Reifying Concurrency for Executable Metamodeling ⋆

Benoit Combemale[1], Julien De Antoni[2], Matias Vara Larsen[2], Frédéric Mallet[2], Olivier Barais[1], Benoit Baudry[1], Robert B. France[3]

[1] University of Rennes 1, IRISA, Inria
[2] Univ. Nice Sophia Antipolis, CNRS, I3S, Inria
[3] Colorado State University

Abstract. Current metamodeling techniques can be used to specify the syntax and semantics of domain specific modeling languages (DSMLs). Still, there is little support for explicitly specifying concurrency semantics of DSMLs. Often, such semantics are provided by the implicit concurrency model of the execution environment supported by the language workbench used to implement the DSMLs. The lack of an explicit concurrency model has several drawbacks: it prevents from developing a complete understanding of the DSML's behavioral semantics, as well as effective concurrency-aware analysis techniques, and explicit models of semantic variants. This work reifies concurrency as a metamodeling facility, leveraging formalization work from the concurrency theory and models of computation (MoC) community. The essential contribution of this paper is a language workbench for binding domain-specific concepts and models of computation through an explicit event structure at the metamodel level. We present a case study that serves to demonstrate the utility of the novel metamodeling facilities and clarify the scope of the approach.

1 Introduction

In a context where software-intensive systems must handle an increasing number of issues in diverse domains, for example, issues related to providing functional features and qualitative guarantees, and to supporting heterogeneous hardware platforms, the use of domain-specific modeling languages (DSMLs) can result in increased productivity while providing effective support for separating concerns. DSMLs can make it easier for stakeholders from different domains (*e.g.*, experts in fault tolerance, security, communication) to participate in the design of a system, by providing linguistic concepts tailored to their specific needs. However, for a DSML to be an effective system design tool, it must be defined as precisely as possible and supported by sound analysis tools [1].

The specification, design and tooling of DSMLs leverage the rich state of the art in language theory. Several metamodeling environments support the specification of the syntax and the (static and dynamic) semantics of a DSML. These two elements

⋆ This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011), and the CNRS PICS Project MBSAR.

of a DSML specify the domain-specific concepts, as well as the meanings of domain-specific actions that manipulate these concepts. Examples of metamodeling environments include Microsoft's DSL tools [4], Eclipse Modeling Framework (EMF) [5], Generic Modeling Environment (GME) [6], and MetaEdit+ [7]. A significant limitation of current metamodeling environments is the lack of support for explicitly modeling concurrency semantics. Concurrency semantics is currently defined implicitly in DSMLs that support concurrent execution in their models. It is typically embedded in the underlying execution environment supported by the language workbench used to implement the DSMLs (*e.g.*, if the language runs on top of a Java Virtual Machine, the semantics of Java threads defines concurrent behavior).

The lack of an explicit concurrency model has several drawbacks. It not only hinders a comprehensive understanding of the behavioral semantics, it also prevents developing effective concurrency-aware analysis techniques. For instance, knowing that a data-flow model (*e.g.*, an activity diagram) follows Kahn process networks semantics ensures *de-facto* properties like latency-insensitive functional determinism but imposes communications through unbounded FIFOs. Restricting the data-flow model to the Synchronous Data Flows semantics allows the computation of finite bounds on the communication buffer sizes. Furthermore, having an implicit concurrency model also prevents the distinction of semantic variants in a model. For example, the fUML specification identifies several semantic variation points. As stated in the fUML specification, some semantic areas "are not explicitly constrained by the execution model: The semantics of time, the semantics of concurrency, and the semantics of inter-object communications mechanism" [2]. The lack of an explicit model of concurrency, including time and communication, prevents one from understanding the impact of these variation points on the execution of a conforming model.

In previous work, we developed an approach that bridges the gap between models of computation and DSMLs [3]. In this paper we use that work as the base for reifying concurrency as a metamodeling facility. We leverage formalization work on concurrency and time from concurrency theory, specifically, theoretical work on tagged structures [4] and on heterogeneous composition of models of computation [5,6]. The primary contribution of this paper is an approach supported by a language workbench for binding domain specific concepts and models of computation through an explicit event structure at the metamodel level. We illustrate these novel metamodeling facilities by designing a DSML specifying concurrent and timed finite state machines. We highlight the benefits and the flexibility of the approach by making a semantic variation on the concurrency specification of the DSML. We also provide pointers to other examples to show that our approach applies to different MoCs and DSMLs.

The paper is organized as follows. Section 2 uses background on language and concurrency theories to identify the key ingredients of a concurrency-aware executable DSML, and to reify them as the association of four language units. Section 3 describes the language workbench built to implement the proposal, and the associated

[4] http://www.microsoft.com/en-us/download/details.aspx?id=2379
[5] http://www.eclipse.org/modeling/emf/
[6] http://www.isis.vanderbilt.edu/Projects/gme/
[7] http://www.metacase.com/mep/

environment for concurrent model execution. Section 4 demonstrates and discusses the DSML implementation and execution environment obtained thanks to our language workbench. The approach is illustrated throughout the paper with the design, implementation and use of timed finite state machines. A comparison to related work and a conclusion follow.

2 Ingredients of a Concurrency-Aware Executable DSML

2.1 Background Knowledge

Current metamodeling environments support defining a modeling language through the specification of the concrete and the abstract syntaxes as well as the mapping from the syntactic domain to the semantic domain. Over the last 50 years, the language theory community has studied the mapping between the syntactic domain and the semantic domain extensively. This has led to three primary ways of defining semantics: *operational semantics*, where a virtual machine uses guard(s) on the execution state to drive the evolution of the models expressed in the language [7,8,9,10]; *axiomatic semantics*, where predicates on the execution state allow reasoning about the models expressed in the language and its correct evolution [11,12,13]; and *translational semantics* [14] that defines an exogenous transformation from the syntactic domain to an existing language (either an existing computer language or a mathematical denotation, *i.e.,* a denotational semantics [15]). A drawback of such approaches is that none of them supports the specification of concurrency in a manner that would allow systematic reasoning (chapter 14 of [13]). Even if these approaches could support the definition of concurrency, the concurrency model would be scattered through the semantic specification, making it difficult to understand and analyze the properties related to concurrency (*e.g.*, deadlock freeness, determinism).

In most language implementations, the concurrency semantics is implicitly embedded in the underlying execution environment used to execute the conforming models. For instance, some executable models supporting concurrent execution rely on the Java concurrent model. On one hand, the concurrency of the model depends on the Java concurrency and on the other hand it does not guarantee similar execution/analysis on platforms with different parallelism possibilities (*e.g.*, single core vs. many cores, processor arrays).

