Grant ANR-12-INSE-0011

ANR INS GEMOC

D3.1.1 - Identification and formal characterization of the operator for composition and Eclipse-based hierarchical component metamodel

Task T3.1

Version 0.2

Authors

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

Contents

Identification and formal characterization of the operator for composition and Eclipse-based hierarchical component metamodel

1. Introduction

1.1 Purpose

The increasing complexity of system development leads to an increasing number of stakeholders involved in the development of a same system. As a way to reduce this complexity, the system is often scattered into artifacts and each artifact can be defined by using a Domain Specific Modeling Language (DSML) suitable to the nature of the artifact and the habits of the stakeholder. This leads to heterogeneous modeling and imposes a composition of the heterogeneous models used during the development. The purpose of this document is to base the composition on a component model, where each primitive component encapsulates a specific model. Our component model originality is to explicit all the composition choices based on the core notion of Event. Events acts as a natural but mandatory concept for the semantic aware composition of models. We illustrate our proposal through two examples of flat and hierarchical composition between a Timed Final State Machine (TFSM) model and fUML activity model.

1.2 Perimeter

In the last years, computer science was overwhelmed by the increasing complexity of systems. This complexity is naturally propagating to the development of such systems, and increase the number of stakeholders involved in the development of a same system. By using multiple Domain Specific Modeling Languages, model driven engineering proposed a way to scatter the problem into multiple concerns were each stakeholder manipulate a language adequate to its concern. In this context, the DSML efficiently specifies the concepts as well as their behaviour within a particular domain [6]. While the number of DSMLs used has gone up, they have shown to be an effective vehicle for the reuse of knowledge and to ease focusing on a specific concern. The benefits provided by the separation of the preoccupations introduced the problem of heterogeneous modeling, i.e., modeling with various DSMLs. Cyberphysical systems, by their different preoccupations are examples of heterogeneous systems. They are usually composed of very different parts where each part has one or several dedicated DSML (e.g., state machine to specify the control and data flow languages to specify sensors acquisition and treatments). From heterogeneous modeling naturally emerges the need to compose the DSMLs (their concepts and their behaviours) in order to obtain the model of the whole system. We propose a semantic aware component model for the composition of heterogeneous executable DSMLs. DSML are composed of a syntax and a semantics. Various ways to decompose a DSML into modeling units has already been proposed in literature. The decomposition is function of the intention of the modeling activity.

1.3 Summary

In the present document we first propose a way to decompose a DSML into units with the intention to enable the semantic aware composition of such languages. This decomposition tries to cross fertilize ideas from both the language and the concurrency theory. In our proposition, the concepts of Abstract Syntax, Domain Specific Actions, MoC (Model of Computation) and Domain Specific Events are central. We propose to encapsulate models that conform a DSML described according to this decomposition into a hierarchical component model. Such components only expose the information required for model composition. This suits with the idea that complex systems are built by assembling components. Larger components can then be obtained by composing simpler ones. A clear definition of what should be exposed is then essential to understand how the DSMLs can be composed to form a whole system. The organization of this document is as follows: we quickly explains the rational of our DSML decomposition and defines our understanding of heterogeneity. After that, we present the component metamodel. The component is used afterwards to describe two kind of compositions: flat and hierarchical; both illustrated by the composition of a Timed Final State Machine model with a fUML [11] activity model. We finish this section with some limits of our approach. Finally, we describe future work and conclude.

2. Document-dependant structure

2.1 DSML Units

The state of the art describes multiple way to decompose a DSML into interesting units. This decomposition is always motivated by the modeling intention. Usually, there is at least a distinction between the syntactic and the semantic domain but many variants exist. In the following, we propose a decomposition of a DSML into 4 units motivated by the desire to compose heterogeneous DSMLs. In language theory, the semantic domain is usually split into some data representing the execution state and some execution function that manipulate the execution state to reflect its evolution. The focus is made on how the execution state is manipulated by functions. In the concurrency theory, some events are put in relation in order to describe one or several partial order(s), each of them specifying the potential parallelism and the possibly timed synchronizations and causalities in the system. In concurrency theory, data manipulation is seldom dealt with and the focus is made on the event structure representing the system (often with an analysis intention). Model of Computations (MoC [3, 12, 2] were inspired by concurrency theory. Such approaches were able to compose model based on different MoC based on the causalities and synchronizations in the model. However, they do not deal with the problem of having varying abstract syntax or varying execution data and functions (they fix them and only the MoC is varying). We propose to take benefits of both the language theory and the MoC approaches by reconciling them via an explicit notion of Domain Specific Events, making an explicit (and possibly complex) mapping between the event in the MoC and the trigger of the execution functions.

