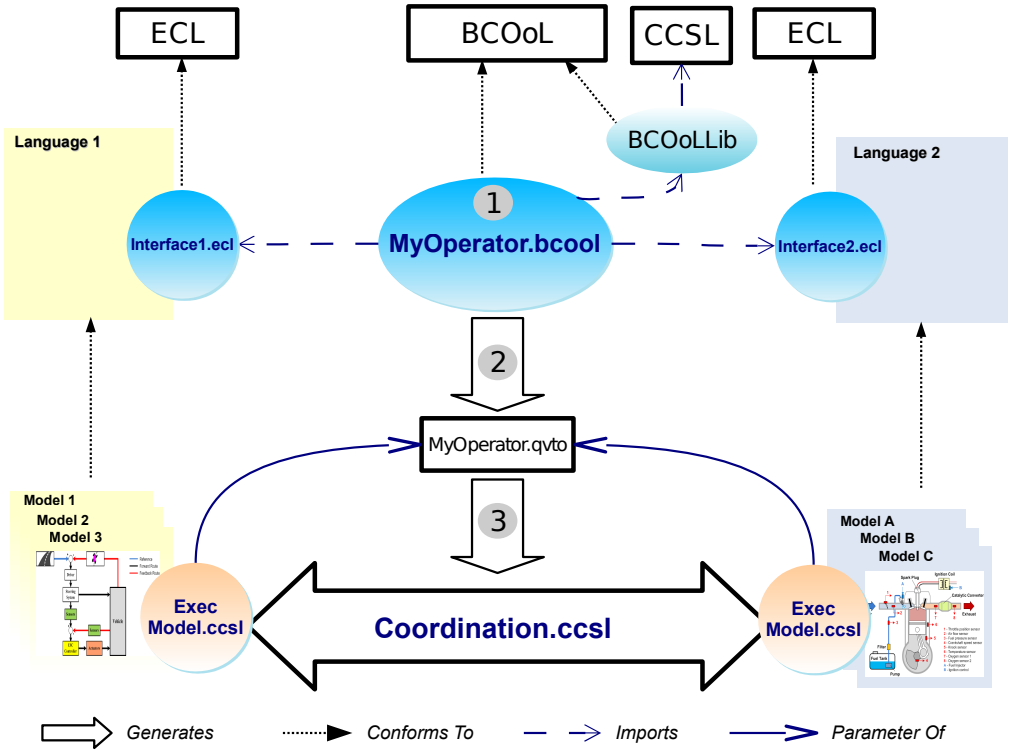

Figure 2.5: B-COoL *Language Workbench*

```
1  SyncProductbetweenTFSMs
2
3  ImportLib "platform:/resource/org.gemoc.bcool.example.tfsmproduct/library/facilities.BcoolLib"
4
5  ImportInterface "platform:/org.gemoc.bcool.example.tfsmproduct/interfaces/tfsm.ecl" as "TFSMLeft"
6  ImportInterface "platform:/org.gemoc.bcool.example.tfsmproduct/interfaces/tfsm.ecl" as "TFSMRight"
7
8  Operator SyncProductOperator(TFSMLeft, TFSMRight)
9
10       CompositionRule MatchingandMergingSharedEvents (TFSMLeft.ocurrs, TFSMRight.ocurrs)
11             MatchingRule : when "(TFSMLeft.ocurrs.name = TFSMRight.ocurrs.name)"
12             CoordinationRule  : RendezVous("TFSMLeft.ocurrs", "TFSMRight.ocurrs")
13       end rule;
14
15 end operator;
```

Figure 2.6: B-COoL specification of synchronized product automaton

to *Green*. The synchronization of these TFSMs is done by using the events *alarmOn* and *alarmOff*. We use the synchronized operator to generate the coordination thus resulting in a strong synchronization between the events. Then, by using TimeSquare, both the coordination and the models can be simulated. Figure 2.9 illustrates the timing output of the simulation of the whole specification.

2.4 Heterogeneous synchronized product operator

In this section, we use B-COoL to capture a coordination pattern where models are strongly synchronized by using events with the same name. This is the generalization of the synchronized product for the heterogeneous case (see section 4). More precisely, we focus on the synchronized product between TFSM language and fUML language.