Work on formal and explicit models of concurrency has been the focus of some research programs since the fifties. Early work in this area resulted in three well-known contemporary approaches: CCS [16], CSP [17] and Petri Nets [18]. Unlike the approaches from language theory, these solutions focus on concurrency, synchronizations and the, possibly timed, causalities between actions. In these approaches, the focus is on concurrency and, thus, the actions are opaque and abstract away details on data manipulations and sequential control aspects of the system. Such models have proven useful for reasoning about concurrent behavior, but they are not tailored to support the description of a *domain-specific* modeling language dedicated to a domain expert. After many years, work on models of concurrency has consolidated, from an analytical point of view, into two different approaches, namely, event structures [19] and tagged structures [4]. In these approaches the non-relevant parts of a model are abstracted away

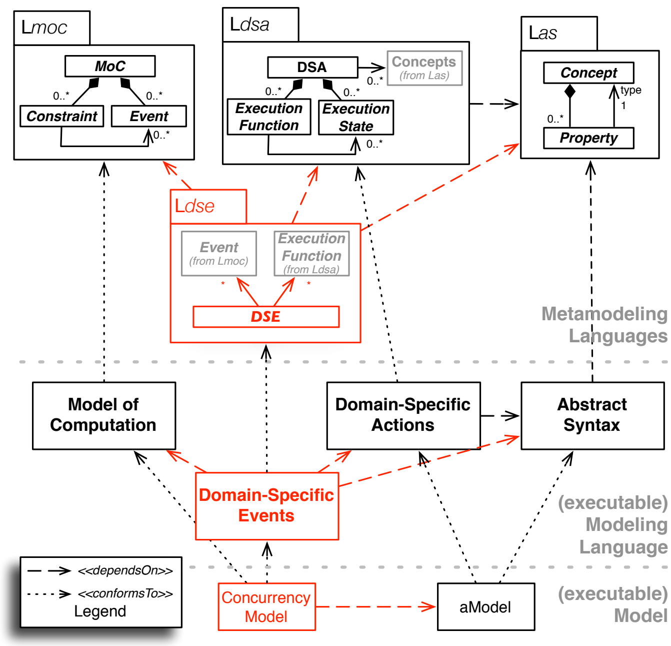

Fig. 1. Modular Design of a Concurrency-Aware Executable Modeling Language

into *events* (also named signal) and the focus is on how such events are related to each other through causality, timed or synchronization relationships. Both event structures and tagged structures have been used to formally specify or compare concurrency models underlying system models expressed in modeling languages. These concurrency models and can be viewed as the concurrent specification of a specific system model. However, such approaches are not related to the computational part of a model and have not been used to specify the concurrency semantics of a language.

2.2 Language Units Identification

Taking a step back from these seminal approaches, we explicitly identify the common language units that constitute the design and implementation of an executable concurrency-aware modeling language (see middle level of Fig. 1). Each language unit is independent of the way it is implemented, and directly benefits from language and concurrency theories described above.

Language Unit #1 The first language unit is the description of the language *abstract syntax* (see Fig. 1). Older approaches build the semantics of the language on top of the concrete syntax but the benefits of using the abstract syntax as a foundation for language reasoning (first introduced in [20]) have been well understood since the 1960s. In the MDE community, the abstract syntax is a first class part of a language definition. The abstract syntax specifies the syntactic domain and is used to anchor the semantics. It is however important to avoid blurring the syntactic domain with language elements that represent the execution state of the model.

Definition 1 *The Abstract Syntax (*AS*) specifies the concepts of the language and their relationships. An instance of the* AS *is a model.*

Consequently, a meta-language for modeling AS (*Las* in Fig. 1) must provide facilities to define the language concepts ($Concept$) and the relationships between them ($Property$).

Language Unit #2 The second language unit, called *Domain Specific Actions* (see Fig. 1), adds new properties that represent the *execution state* of a model and a set of *execution function*s that operate on these properties during the execution of a model.

The execution state can be represented, for example, by the *current state* in a Finite State Machine (FSM). It can also be specified independently of the abstract syntax, as in, for example, the incidence matrix that encodes the state of a Petri net. Such information is needed to specify the state of a model during its execution but is not needed to specify the model's static structure. It is consequently part of the semantic domain.

The DSA is also composed of *execution functions* that specify how the execution state sequentially evolves during the model execution. For instance, when a transition is fired in a FSM, the current state is updated. This is one of the roles of the execution functions. They also specify how the concepts of a language behave. For instance if the language contains a *Plus* concept, then an execution function must specify how the *Plus* instances actually behave during the model execution.

Definition 2 *The Domain Specific Actions (*DSA*) represent both the execution state and the execution functions of a DSML. An instance of the* DSA *represent the state of a specific model during the execution and the functions to manipulate such a state.*

No hypothesis is made on how to specify the DSA (*Ldsa* in Fig. 1). However, the specification of the DSA depend on the AS since it describes a part of its semantic domain. The execution state would be defined with structural properties representing the semantic domain, in the same way *Las* supports the definition of the syntactic domain. The execution functions can be specified in very concrete terms (*e.g.*, operational semantics that uses an action language to specify rewriting rules), or in more abstract terms (*e.g.*, denotational semantics that provides functions specifying the execution functions). The latter approach only denotes mathematical properties about the result, and does not specify any details on how to implement the resulting functions. This is even more abstract in an axiomatic semantics, where pre/post conditions on the execution state of the system are specified and all the functions that respect such conditions are considered as correct execution functions.

Note that the global ordering of the execution functions is not specified in the DSA since it can be concurrent (and timed). This is the role of the third language unit.

Language Unit #3 Concurrency theory has proposed many approaches, but roughly speaking a concurrency model is a way to specify how different events are causally and temporally related during an execution (in our case, the execution of a model conforming to a DSML). These ideas have been used in the notion of Model of Computation (MOC) [6,21,5]. All definitions of MOCs share the fact that a MOC acts as a director for some pieces of code. The MOC is then acting as an explicit concurrency pattern, which provides MOC-dependent analysis properties. The third language unit is then called *Model of Computation* (see Fig. 1) and explicitly specifies the concurrency.

Definition 3 *The Model of Computation (*MOC*) represents the concurrency aspects in a language, including the synchronizations and the, possibly timed, causality relationships between the execution functions. An instance of a* MOC *is defined for a specific model, conforming to the DSML. It is the part of the* concurrency model *that specifies the possible partial orderings between the events instantiated with regards to the model.*

A meta-language for modeling MOC (*Lmoc* in Fig. 1) would allow the definition of events and the specification of causal relationships (and synchronizations) such as scheduling, temporal constraints, and communications. The events can be discrete (*i.e.,* a discrete event is a possibly infinite sequence of occurrences), or dense (*i.e.,* a dense event is an infinite set of occurrences and there are an infinity of occurrences between any two event occurrences in the set). *Lmoc* must be independent of a specific AS or DSA.