Consequently a is a 4-tuple $DSML \triangleq < AS, DSA, DSE, MoC >$ (see Figure 2.1) where AS is the abstract syntax, DSA are the Domain Specific Actions (defining both the execution state and the execution functions), DSE are the Domain Specific Events and where the MoC is the Model of Computation (defining both Event and constraints between them). These units are quickly detailed in the following to understand the role of each units in the composition but such units should be further detailed (and possibly refined). The abstract syntax defines the concepts of the language as well as theirs relations. In our experiments the abstract syntax has been modeled in Ecore [13]. The DSA contains the set of data needed to represent the state of a model at runtime (for instance the number of data in a FIFO at a specific instant). This execution state enriches the abstract syntax on which it is based. The DSA also specifies, by using an action language, the actions on the execution state (i.e., the execution functions). These functions specify how the execution state evolves when an execution function is invoked. In our implementations, we used KerMeta [10] for the definition of the DSA. It eases the definition of both execution state and functions by weaving them directly in the AS. The Model of Computation defines a (partial) order, which represents the schedule of the execution functions (i.e., when they are invoked). The MoC has been defined in CCSL [9] in our implementation because it provides the possibility to describe all the required relations between events independently of any AS or DSA elements. Finally, the links between the events and the invocation of the execution functions are defined by the DSE (Domain Specific Events). The DSE are used as a necessary and unique articulation point between the DSA and the MoC. The events from the MoC could be directly used but the DSE introduce more modularity with the possibility to change the abstraction level from the MoC to the DSA: one domain specific event can be produced by a, possibly timed, combination of MoC events (for instance by using complex event processing language [8] or CCSL itself).

Let illustrate the use of these different units on the Final State Machine (FSM) language. The *FSM* itself, *states*, *transitions* and *guards* are described in the AS because they are statically defined. However, information like The *current state* is only relevant at runtime and is defined in execution state of the DSA.

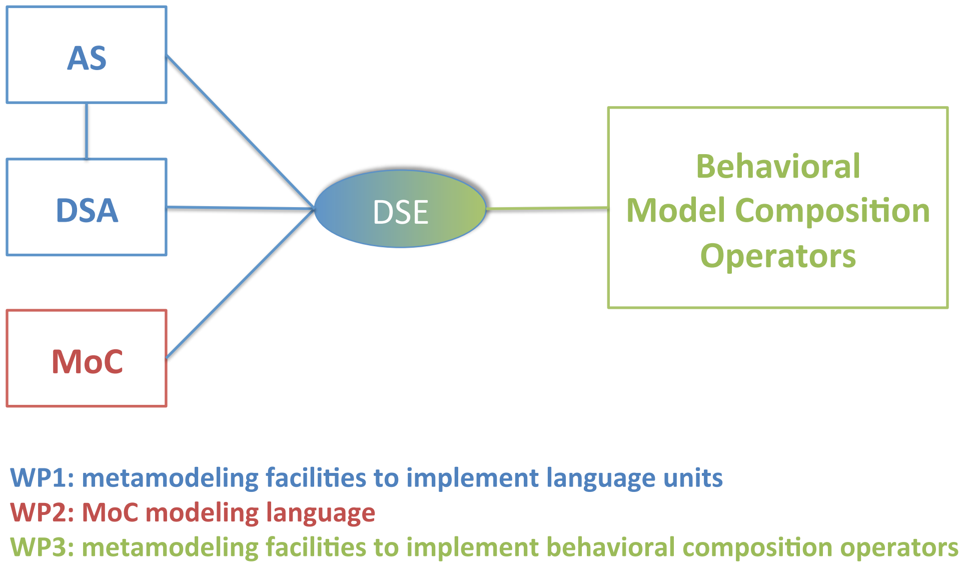
AS
DSA
MoC
DSE
Behavioral Model Composition Operators
WP1: metamodeling facilities to implement language units
WP2: MoC modeling language
WP3: metamodeling facilities to implement behavioral composition operators

Figure 2.1: the Modeling units in GeMoC

Also, functions like *fire()* manipulate the current state and possibly execute other actions. Such functions are defined as execution functions of the DSA. The MoCC specifies when the *fire()* function is triggered. For instance when the current state is the source of the transition and when the guard of the condition is true. One of its role among other is to specify what to do if several transitions can be fired.

Given this definition of a DSML, we can define two DSMLs to be heterogeneous if at least one unit differs from one to the other. This is a quite simple definition but it is sufficient in our case since we want to be able to compose highly heterogeneous, slightly heterogeneous and even homogeneous DSMLs in the same way. To achieve this goal we encapsulate models conforming to a DSML into a hierarchical component model. This component model exposes only event (from DSE) and data (from the AS or the DSA) that the model wants to share with other DSML(s). We describe this component model as well as its use for composition in the next section.

2.2 Heterogeneous Model Composition

2.2.1 Model Encapsulation

In order to reason about the composition of heterogeneous models, we used the component paradigm. A *Component* encapsulates an instance of a model conforming to a DSML (i.e., a tuple $< AS, DSA, DSE, MoC >$). A component can be hierarchical and exposes only the internal elements of its model to be used during composition with other (heterogeneous) models. From the composition point of view the component is a black box. It relies on *Port*s to expose internal data values from its AS or DSA and to expose *Event*s (from its internal DSE or from a combination of them). Only these elements can be used during the composition. Consequently, two kinds of ports are defined: *EventPort*s and *DataPort*s. An excerpt of the metamodel is presented in Figure 2.2. We voluntary kept the component model simple and we focus here on the ports, connectors and the concepts needed for the composition.