To illustrate the pattern, we show in Figure 2.11 the example of a heterogeneous model that is synchronized by using events with same name. This example corresponds with a coffee machine; when a coin is inserted the coffee machine becomes unlocked and the user can choose the coffee. When the coffee is ready, the coffee machine becomes locked. A TFSM model has been used for the control flow aspects,


```
1  BCoolLibrary facilitiesinCCSL{
2        imports{
3              import "./ccsllibrary/kernel.ccslLib" as kernel;
4              import "./ccsllibrary/CCSL.ccslLib" as CCSLLib;
5        }
6        RelationDefinitions{
7              RelationDefinition RendenVousDef[RendeVous]{
8                    Relation ClockCoincid[Coincides] (Clock1->DseA, Clock2->DseB)
9              }
10             RelationDefinition CausalityDef[Causality]{
11                   Relation ClockCauses[Causes] (Clock1->Dse1, Clock2->Dse2)
12             }
13       }
14       RelationDeclarations{
15             RelationDeclaration RendenVous(DseA:dse, DseB:dse)
16             RelationDeclaration Causality(Dse1:dse, Dse2:dse)
17       }
18 }
```

Figure 2.7: B-COOL *library for CCSL*

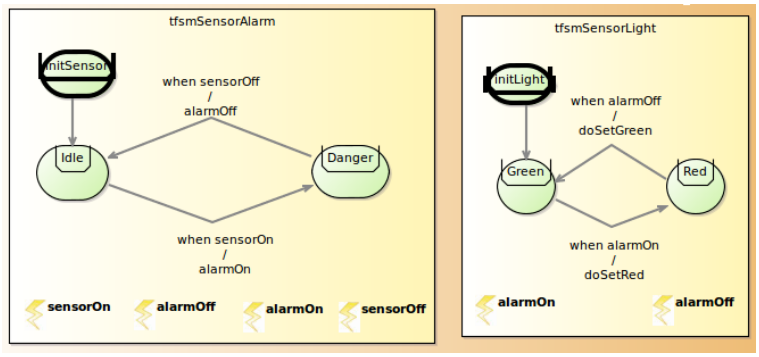

Figure 2.8: Application of the running example operator in B-COOL

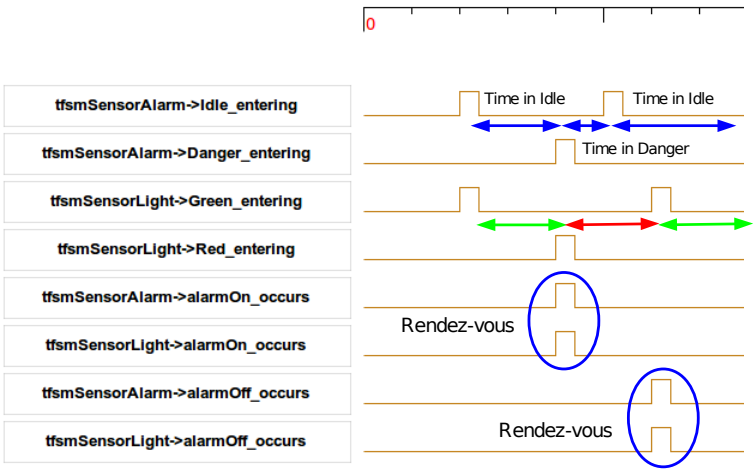

Figure 2.9: Simulation of the resulting coordination

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

Figure 2.10: Partial ECL *specification of Activity Diagram*

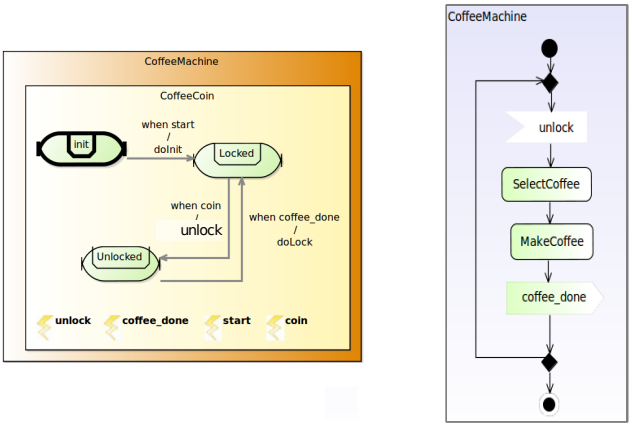

Figure 2.11: Example of coordination pattern presented in section 2.4

whereas an Activity diagram has been used for the data flow aspects. The models are synchronized by using the events *unlock* and *coffee_done*.

The instance of FSMEvent named unlock is used to synchronize with the instance of Action named unlock. To do so, instance of the DSE *occurs* in the context of FSMEvent (see Figure 2.2) is coordinated with instances of DSE *startAction* in the context of Action. Figure 2.10 partially represents the ECL specification of the Activity where two DSE are defined in the context of Action: startAction and finishAction.

In B-COOL , we capture this coordination pattern by defining a new specification named *SyncProduct-TfsmwithfUML* (Figure 2.12). We begins the specification by importing the language behavioral interface of each language. To do so, the ECL specification are imported (Figure 2.12: line 5 and 6).