2.3 Reifying Language Units Coordination

In our approach, all language units previously presented are specified separately (see middle level of Fig. 1). This separation benefits modularity, reuse and the identification of the concurrency related analyses supported by the language. The modeling units must then be consistently coordinate to provide an executable modeling language with reified concurrency. This coordination has to keep the language units separated while providing a natural articulation between them.

The AS and the DSA are kept separated to support several implementations of the DSA for a single AS (to deal with semantic variation points, or with semantics for different purposes, *e.g.*, interpreter or compiler). There exists a mapping between the DSA and the AS, however the DSA is dedicated to a specific AS (see dependency between AS and DSA in Fig. 1), and both AS and DSA are dedicated to the DSML under design. Consequently, we did not reify this mapping. The mapping is more conveniently described directly in the DSA.

The definition of the DSML behavioral semantics then consists in specifying the coordination of a given MOC with the DSA. This coordination must keep the MOC and DSA independent to enable the (re)use of a MOC on different AS/DSA or changing the MOCs on a single AS/DSA. Hence, the coordination specification can be put neither directly in the MOC nor in the DSA. For this reason, we reify the binding as a proper language unit that bridges the gap between the MOC and the DSA. This is done through the notion of *Domain Specific Event*, a novel metamodeling facility that we propose to reify.

Language Unit #4 The *Domain Specific Events* (DSE, see Fig. 1) specify the coordination between the events from the MOC and the execution function calls from the DSA. The DSE depend on both the MOC and the DSA. This coordination contains four parts:

DSE → DSA The DSE specify events that are associated with one or more execution functions. When such an event occurs, it results in the call of the associated execution functions. The meta language for modeling DSE (*Ldse* on Fig. 1) has to make some choices about how much associated functions can be associated with an event (*e.g.*, single one, any) and if several functions are associated with a single event, it must specify how these calls must be done (*e.g.*, in sequence, in parallel).

MOC → DSE The MOC events can be specified at a abstraction level different than the execution functions from the DSA. For this reason, the DSE specify how the defined events are obtained from the ones constrained by the MOC. This specification can be, for example, the filtering of occurrences from an event or the detection of an occurrence pattern from various events. It can also be the observation of some dense events from the MOC. In this case the DSE are used to specify the relevant observations on the dense event from the MOC and, in such a way, they specify the events that can be observed by looking at the execution of the conforming models. Such an adaptation between the low level events from the MOC and the ones in the DSE can be arbitrarily complex (ranging from a simple mapping to a complex event processing). However, when *Ldse* allows adaptations more complex than a simple mapping, one must ensure that the adaptation is not breaking any concurrency-related assumptions from the MOC.

DSA → DSE The MOC and the DSE represent the specification, at the language level of the concurrency model (dedicated to a specific model conforming to the DSML). This concurrency model specifies the acceptable partial orderings of both the events constrained by the MOC and the ones from the DSE. During a specific execution, the call to some execution functions can restrict such partial orderings. For instance, if the DSML specifies a conditional concept (*e.g.*, *if-then-else*), a MOC usually specifies that going through the *then* branch or through the *else* branch depends on the evaluation of the condition (*i.e.,*the condition evaluation causes either the *then* or the *else* branch, exclusively). Both paths are specified in the concurrency model as acceptable but the actual path taken during an execution depends on the result of the call to an execution function. The specification of the feedback from the execution function calls to the execution engine of the concurrency model must be specified in the DSE.

MOC ← DSE → AS Finally , the DSE must specify how the MOC is applied on a specific model that conforms to the DSML (*i.e.,* how to create the concurrency model according to the MOC constraints and the AS concepts). Depending on the language used for the MOC modeling, this specification can be of a different nature, however it requires the capacity to query the AS to retrieve the parameters needed for the creation of the concurrency model. For instance, in a FSM the DSE can specify that a specific constraint must be instantiated for all the *Transition* instances in the model. Also, it can retrieve the actual parameter of the constraint by querying the AS. Once again, depending on the possibility offered by *Ldse*, one must ensure the preservation of the MOC assumptions (*e.g.*, by using proven compilers or a language supporting clear and simple composition of constraints from the MOC).

Definition 4 *The Domain Specific Events (*DSE*) represent a coordination between the* MOC *and the* DSA *to establish the concurrency-aware semantic domain. It is composed of a set of domain specific events, a mapping between these events and the execution functions from the* DSA*, a possibly complex mapping between the events constrained by the* MOC *and the domain specific events; the specification of the impact of the execution function results in the execution of the concurrency model and finally the specification of the* MOC *application on a specific model that conforms to the DSML.*

As highlighted by the previous description, the coordination between the MOC and the DSA (*i.e.,*the DSE) is a key point to enable concurrency-aware semantic domain. However, this coordination is often implicit or hard coded. We believe that its reification enables effective use of a language that includes concurrency and computational aspects. In this section, we have identified the key ingredients for designing a concurrency-aware executable DSML that leads to the architectural pattern proposed in Figure 1. Consequently, we consider in this paper the following definition for a concurrency-aware executable DSML:

Definition 5 *A* concurrency-aware executable DSML *is a domain-specific modeling language whose conforming models are executable according to an explicit concurrency model. Its definition includes at least the abstract syntax and the behavioral semantics (including the* DSA*, the* MOC *and the* DSE *to coordinate them). In the context of this paper, a concurrency-aware executable DSML (xDSML) is defined as a tuple* $\langle \mathrm{AS}, \mathrm{DSA}, \mathrm{MOC}, \mathrm{DSE} \rangle$.

3 A Language Workbench to Design and Implement Concurrency-Aware Executable DSMLs

The reification of concurrency for executable metamodeling has been presented in its general form and several implementations of it can be realized. In this section we present the actual implementation of our language workbench that was used to validate the proposition. We have tried to take the most adequate language/technology for each language unit so that the model expressed in the resulting language can actually be executed. Our implementation solution is illustrated by the definition of a concurrent Timed Finite State Machine (TFSM) language; a language where different state machines augmented with timed transitions can be concurrently executed. Here timed transitions possibly refer to different (independent) clocks.

This section is organized according to the implementation choices presented in Figure 3. It starts with the description of the AS, then the DSA, followed by the MOC and to finish, the DSE reification is specified.