An *EventPort* is a way to expose one event (named *portEvent*) to other components. The *portEvent* is specified by a combination of one or several events from the DSE. This combination is specified in a *PortComplexEventProcessing* concept. A *PortComplexEventProcessing* describes how events(s) from the DSE are combined and related to the port event. It is used to expose the most appropriate event to the other components. For instance, if a component wants to expose an event that states the internal model has finished a computation, such event can simply refer to one ‘finished’ event in the DSE; but it can also

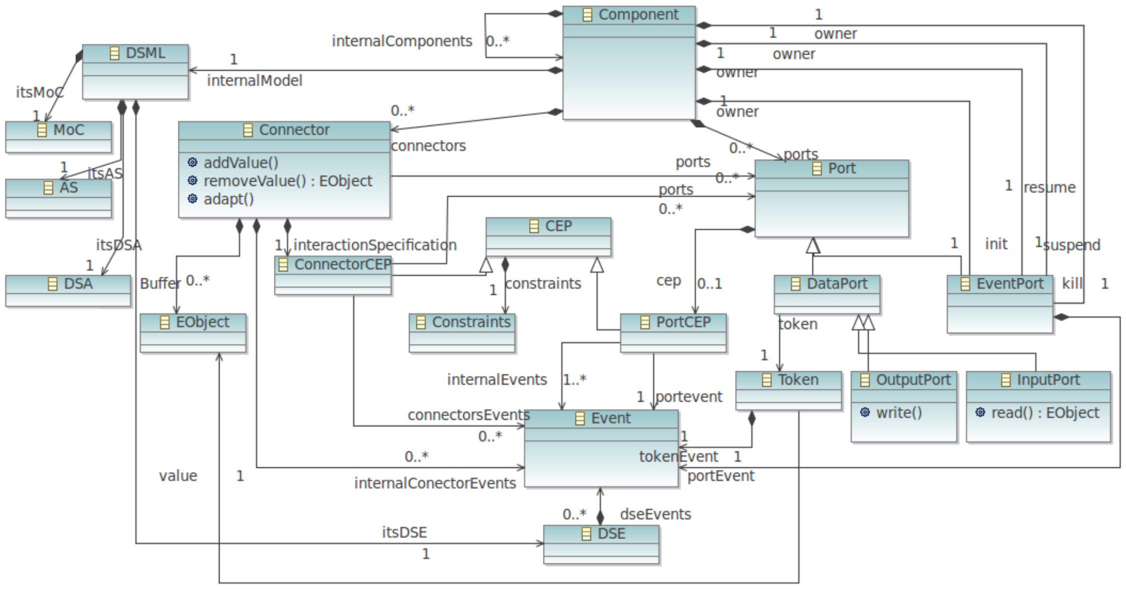

Figure 2.2: Excerpt of the Component metamodel

be defined as the last 'finished' event of internally concurrent DSML entities. One can imagine various ways to specify such combination. In our experiments we defined it by a set of *Constraints*, not detailed here but specified in the CCSL language [9]: a declarative and formal constraint language over events.

A *DataPort* is a way to expose an internal data value to other component. This data value comes either from the AS or from the DSA of the encapsulated model. Instead of exposing directly its internal data value, we use the concept of *Token*. A token is composed by an event and associated to an *EObject* representing a *value*. A token is a way to associate a data with an event specifying when it is updated. The event can then be used to enforce some synchronizations (*e.g.,* the data must be read each time it has been updated). Contrary to event ports, data port are oriented and we distinguish between *OutputDataPort* and *InputDataPort*. An output data port contains a method named *write(EObject newValue)*, which is invoked when the associated token event occurs. Similarly, an input data port contains a method named *read():EObject*, which is invoked when the associated token event occurs. The event from a token of an output data port indicates that the *value* of the token has changed and the event from a token of an input data port indicates that the *value* of the token has been read by the internal model. Like for event ports, the token event is obtained by a combination of one or several events from the DSE. This can be used to change the rhythm from the internal update to the one exposed to other components (for instance if the model is computing the fix point of a value, it is exposed only when this computing is finished).