We specify the coordination between these interfaces by defining an operator named *SyncProductOperator* (Figure 2.12: line 9). The operator will coordinate models conform to TFSM language with models conform to fUML language by comparing instances of DSE *occurs* and instances of DSE *startAction*. This is specified by defining a composition rule named *MatchingandCoordinationSharedEvents*. Then, the instances of these events are selected by comparing the name attributed defined in its context (Figure 2.12: line 12). The selected pairs are thus coordinated by the Rendezvous event relation (Figure 2.12: line 13).

Once defined, the operator can be applied between any pair of model conform to TFSM and fUML language resulting in the automatic generation of the coordination.

```
1  SyncProductTfsmwithfUML
2
3  ImportLib "platform:/resource/org.gemoc.bcool.example.productfumlandtfsm/interfaces/library/facilities.bcoollib"
4
5  ImportInterface "platform:/resource/org.gemoc.bcool.example.productfumlandtfsm/interfaces/activitySemantics.ecl" as "I_Activity"
6  ImportInterface "platform:/resource/org.gemoc.bcool.example.productfumlandtfsm/interfaces/interfaces/TFSM.ecl" as "I_TFSM"
7
8
9  Operator SyncProductOperator
10
11       CompositionRule MatchingandCoordinationSharedEvents ( I_Activity.startAction,  I_TFSM.occurs)
12            MatchingRule : when "I_Activity.startAction.name = I_TFSM.occurs.name"
13            CoordinationRule  : RendezVous("I_Activity.startAction","I_TFSM.occurs")
14       end rule;
15
16 end operator;
```

Figure 2.12: B-COOL *specification of synchronized product automaton between Heterogeneous languages*

3. References

[1] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.

[2] C. André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research report, INRIA and University of Nice, May 2009.

[3] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Pract. Exper.*, 5(1):23–70, February 1993.

[4] André Arnold. Transition systems and concurrent processes. *Mathematical problems in Computation theory*, 21, 1987.

[5] Daniel Balasubramanian, Corina S. Păsăreanu, Michael W. Whalen, Gábor Karsai, and Michael Lowry. Polyglot: Modeling and analysis for multiple statechart formalisms. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 45–55, New York, NY, USA, 2011. ACM.

[6] L. Barroca, J.L. Fiadeiro, M. Jackson, R. Laney, and B. Nuseibeh. Problem frames: A case for coordination. In Rocco Nicola, Gian-Luigi Ferrari, and Greg Meredith, editors, *Coordination Models and Languages*, volume 2949 of *Lecture Notes in Computer Science*, pages 5–19. Springer Berlin Heidelberg, 2004.

[7] P. Bjureus and A. Jantsch. Modeling of mixed control and dataflow systems in mascot. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(5):690–703, Oct 2001.

[8] Artur Boronat, Dr. Jos, Meseguer U. I. Urbana-champaign, Dr. Jos, . Cars, U. P. Valncia, Dr. Reiko, and Heckel U. Leicester. Moment: A formal framework for model management, 2007.

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

[10] Barbara Chapman, Matthew Haines, Piyush Mehrota, Hans Zima, and John Van Rosendale. Opus: A coordination language for multidisciplinary applications. *Sci. Program.*, 6(4):345–362, October 1997.

[11] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Fr'ed'eric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In Richard F. Paige Martin Erwig and Eric van Wyk, editors, *6th International Conference on Software Language Engineering (SLE 2013)*, Lecture Notes in Computer Science, Indianapolis, Etas-Unis, 2013. Springer-Verlag.

[12] Julien Deantoni and Frédéric Mallet. TimeSquare: Treat your Models with Logical Time. In Carlo A. Furia, Sebastian Nanz, editor, *TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012*, volume 7304, pages 34–41, Prague, Czech Republic, May 2012. Czech Technical University in Prague, in co-operation with ETH Zurich, Springer.

[13] Marcos Didonet, Del Fabro, Jean Bzivin, and Patrick Valduriez. Weaving models with the eclipse amw plugin. In *In Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006.

[14] 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*, 91(1):127–144, 2003.

[15] Esper. Espertech, 2009.

[16] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. Models in software engineering. chapter A Generic Approach for Automatic Model Composition, pages 7–15. Springer-Verlag, Berlin, Heidelberg, 2008.

[17] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Merging models with the epsilon merging language (eml. In *In Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006*, 2006.

[18] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. Technical report, Amsterdam, The Netherlands, The Netherlands, 1998.

[19] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering, IEEE Transactions on*, 21(4):314–335, Apr 1995.

[20] Matias Vara Larsen and Arda Goknil. Railroad Crossing Heterogeneous Model. In *GEMOC workshop 2013 - International Workshop on The Globalization of Modeling Languages*, Miami, Florida, USA, September 2013.

[21] Glynn Winskel. *Event structure semantics for CCS and related languages*. Springer, 1982.