3.1 Abstract Syntax Design

In model-driven engineering, the abstract syntax is usually expressed in an object-oriented manner. For example the *de facto* standard meta-language EMOF (Essential

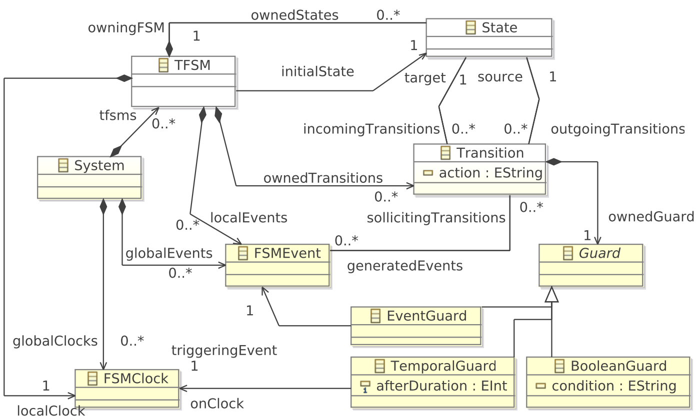

Fig. 2. Abstract Syntax of TFSM (using Ecore)

Meta Object Facility) [22] specified by the Object Management Group (OMG) can be used. EMOF provides the following language constructs for specifying an abstract syntax: package, class, property, multiple inheritance (specialization) and different kinds of associations among classes. The semantics of these core object-oriented constructs is close to a standard object model (*e.g.*, Java, C#, Eiffel).

In practice, we have chosen Ecore to design abstract syntax, a meta-language part of the *Eclipse Modeling Framework* (EMF) [23] and aligned with EMOF. This choice is motivated by the wide acceptance of Ecore and its correspondence to the MOF standard. Additionally, EMF is well tooled and many other tools are based on it (*e.g.*, XText, OCL, GMF, Obeo Designer), so that a language developed in our workbench can benefit from such tools. Note that any meta-language aligned with EMOF can be used in our approach.

Briefly, the AS of TFSM starts with a *System* composed of a set of $TFSM$s, a set of global $FSMEvent$s and a set of global $FSMClock$s (see Fig. 2). Each TFSM is composed of *State*s among which an initial state is identified. Each state can be the source of outgoing guarded $Transition$s. A guard can be specified by the reception of a $FSMEvent$ ($EventGuard$), by a duration relative to the entry time in the incoming state of the transition (*TemporalGuard*) or by a boolean condition (*BooleanGuard*). The duration of a temporal guard is measured on an explicit reference clock. An action is associated with a transition and is represented in the abstract syntax as a *String*. The condition of the boolean guard is also specified as a *String*. These strings represent model level code defined by the designer (*i.e.,*written using an opaque action language). In our experiments such model level code is written in the Groovy language[8]. Groovy was chosen for its capacity to be dynamically invoked. However any other action language could be used. Finally, a transition can also generate a set of event occurrences when fired.

Note that in the abstract syntax, we refrain from adding concepts about the execution state or functions. These concepts are specified in the DSA.

[8] http://groovy.codehaus.org/

3.2 Domain Specific Actions Design

The domain-specific actions (DSA) enriches the abstract syntax with data representing the execution state and with functions representing the execution functions. Since EMOF (and Ecore) does not include concepts for the definition of the behavioral semantics and OCL is a side-effect free language, we have used the Kermeta language to define the DSA of a DSML. Kermeta is an extension of Ecore that provides an action language used to express the behavioral semantics of a DSL [24]. Using the Kermeta language, an execution function is expressed as methods of the classes of the abstract syntax [24]. The body of the method imperatively describes what is the effect of executing an instance of the concept. The Kermeta language is imperative, statically typed, and includes classical control structures such as blocks, conditionals, loops and exceptions. The Kermeta language also implements traditional object-oriented mechanisms for handling multiple inheritance and generics. For multiple inheritance, Kermeta borrows the semantics from the Eiffel programming language [25]. Kermeta does not provide any solution to specify the concurrency model. Indeed, the concurrency semantic model is provided through the Java implicit concurrency model embedded in the underlying execution environment. As a consequence, the designer can use a foreign function interface mechanism to call the Java Thread API but there is no specific support to describe the concurrency model explicitly.

In the approach and the language workbench proposed, the AS and the DSA are conceptually and physically (at the file level) defined in two different modules. The `aspect` keyword enables DSML engineers to bind the AS and the DSA together. It allows DSML engineers to reopen a previously created class in the abstract syntax to add some new pieces of information such as new methods (execution functions) or new properties (execution state representing the semantic domain). It is inspired by open-classes (aka. static introduction) [26].

In the case of TFSM (cf. Listing 1.1), we have added the $currentState$ as an attribute of $TFSM$. We have also added $numberOfTicks$ as an integer attribute of $FSMClock$. All the instances of $TFSM$ in a system possess a current state. Also, all instances of $FSMclock$ have an integer representing their actual time. The execution state of the system is then a set of current states and a set of Integers. The choice of what should be added as attribute depends on the information we want to capture in the execution state of the models. Such information can usually be specified in various ways. For instance, we could have specified the execution state by a set of sensitive transitions instead of a set of current states. Kermeta aspects are also used to specify operations on metaclasses. They provide an operational specification of the execution functions as described in the DSA language unit. The advantage is then the executability of such operations. In TFSM, we have added six operations:

- *init()* on $TSFM$: Operation $init()$ is used to initialize the execution state of the $TFSM$ (*i.e.,*the current state in our case, lines 5 to 8).
- *fire()* on $Transition$: Operation $fire()$ is in charge of changing the current state from the source state to the target state of the transition. It is also in charge of executing the groovy code specified in the action attribute (lines 12 to 18).
- *init()* on $FSMClock$: Operation $init()$ is used to initialize the *numberOfTicks* (not shown in the listing).

– *ticks()* on *FSMClock*: Operation *ticks*() is used to increment the *numberOfTicks* of *FSMClock* (line 24 to 27).

Listing 1.1. part of the Kermeta aspects specifying the DSA

```
1  aspect class TFSM
2  {
3   //Attribute used at runtime to store the current state
4   attribute currenteState : tfsm::State
5   operation init() : String is do
6     currentState := self.initialState
7     result:= "call␣to␣init()␣:␣" + name
8   end
9  }
10 aspect class Transition
11 {
12  operation fire() : String is do
13    var groovyExpression : String init self.action
14    var res1 : kermeta::standard::Object init extern org::
15      kermeta::extra::groovyembedded::GroovyEmbeder.runOnScript(
           groovyExpression)
16    self.source.owningFSM.currentState := self.target
17    result := "fire:␣" + name + "␣->␣" +self.action
18  end
19 }
20 aspect class FSMClock
21 {
22   //Attribute used at runtime to store the number of tick
23   attribute numberOfTicks : Integer
24   operation ticks() : void is do
25     numberOfTicks := numberOfTicks + 1
26     result := "ticks:␣" + name
27   end
28 }
```

Note that while the DSAs are described by Kermeta aspects over the concepts of the AS, none of them specifies the execution workflow (like a *main*() operation). The schedule of the different operation calls is made by the concurrency model according to the MOC used in the DSML.