Let illustrate the encapsulation of a model into a component in the figure 2.3. In this case a model conform to a TFSM language is used. The component model hides the details of the model and exposes only the needed element to the composition to other component. The example is illustrated in the Figure2. The encapsulation is achieved through ports and defining correctly the event relations stated by the PortComplexEventProcessing. In this example we are interested in expose the events sensor_fire, alarm0n, alarmOff and Clk. Notice that while event alarmOn is triggered by the model, the others are triggered by others component. In this way, these ones are used to fire transitions. In this example we are also interested in expose data. Then the value of variable i, defined in the AS is exposed. The event and data are exposed through ports. For each element to be exposed a port is created. Depending of the kind of the element either the event ports or data ports are used. In this way the event ports: portAlarmOn, portAlarmOff, portClk and portSensor are created. Theses will be used to expose events from the DSE. To expose data from the AS we create the port portCounter. We define it as an OutputPort so the port exposes its value but it is not possible to change it from other component. Once the ports are created, we have to specify how the portEvent in each port is related with the events in the DSE. In that way the PortComplexEventProcessing must be defined (Event Relation in the picture). Such kind of relation may be expressed in a language like CCSL. The event port portAlrmOn exposes the event AlarmOn. So the portEvent in the portAlrmOn is related with the event AlarmOn. In this case we define a coincident relation between the portEvent and the event AlarmOn. The coincident relation states that both events fire simultaneously in time. The same eventual relation are between the portEvent in portSensor and the event sensor_fire. But in this case the event port is triggered for another component. The Clk event is also triggered from outside. This event represent the notion of the clock for the TFSM. In this way, the event Clk is exposed through the port portClk. The eventual relation states that event Clk and portEvent fire simultaneously in time. The variable i is exposed through the data port portCounter. The eventual relations in this case states that tokenEvent fires in coincident with the changes of the value of i. When tokenEvent fires, the value of the data ports is updated to new value of i.

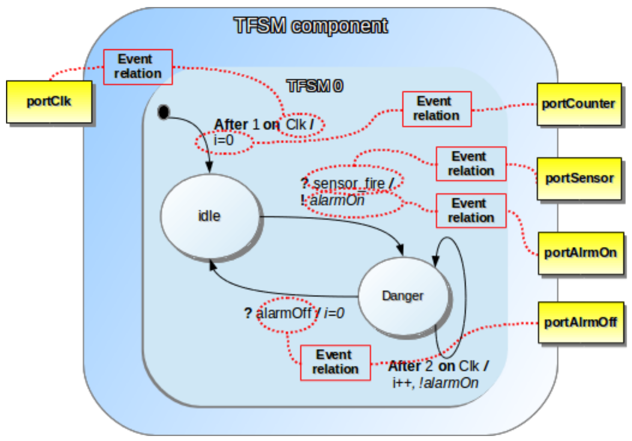

Figure 2.3: Model encapsuled

The component model hides the details of the DSML and only exposes the needed element for the composition. The component model does not perform the composition. Instead of that, it eases the composition only taking in account only the element required for the composition. Once the model is encapsulated, the designer has just to deal with component and ports. So it is needed to define how component communicates.

To connect ports we defined a *Connector*. A connector is used differently depending on the actual type of the ports it connects. We first describe what is true for every connection and then we detail specificity according to the ports involved in a connection. The glue of a connector, which specifies its behaviour, is specified by a *ConnectorComplexEventProcessing* named *interactionSpecification*. This interaction specification is associated to ports from which it constraints the events (either the events of an event port or the event of a data port token). These constraints are not defined by the MoC of a specific DSML since it specifies the adaptation between potentially heterogeneous models. It is the goal of an integration designer to specify them. In other words, an event connector, through its interaction specification, allows the designer to define how a port in a component is synchronized with other ports, independently of the encapsulated model. Here, the events are used as a common synchronization artifact between all languages, making their composition possible.

A connector has an interaction specification. However, this specification ensure different goals depending on the actual type of the ports involved in the connection. For instance, it is remarkable that event ports are not oriented. When a connector involves only event ports, we want to keep a maximum of flexibility in the definition of the connector interaction specification. This way, if the interaction specifies a causality between one port event and another one then it indirectly specifies a orientation. In opposite, if it specifies a simultaneity between the connected port events, it specifies a “rendez-vous” like (non oriented) synchronization[1].

[1] These are simple synchronization examples but the whole CCSL expressiveness is supported

Things are different when a connector involves only data ports. In this case, the connector is also in charge of storing (in its *buffer*) and adapting (by using the *adapt()* method) the values of the data port tokens. To specify the "storing policy" of the connector, a connector has two methods: *addValue(EObject newValue)* and *removeValue(): EObject*. The interaction specification is then specifying constraints between events from ports and, if needed, between internal events to specify when its internal methods are invoked. For instance, a typical interaction specification between one output data port (*outP*) and one input data port (*inP*) can state that:

- when *write(newValue)* is invoked (*i.e.,* when the token event of the output data port occurs), the token value is added in the connector buffer by invoking *addValue(outP.token.value)*; and
- when *read()* is invoked (*i.e.,* the token event of the input data port occurs), one value from the buffer is retrieved by invoking the *removeValue()* method and the internal token value of *inP* is updated.

Note that the specification of addValue and removeValue methods can be different according to the storing policy desired (*e.g.,* FIFO, data overwriting to keep only the fresher one). However, the constraint over the associated events must be consistent with the storing policy (*e.g.,* enforcing that a data has been written before to be read). Such data interaction allows us (1) to implement many kind of communication schemes like Buffered, Broadcast, routed, pushed pulled, and (2) to realize some value adaptations by invoking the *adapt()* method (*e.g.,* converting from Real into Integer)

In addition to connections involving only event ports or only data ports, connections involving both are possible. For instance this could be the case if we want to connect an FSM component with an SDF component. In such case, one could want to connect an output data port from the SDF component to an event port from the FSM component. In that situation the FSM port event can be related to the event of the SDF port token; however, the value of the port token is not stored and ignored. The conjugated example considers a connection between an event port and an input data port. In this case, the interaction specification could specify that an event occurrence from an event port invokes the adapt() method, which creates a new data value (like an integer created according to event occurrence number) and this value can be stored for the data input port to read it. Additionally, the interaction specification can be used to synchronize the event port with the reading of the data (*e.g.,* the data is read every 3 times the port event occurs).

In the definition of the component model we tried to be as general as possible to be able to express various compositions. We based our component model on the fact that events are the core elements from which synchronization between component is made. Data have been reconciled to events through the explicit notion of token. Based on this, we identified that there exist two main families of compositions, making different use of the events: the flat composition and the hierarchical composition. These composition are described in the next subsections..

2.2.2 Types of Composition

So far, we are able to encapsulate a DSML in a component. The component is seen as a black box which exposes data (from AS or DSA) and events (from its internal DSE or from a combination of them) through ports. A complex system can be modeled as an assembly of components; we call such assembly a *Flat Composition*. The flat composition consists in setting correctly the interaction specifications in the connectors. This defines how components communicate / are synchronized each others. Each model encapsulated in a component has a (possibly infinite) set of possible execution paths; and the interaction specification is a set of constraints between the events exposed by ports. Consequently, when a component is connected to another one, its set of possible execution paths is either the same or a subset of the original set. The resulting behaviour of connected components is a subset of the union of the initial behaviours plus the behaviour of the interaction specifications. For instance, if the interaction specification is set correctly between two FSM components, it can specify a synchronous product of the state machines.

Let illustrate the flat composition on a simple example represented Figure 2.4. The system is composed by two heterogeneous components. The first one, called *TFSM* encapsulates a simple Timed Final State Machine model. This state machine waits 1 after its initialization, initializes a variable *i* to 0 and then goes in the state *S1*. From the entering in this state it waits 10 and then emits an event called *ping*. Then it enters the state *S2* and waits for the reception of an event called *pong*. When pong is received it increments *i* and returns in state S1. In the TSFM component, two event ports are exhibited: *p1* and *p2*. On the picture there is a link with an equality symbol that connects *p1* with the internal event *ping*. This link represents the complex event processing element that link the event in the port *p1* with the DSE event that states when the *ping* event occurs. The symbol simply specifies here that these events are simultaneous[2]. The same kind of link is realized between *p2* and *pong*.

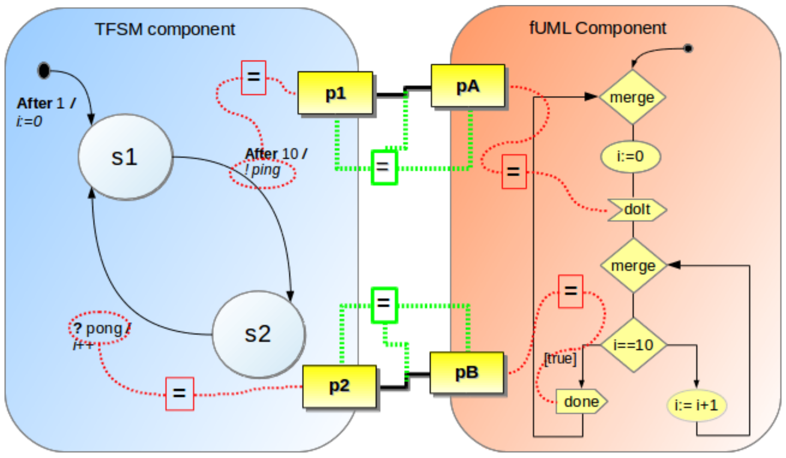

Figure 2.4: Simple Flat Composition

The second component, called *fUML* encapsulates a simple UML activity model following the fUML semantics. When it is initialized, an action that initializes a variable i to 0 is executed after going through a merge node. Then the activity waits for the signal *doIt*. When the signal is received, it goes to a decision node (through a merge node) to test if *i* is equal to 10. If not, it increments *i* and goes to the test again. If true, then it sends the signal *done* and goes back to the waiting of signal *doIt*. In the fUML component, two event ports are exhibited: *pA* and *pB*. The links between the ports and the internal language elements have the same meaning than in the first component. It means that there is a complex event processing between the event from the port and the event from the DSE that invokes the execution function of the linked element.

Finally, two connectors are defined to specify that the connected port events are simultaneous (*i.e.,* it is a "rendez-vous" connector). From this composition results a "classical" ping pong exchange between heterogeneous components. Note that this composition is arbitrary and simple but it highlights the mechanism introduced in a flat composition. Both the way to expose internal events in ports and to connect ports together impact the way the behaviours of the system are composed.