3.3 Model of Computation Design

The MOC defines the concurrency, the synchronizations and the possibly timed causality relationships in a DSML. The meta-language used for the specification of the MOC must be able to specify constraints on events independently of the AS and the DSA on which it is applied. We have chosen the Clock Constraint Specification Language (CCSL) [27] for specifying the MOC (at the DSML level), as well as to represent its instances as concurrency models (at the model level). In CCSL, a concurrency model is a set of constraints whose definitions and formal parameters are given in libraries. We use the library mechanism to specify MOC specific constraints. These constraints specify the correct evolution of the events given as formal parameters of the constraints. More precisely it is a reusable set of constraints considered as consistent with regards to a specific MOC; it defines the possibly timed synchronizations and causality relationships between some events and has already been shown to be a good candidate for the specification of the concurrent and temporal aspects of a language [27]. It is not possible to specify any computational aspects in CCSL so that it fits with the separation of the concurrent and temporal aspects in the MOC from the computational aspects in the DSA.

We have defined new constraints dedicated to the TFSM MOC in a specific library. For instance we have defined *TemporalTransition* and *EventTransition* constraints whose declarations are presented in Listing 1.2[9]. Each declaration exposes a set of formal parameters, which are needed to specify the constraint between the events (named *clock*s in CCSL). For instance, for the temporal transition relationship, four events are important, the event that starts the "timer", the event used to measure the time, the event that disables the transition (*i.e.,*makes it non fireable until the next timer starts), and the clock that actually fires the transition. Additionally, the integer representing the delay after which the transition should be fired is also a parameter. Such parameters represent the information that should be provided by a DSML so as the MOC can be used. Such declarations do not make any assumptions about AS and DSA. These constraints define the acceptable concurrency and the possibly timed synchronizations and causalities at the language level. A change in the library affects the execution of all models expressed in a language that uses the MOC (*i.e.,*the constraints).

Listing 1.2. Excerpt of a MoC library used for TFSM (using CCSL)

```
RelationDeclaration TemporalTransition(TemporalTransition_MakeFireable:clock,
     TemporalTransition_RefClock:clock, TemporalTransition_Reset:clock,
     TemporalTransition_delay:int, TemporalTransition_Fire:clock)
RelationDeclaration EventTransition(EventTransition_MakeFireable:clock,
     EventTransition_Trigger:clock, EventTransition_Reset:clock,
     EventTransition_Fire:clock)
```

3.4 Domain Specific Event Design

The DSE put MOC and DSA together to constitute the behavioral semantics of the DSML. They contain the events relevant to the DSML perspective and how they are linked to the execution functions of the DSA; and on the other hand they specify queries on the AS to specify the actual parameters that have to be used by the concurrency model on a specific model. To do so, a specific meta-language named ECL (standing for *Event Constraint Language* [28]) is developed as an extension of OCL [29] with events. The ECL file specifies the constraints used in the concurrency model for a specific model, by specifying the link between the MOC, the DSA and the AS of a DSML. ECL benefits from the OCL query language and its possibility to augment an abstract syntax with additional attributes (without any side effects). Using ECL it is then possible to define new DSE in the context of a specific concept of the AS. DSE also specify, if needed, the execution function that must be called when specific events occur. For instance, in the TFSM example which is partially represented in Listing 1.3, we have defined three domain specific events in the context of $FSMEvent$, $FSMClock$ and $Transition$ (lines 5–10 in Listing 1.3). The events defined in the context of $FSMClocks$ and $Transition$, respectively call when they occur the execution function $ticks()$ defined in the context of $FSMClock$ and the execution function $fire()$ defined in the context of $Transition$.

The ECL file imports a MOC library (line 2 in Listing 1.3). It is used to define some invariants that specify in which context and with which parameter(s) a constraint from

[9] The definitions are not given for the sake of clarity

the MOC is used. The specification of the actual parameters are specified by querying the AS. To specify the mapping between MOC and DSA, it is also possible to create intermediate events by using expressions over existing DSE. For instance, lines 13 to 23 represents the invariant that specifies that for each transition of the AS whose guard is of type *TemporalGuard*, if the source state of this transition has more than one other outgoing transition (line 16), then there is a constraint of type *TemporalTransition* in the concurrency model (line 21). The parameters of the constraints can be queried on the AS like in the line 17 or 22 and 23. It can also be specified by an expression over existing domain specific events like specified in line 18 to 20, which specify a new event defined by the *Union* of all the fire events from other outgoing transitions from the same source state. It is used here to specify when the event transition must be disabled (see the formal parameters line 8 in Listing 1.2). These queries define how the structure of the AS is used to retrieve the actual parameters. For instance, the actual duration of the temporal transition is defined by the *afterDuration* attribute defined in the AS (line 17).

Listing 1.3. Excerpt of the Domain-Specific Events of TFSM (using ECL)

```
1  import 'http://fr.inria.aoste.gemoc.example.tfsm'
2  ECLImport "TFSMMoC.ccslLib"
3  package tfsm
4  // DSE definition, and mapping of the DSE to the DSA (i.e., Kermeta method)
5    context FSMEvent
6      def: occurs : Event()
7    context FSMClock
8      def: ticks : Event(self.ticks())
9    context Transition
10     def: fire : Event(self.fire())
11 // Mapping of the DSE to the MOC
12   context Transition
13   inv fireWhenTemporalGuardHoldsVariousTransition:
14     (self.ownedGuard.oclIsKindOf(TemporalGuard)
15       and self.source.outgoingTransitions->
16         select(t|t <> self)->size() > 0) implies
17       let guardDelay : Integer = self.ownedGuard.oclAsType(TemporalGuard).
             afterDuration in
18       let otherFireFromTheSameState: Event =
19         Expression Union (self.source.outgoingTransitions->
20                              select(t|t <> self).fire) in
21       Relation TemporalTransition(self.source.entering,
22                     self.ownedGuard.oclAsType(TemporalGuard).onClock.ticks,
23                     otherFireFromTheSameState, guardDelay, self.fire )
24 // Using a MoC constraint specifying a rendez-vous semantics
25   context FSMEvent
26     inv occursWhenSolicitate:
27       (self.sollicitingTransitions->size() >0) implies
28       let AllTriggeringOccurrences : Event = Expression
29       Union(self.sollicitingTransitions.fire) in
30       Relation FSMEventRendezVous(AllTriggeringOccurrences, self.occurs)
```

Listing 1.3 shows another invariant, which defines the *FSMEventRendezVous* constraint on the MOC. This constraint is changed in section 4 to highlight the impact of a MOC variation. From such a specification, it is possible to generate a CCSL specification that represents the concurrency model for any model that conforms to the AS; *i.e.,*a model that contains the actual constraints and their parameters according to a specific model. ECL restricts the requirements expressed in section 2. For instance, it is not possible yet to specify how the result of an execution function call influences the execution path taken by the execution engine at runtime. Such information is for now defined in the configuration of the execution engine. Information about the execution engine is given in the next subsection.