An interesting question that arises from this example is: "when are the internal model of the components initialized ?" One could answer "when the whole system starts". However the answer should be explicit, a forciori if we consider the second composition family: the *Hierarchical Composition*. The hierarchical composition is a composition where a component, or an assembly of components, refines a specific model element of a DSML. For instance, the state *S1* in the *TFSM* component could be refined by a model encapsulated in a sub-component. Moreover, the language of the model in the sub-component could be different from the one of its container. In the example Figure 2.5, a state *S0* of a TFSM model is refined by an activity diagram that follows the fUML semantics. The same question than from the flat composition arises: "when is the internal model of *S0* activated ?". Some other questions can be interesting too: "What happens when the internal model finishes ?" or "if the *exitS0* event occurs before the end of the internal model, do we wait the internal model to finish ? do we stop it ? do we suspend it to resume it next time ?".

These questions highlight the problem of the control delegation from a sub-component to the model

[2]The reader should note the difference between the concept of Event in the DSE and the DSML notion of Event, which does not exist in all DSMLs

element that contains it. Instead of making some arbitrary choices (like often imposed by the implementation of existing tools), we want to answer all these questions by "It depends on the designer desires !". In other words we want to provide a framework where the designer is able to explicit the (possibly complex) rules that define the control delegation.

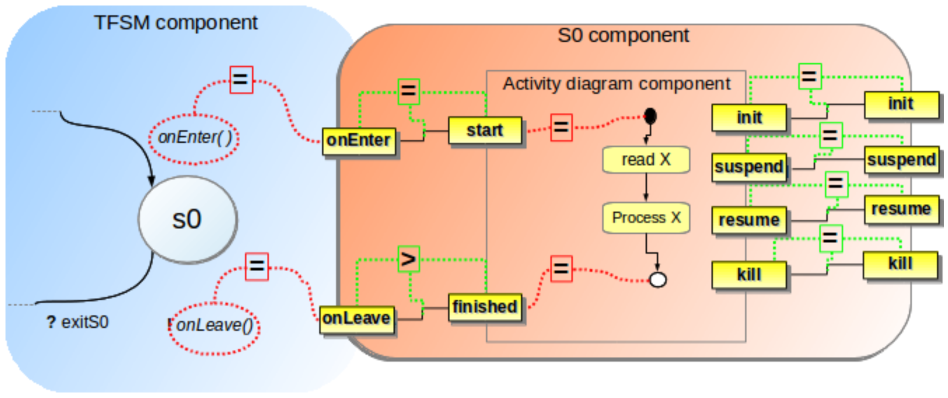

Figure 2.5: Hierarchy composition

To enable an explicit specification of the control delegation, a component has to contain four specific event ports: *suspend*, *resume*, *kill* and *init*. These control ports represent an interface to delegate the control from a sub-component to its model element. The control ports have a predefined behaviour, which affect the encapsulated model, consequently they do not have any *cep*. The init port activation (*i.e.,* when its port event occurs) initializes the internal structures of the model in memory in order to be ready to start the execution. From this instant, if some MoC or DSE events are possible, they can occur. It emerges that for every component, the init port should be activated before to expect any behaviour. The *suspend* port, when it is activated, avoid any DSE of the internal model to occur. Conjointly, The *resume* port re-authorize the DSE to occurs when it is activated. The *kill* port is more radical since it removes the model from the memory when activated. No behaviour from this model will then append until the *init* is activated.

In Figure 2.6 the component is represented with a FSM model. This allow us to understand better the meaning of the control ports. Thus, it is possible to identify the possibles transitions depending the current state of the component.

In addition to these specific port, the internal model usually needs to be synchronized with its container. For instance in the example Figure 2.5, the activity model needs to know when to start and to finish. Such event ports can be added as any other ports in a flat composition. It means that the associated events are specified (in a complex event processing) by a combination of one or several events from the DSE of the sub-component model. For instance in the example, the start *portEvent* may be simultaneous with the DSE that triggers the entering in the initial state while the finished port event may be simultaneous with the leaving of the final state. Such choices are left open to the designer.

Once the useful ports are defined (*i.e.,* connected to the internal events when needed), the designer must specify how these ports are related to the events from the container of the sub-component model (*i.e., S0* in our case). In a general case, the container model element is not a component since it relies on the abstract syntax of the model it belongs to. In the example Figure 2.5 it is a *State*. To be able to correctly connect it to its sub-component(s), we state that any model element can be represented by a component from our metamodel. Such component only exposes some events according to the execution functions of its DSML. More precisely, it exposes the events from the DSE which are either associated to an execution function of its owned structural feature or the DSE events associated to one of its execution function (weaved to it by the DSA, cf. section 2.3). Additionally it also exposes its control ports, as shown in Figure 2.5. A state in the TFSM DSML has two execution functions weaved to it: *onEnter()* and *onLeave()*. Consequently, the *S0* component exposes its four control ports plus two ports corresponding to the two DSE events that invoke the two functions weaved to it. From a component port to its sub-component port(s) it is possible to create