3.5 Execution Engine

Each unit of a DSML is described in our language workbench using technologies built on top of the Eclipse Modeling Framework (EMF). To summarize, we describe the AS with the meta-language Ecore part of EMF (*1* in Fig. 3). Then we describe the DSA (both execution state and functions) with Kermeta [24] (*2* in Fig. 3). We mainly use Kermeta for its weaving capability on the abstract syntax. DSA are specified using aspects on metaclasses that enable the addition of execution state attributes and execution functions. Then, we specify the MOC by constraints definition as a CCSL library (*3* in Fig. 3). Finally we define DSE and link them with the execution functions by using ECL (*4* in Fig. 3). ECL is also used to specify, at the language level, the constraints used in a concurrency model for a specific model.

In the workbench, EMF generates an API for the AS that can load and save models conforming to the DSML. Kermeta methods and properties are compiled as a set of Scala traits that are woven within this model API [30]. As a result, Kermeta provides an extended version of the Java API, encapsulated in a jar file, on which it is possible to call the execution functions weaved within the AS (*5* in Fig. 3). Then, for a specific model conforming to the DSML, the ECL file can be used to automatically create a concurrency model in CCSL (*6* in Fig. 3). The concurrency model is directly linked to the model elements. This model represents all the partial ordering of events considered as correct with regards to the MoC. In our workbench, it is interpreted by a tool named TIMESQUARE [31] to provide one partial ordering between the domain specific events in the model (*7* in Fig. 3). To call the execution functions defined in the DSA, TIMESQUARE has been extended in our language workbench with a new back-end able to use the jar files to execute the model (*8* in Fig. 3). In the proposed language workbench, EMF serves as a common technical foundation. Kermeta provides an API fully compatible with EMF that eases the integration within the language workbench.

As a result, this language workbench provides a set of Java libraries allowing to call execution functions on a model. The call to the execution functions is driven by TIMESQUARE. The (possibly simultaneous) ordering of the calls to the execution functions represents the concurrent-aware execution of the model. We illustrate in the next section the use of the TFSM language on five concurrent TFSMs, which model road traffic lights and their controller.

4 Demonstration and Discussion: Using TFSM on Concurrent Road Traffic Lights

To go further in our approach, we present an example of five concurrent TFSMs built using the executable language proposed in the previous sections and the tools mentioned above. Our case study is a simple modeling of crossroad traffic lights. In our example, the traffic lights regulate the traffic on a main road and a secondary road. The traffic lights are synchronized differently during the day or the night. During the day, the two

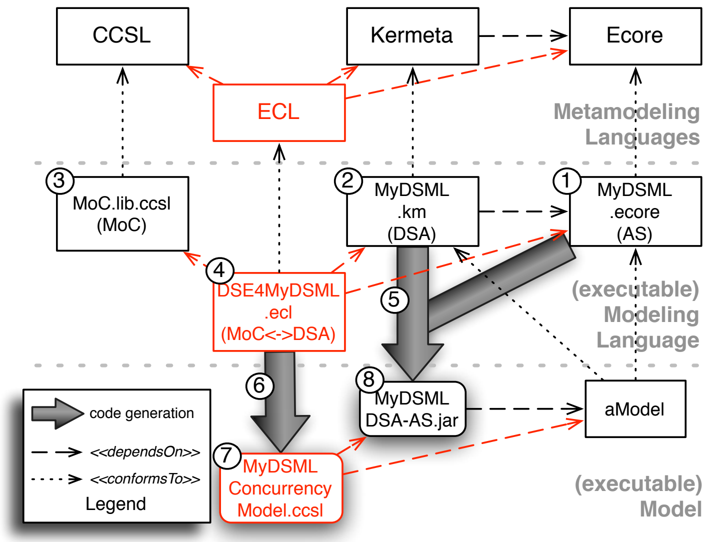

Fig. 3. Architecture of the language workbench and the associated execution engine for concurrent model execution

traffic lights on the main road are red during two minutes and then switch to green. They remain green until a controller sends the *switch* event that make the two main traffic lights become red again. The two other traffic lights have exactly the same behavior but are green when the main traffic lights are red and red when the main traffic lights are green. The controller has two states Day and Night that change depending on the reading of a sensor answering whether it is night or day. During the day it sends a switch event every 4^{th} minute and during the night every 6^{th} minute. In Figure 4, the controller TFSM (named *Control*) and one of the main road traffic lighs (named *Semaphore0*) are shown. The abstract syntax has been tooled with Obeo designer to obtain a graphical concrete syntax.

By using this example we want to highlight the impact of a simple change in a MOC applied on the same AS/DSA. The MOC variation consists in the synchronization between the firing of a transition, and the production of the occurrences of its *gen-*

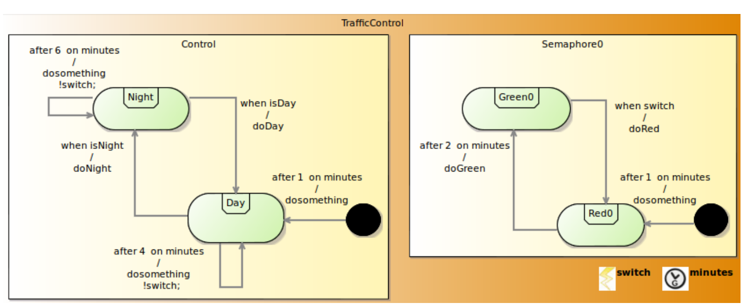

Fig. 4. Partial traffic light model

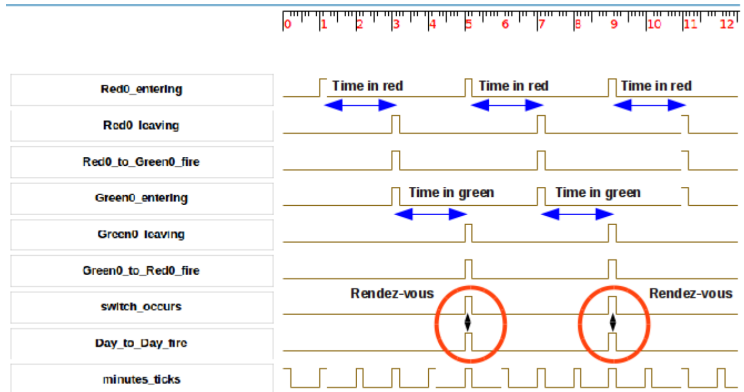

Fig. 5. Timing output of the simulation with the rendez-vous semantics

eratedEvents. In the first case, we use a strong synchronization, meaning that if one transition is fired and generates an event occurrence, all the transitions waiting for this event are simultaneously fired (The transitions with an event guard on this event). For that purpose, we use a constraint named *FSMEventRendezVous* in the MOC library (see the lines 27 to 32 of the Listing 1.3). Once the concurrency model has been generated for the model of Figure 4, we can execute the concurrency model in TIMESQUARE. The model execution produced a timing diagram representing the occurrences of the event according to the time (Figure 5). On this picture we can see that the firings of the transition named *Day_to_Day* from the traffic light controller are simultaneous with the occurrences of the switch event, themselves simultaneous with the firings of the *Green0_to_Red0* transition. In this case, the time spent in the *Red0* state and the time in the *Green0* state is the same: 2 minutes. This mechanism is a strong synchronization so that if the TFSM of *Semaphore0* (Figure 4) is in the *Red0* state when the *Day_to_Day* transition is fired, a deadlock happens.

In a second case, we have changed this strong synchronization with a causal relationship, meaning that if one transition is fired and generates an event occurrence, all the transitions waiting for this event must be fired in a later step (it abstracts the sending and the reception of a FSMEvent). For this purpose, we have modified the MOC library and have replaced the *FSMEventRendezVous* with a constraint named *FSMEventSendReceive*, defining the causal relationship. In this case, all the parameters remain identical, the constraint definition in the MOC library is the only modification made. The model execution produces a different timing diagram and the time spent in the *Green0* and in *Red0* state is now different. The time between the *Day_to_Day* transition firing and the occurrence of the switch event is not bound and could vary. In this case, contrary to the previous case, if the TFSM of *Semaphore0* is in the *Red0* state when the *Day_to_Day* transition is fired, no deadlock happens.

By defining the concurrent TFSM language according to the approach and workbench proposed in this paper, we have executed the TFSM instances concurrently. By changing a single constraint in the MOC library, we get a different behavior of the system, highlighting the importance to make explicit and to reify the MOC and

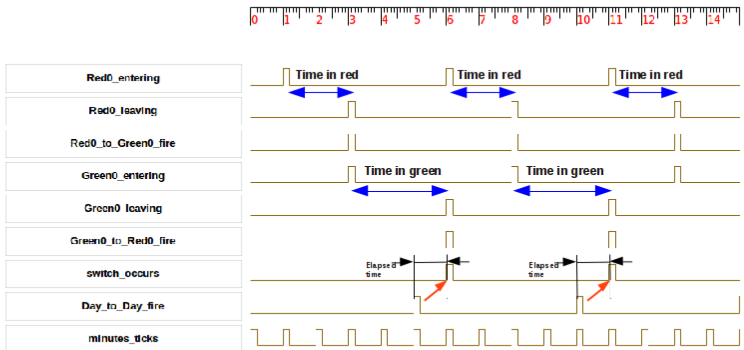

Fig. 6. Timing output of the simulation with the send-receive semantics

DSE. The presentation of the whole example with video of its compilation and execution (including diagram animation) can be found on the companion web page: `http://gemoc.org/sle13`. This web page also introduces two other usages of the language workbench. The first one shows the definition of the Actor Computing Model using the workbench. It allows the simulation of the behavior of a set of Actors. The second one illustrates an example of the Logo language[10] with two turtles sharing the same playground.

5 Related Work

Much work has been done on the design and implementation of both DSML and models of computation. In this paper, we propose a conceptual and technical framework to take benefits from both underlying theory. This section presents related work in the field of language design and implementation, and then in the field of models of computation.

The problem of the modular design of languages has been explored by several authors (*e.g.*, [32,33]). For example, JastAdd [33] combines traditional use of higher order attribute grammars with object-orientation and simple aspect-orientation (static introductions) to get a better modularity mechanism. With a similar support for object-orientation and static introductions, Kermeta and its aspect paradigm can be seen as an analogue of JastAdd in the DSML world. The major drawback of such approach is that none of them provides a native support for concurrency.

A language workbench is a software package for designing software languages [34]. For instance, it may encompass parser generators, specialized editors, DSLs for expressing the semantics and others. Early language workbenches include Centaur [35], ASF+SDF [36], and TXL [37]. Among more recent proposals, we can cite Generic Model Environment (GME) [38], Metacase's MetaEdit+ [39], Microsoft's DSL Tools [40], Krahn et al's Monticore [41], Kats and Visser's Spoofax [42], Jetbrain's MPS [43]. The important difference of our approach is that we explicitly reify the concurrency concern in the design of an executable language, providing a dedicated tooling

[10] `http://en.wikipedia.org/wiki/Logo_(programming_language)`

for its implementation and reuse. Our approach is also 100% compatible with all EMF-based tools (at the code level, not only at the abstract syntax level provided by Ecore), hence designing a DSL with our approach easily allows reusing the rich ecosystem of Eclipse/EMF.

Models of computation, and in particular the concurrency concern, have been mainly tooled in three different workbench: Ptolemy [6], ModHel'X [44] and ForSyDe [45]. Each of them have their own pros and cons but they are all based on a specific abstract syntax and API. On one hand the unique abstract syntax avoids their use in the context of specific DSMLs and on the other hand the use of an API to apply a specific MOC creates a gap between the MOC theory and the corresponding framework. In our approach we use the notion of DSE to link a MOC to the DSA of a specific DSML and we use CCSL to specify the MOC in a formal way, closer to theory like event structures or tagged signals. A similar approach has been used in BIP [46], where a specific algebra is used to describe the interactions through connectors between behaviors expressed in timed automata. From the properties of the connectors, it is possible to predict global properties of the models. This approach is interesting in its analysis capacity but is tailored to the composition of timed automata. Finally another approach based on CCSL has been used in [47] to describe two MOC and the interactions between heterogeneous models of computation. This approach improved ModHel'X workbench but is still dedicated to apply a MOC to a specific abstract syntax. However, it gives good hint for the use of the approach proposed in this paper for the composition of heterogeneous executable modeling languages.

6 Conclusion and Perspectives

This work proposes an approach that reifies the key concerns to design and implement a concurrency-aware executable DSML (AS, DSA, MOC and DSE). The approach is supported by a language workbench based on EMF, including a meta-language dedicated to each concern to design concurrency-aware executable DSMLs in a modular way. Then, the implementation of a DSML automatically results in a dedicated environment for concurrent execution of the conforming models. The explicit modeling of concurrency as first-class concern paves the way to a full understanding and configuration ability of the behavioral semantics. Additionally, the modular design enables the reuse of existing MoCs that come with specific analysis capabilities and tool support. We illustrate our approach and language workbench on the design, the implementation and the use of variants, of concurrent and timed finite state machine. A complementary video is available on the companion webpage, as well as other DSML families implemented according to our approach: `http://gemoc.org/sle13`.