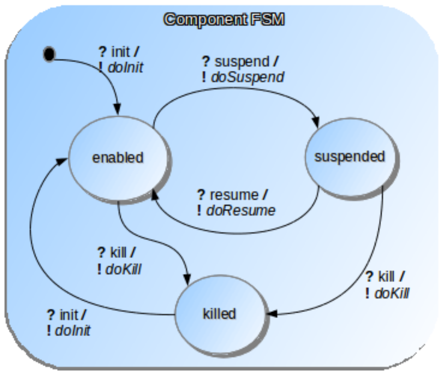

Figure 2.6: Control port FSM description

connectors with their possibly complex adaptation as explained previously. In our example the interaction specification of each control port connectors specifies a simultaneity between the connected port events. Consequently, all control ports are delegated to the model element (that should them-self be delegated to the TFSM component, etc). The *start* port of the sub-component is linked to the *onEnter* port and the interaction specifies a simultaneity: when the state S0 is reached, the *portEvent* in port *start* occurs, thus performing the start of the activity model. the *finished* port of sub-component is linked to the *onLeave* port and the interaction specifies a precedence: when the state *S0* is left, the *portEvent* in port *finished* should have already occurred. In other words, the internal activity model is started as soon as the TFSM model enters in the state *S0* and the internal activity model must be finished before to leave the state *S0*. Once again many different choices could be done there. For instance, another system could specify that *onEnter* is linked to the *init* port of the sub-component and *onLeave* to its *kill* port. The resulting behaviour highly depends on such delegation specification. In this framework both the hierarchical and the flat composition are specified by constraints over the respective evolution of the different models. So the compositions are always restrictions on the union of all possible execution paths taken individually.

Finally, the control port can also be used in the flat composition and it is now possible to answer the first question we asked: “when are the internal model of the components initialized ?”: when the event of the *init* control port occurs. Note that the activation of the control port of the top level component may be done by the simulation/execution framework.

2.2.3 Limits

Our semantic aware component model for heterogeneous composition of executable DSML focuses on the mandatory concepts for an explicit composition. It provides flexibility at different level: when creating the events/data to expose to the other DSML but also when connecting ports, in a hierarchical or not manner. This flexibility can appear to be cumbersome to deal with for an integration designer. The tooling of such component model should help the integration. This can be done by creating ”by default” connectors that can be changed as needed. It could also be helpful to provide adaptation on the shelf to avoid repetitively specifying the same adaptation. However, we believe it is more important to ask the integration designer to express its desires than to try to help him with imposed built-in choices.

Note that a from our experiments and feedback, making explicit the composition avoids designers to distort their models in order to use them with imposed compositions. However, the component model itself does not help the integration designer to make better choices and the composition can lead to a deadlock. For this reason we based our composition (and the MoC of a specific DSML) on a formal language (CCSL). The analysis of the composed execution paths could then helps the designer with early detection of undesired properties (*e.g.,* deadlocks). Another point that could be improved in this approach is the definition of the control port delegation interface. From our experiments we identified four control ports but more experiments should be driven in order to be sure it fits a maximum of case.

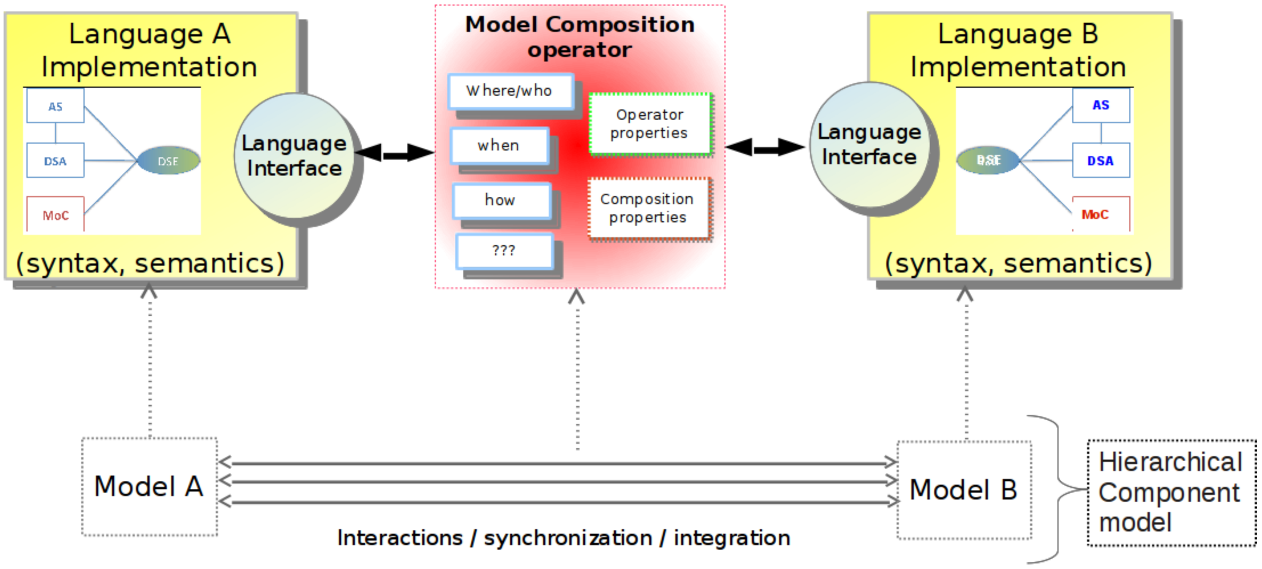

Figure 2.7: Ideas toward language behavior operators

2.3 Towards language behavior composition operator

In this document we present the heterogeneous component model that supports the coordination of heterogeneous models. It specifies the mechanisms needed at the model level to realize the coordination. The described mechanisms seem to be very expressive according to our experiments. However, it can be painful to specify all the low level coordination between each models. What is expect in the following is a reification of useful behavioral composition operator at the language level (see Figure 2.7). Such operators are expected to be described on a specific part of the language, *i.e.,* on a *language interface*. The application of the composition operator on two or more models should generate the coordination/integration/synchronizations between the models as described in the previous sections. An important question is what a behavioral composition is made up with. We were mainly inspired by three very different papers [4, 7, 5] but many other works were inspirational and a more detailed description of related will be made in a future deliverable.

We will detail the first attempt to define the elements of a composition operator by using a well know example: *the synchronous product automata* [1]. This operator is using the name of events and realizes a strong synchronization in between them. While this operator is expressed in a single language, we want to apply it in between two languages. To do so we need to express, based on the language automata, where the composition operator must apply (or in other words which are the elements of the language interface to be used). Considering that the DSE as identified in GeMoC are part of the language automata, the synchronous product *where/who* specification can be:

$$\forall dse1 \in languageA, dse2 \in languageB : dse1.name = dse2.name \tag{2.1}$$

In this equation we extract the DSE from both languages that have the same name. Based on this, we need to specific under which dynamic condition the coordination must be done. It is a generalization of what is named *Control-Flow Based Crosscutting* in AspectJ [7]. In the synchronous product operator, the coordination is always done independently of a dynamic pattern matching (*always*), however, one could what to specify that the synchronous product is made only every two occurrences of the events. This is what we naed *When* in Figure 2.7. The *How* part of a behavioral composition operator specifies what kind of coordination is made. In the synchronous product, it is a strong synchronization (a Coincidence ($=$) in CCSL). So, our synchronous product operator could look like:

$$\begin{aligned}
&\forall\ dse1 \in languageA,\\
&\quad dse2 \in languageB\\
&\qquad\qquad \text{where } dse1.name = dse2.name\\
&\qquad\qquad \text{when } always\\
&\qquad\qquad\quad \text{how } dse1 \boxed{=} dse2
\end{aligned} \tag{2.2}$$

Of course, because this is very preliminary work, we are not sure this is sufficient and the experiments and related work will help to characterize the element needed for the expression of behavioral composition operators.

We also identified that a behavioral composition operator can be characterize by the way it can be used, *i.e.,* by its own properties like for instance associativity, commutativity, etc [4, 5]. Finally, one important characteristic of such operator is its impact on the composition itself. Such characteristics specify the impact of the coordination on the existing properties of each language [4]. Some examples of these characteristics are:

- deadlock-freedom: if model A and model B are deadlock-free, the application of the composition operator results in a deadlock-free behavior.
- safety-secure: if a safety property is true on model A, the application of the composition operator preserves the property

All such points provide a clear and details road-map for the future works that will be made in WP3.

2.4 Conclusion

We proposed in this document a component model, which encapsulates DSML models. Each DSML must be defined in a way that ensure the possibility to compose it in a semantic aware way. Consequently a DSML is defined as a tuple composed of the Abstract Syntax, the Domain Specific Actions (execution state + execution functions), the MoC (Model of Computation) and Domain Specific Events (a possibly complex mapping between events from the MoC and execution functions). From this kind of DSML, each component exposes events (from the DSE) and data (from the AS or the DSA) through ports. It is then possible to create a system by specifying flat and hierarchical compositions of components. The flat composition is made by connecting port of components and explicitly specifying the constraints between the events in the connected ports. The hierarchical composition considers any model element as a component, which exhibits two kind of ports to be used by sub-components: control ports that provide predefined constraints over the model and event ports that represent the DSE events that invoke the model element execution functions. From such approach we were able to experiment composition of different DSMLs but we still lake adequate tooling to make the approach usable.

However, this component model is just a first stone required to be able to specify behavioral composition operators; defined on languages and applied on models. This future works have been well detailed in section 2.3 and represent the road-map of WP3.

3. References

[1] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[2] 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.

[3] 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.

[4] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. In *Formal Methods for Components and Objects*, pages 443–466. Springer, 2003.

[5] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An algebraic view on the semantics of model composition. In *Model Driven Architecture-Foundations and Applications*, pages 99–113. Springer, 2007.

[6] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[8] David C. Luckham and Brian Frasca. Complex event processing in distributed systems. Technical report, Stanford University, 1998.

[9] Frdric Mallet, Julien DeAntoni, Charles Andr, and Robert de Simone. The clock constraint specification language for building timed causality models. *Innovations in Systems and Software Engineering*, 6:99–106, 2010.

[10] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *MoDELS*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

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

[12] Ingo Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, April 2003.

[13] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, 2008.