In future works, we plan to focus more on the DSA and DSE relationships. Up to now, the events are driving the execution of actions, but only a crude feedback is allowed from the actions. The understanding of what kind of feedback is expected needs to be further explored. Finally, the explicit definition of concurrency in the behavioral semantics of DSML opens many perspectives. In particular, we are exploring the way to support heterogeneous execution models (*e.g.*, synchronization and composition of interpreter or compiler). The goal here is to make explicit the composition of heterogeneous DSMLs by using the information provided by the reified language units.

References

1. Combemale, B., Crégut, X., Pantel, M.: A Design Pattern to Build Executable DSMLs and associated V&V tools. In: APSEC, IEEE (December 2012)
2. Object Management Group, Inc.: Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0. (2011)
3. Combemale, B., Hardebolle, C., Jacquet, C., Boulanger, F., Baudry, B.: Bridging the Chasm between Executable Metamodeling and Models of Computation. In: SLE, Springer (2012)
4. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A framework for comparing models of computation. IEEE Trans. on CAD of Integrated Circuits and Systems **17**(12) (1998) 1217–1229
5. Jantsch, A.: Modeling Embedded Systems and SoCs. Morgan Kaufmann Publishers Inc. (2004)
6. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity – the Ptolemy approach. Proc. of the IEEE **91**(1) (2003)
7. Plotkin, G.D.: A structural approach to operational semantics. (1981)
8. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the use of graph transformations for the formal specification of model interpreters. Journal of Universal Computer Science **9** (2003)
9. Bendraou, R., Jézéquel, J.M., Fleurey, F.: Combining aspect and model-driven engineering approaches for software process modeling and execution. In: Trustworthy Software Development Processes. LNCS. Springer (2009) 148–160
10. Knuth, D.E.: Semantics of context-free languages. Theory of Computing Systems **2**(2) (1968) 127–145
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580
12. Gries, D.: The science of programming. Volume 198. Springer (1981)
13. Winskel, G.: The formal semantics of programming languages: an introduction. MIT press (1993)
14. Fredlund, L.a., Jonsson, B., Parrow, J.: An implementation of a translational semantics for an imperative language. In: CONCUR. LNCS. Springer (1990) 246–262
15. Scott, D.S., Strachey, C.: Toward a mathematical semantics for computer languages. Oxford University Computing Laboratory, Programming Research Group (1971)
16. Milner, R.: A calculus of communicating systems. Springer (1982)
17. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8) (1978) 666–677
18. Petri, C.A.: Introduction to general net theory. In: Advanced Course: Net Theory and Applications. (1975) 1–19
19. Winskel, G.: Event structures. In Brauer, W., Reisig, W., Rozenberg, G., eds.: Petri Nets: Applications and Relationships to Other Models of Concurrency. LNCS. Springer (1987)
20. McCarthy, J.: Towards a mathematical science of computation. Information processing **62** (1962) 21–28
21. Boulanger, F., Hardebolle, C.: Simulation of Multi-Formalism Models with ModHel'X. In: ICST, IEEE (2008) 318–327
22. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Core. (2006)
23. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley (2008)

24. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Meta-Languages. In: MoDELS. LNCS, Springer (2005) 264–278
25. Meyer, B.: Eiffel: the language. Prentice-Hall, Inc. (1992)
26. Clifton, C., Leavens, G.T.: Multijava: Modular open classes and symmetric multiple dispatch for java. In: OOPSLA. (2000) 130–145
27. Mallet, F., DeAntoni, J., André, C., de Simone, R.: The Clock Constraint Specification Language for building timed causality models. Innovations in Systems and Software Engineering **6** (2010) 99–106
28. Deantoni, J., Mallet, F.: ECL: the Event Constraint Language, an Extension of OCL with Events. Research report RR-8031, INRIA (July 2012)
29. Object Management Group, Inc.: UML Object Constraint Language (OCL) 2.0. (2003)
30. Jézéquel, J.M., Combemale, B., Barais, O., Monperrus, M., Fouquet, F.: Mashup of meta-languages and its implementation in the kermeta language workbench. SoSyM (2013)
31. Deantoni, J., Mallet, F.: TimeSquare: Treat your Models with Logical Time. In: TOOLS. Volume 7304 of LNCS., Springer (May 2012) 34–41
32. Wyk, E.V., Moor, O.d., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: CC'02, Springer (2002) 128–142
33. Ekman, T., Hedin, G.: The JastAdd system – modular extensible compiler construction. Sci. Comput. Program. (2007) 14–26
34. Volter, M.: From Programming to Modeling-and Back Again. Software, IEEE **28**(6) (2011)
35. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: the system. In: 3rd ACM software engineering symposium on Practical software development environments, ACM (1988) 14–24
36. Klint, P.: A meta-environment for generating programming environments. ACM TOSEM **2**(2) (1993) 176–201
37. Cordy, J.R., Halpern, C.D., Promislow, E.: TXL: a rapid prototyping system for programming language dialects. In: Conf. Int Computer Languages. (1988) 280–285
38. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. IEEE Computer **30**(4) (1997)
39. Tolvanen, J., Rossi, M.: MetaEdit+: defining and using domain-specific modeling languages and code generators. In: Companion of the 18th annual ACM SIGPLAN conference OOPSLA, ACM (2003) 92–93
40. Cook, S., Jones, G., Kent, S., Wills, A.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional (2007)
41. Krahn, H., Rumpe, B., Volkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: Objects, Components, Models and Patterns. LNBIP, Springer (2008)
42. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. In: OOPSLA '10, ACM (2010) 444–463
43. Voelter, M., Solomatov, K.: Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. In: SLE. LNCS, Springer (2010)
44. Hardebolle, C., Boulanger, F.: Multi-Formalism Modelling and Model Execution. International Journal of Computers and their Applications **31**(3) (July 2009) 193–203
45. Sander, I., Jantsch, A.: System Modeling and Transformational Design Refinement in ForSyDe. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **23**(1) (2004) 17–32
46. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time systems in BIP. In: 4th IEEE SEFM. (September 2006) 3–12
47. Boulanger, F., Dogui, A., Hardebolle, C., Jacquet, C., Marcadet, D., Prodan, I.: Semantic Adaptation Using CCSL Clock Constraints. In: Workshops and Symposia at MODELS 2011. LNCS, Springer (2012) 104–118