Towards a Meta-Language for the Concurrency Concern in DSLs

Julien Deantoni‡, Issa Papa Diallo*, Ciprian Teodorov*, Joel Champeau*, Benoit Combemale†
* Lab-STICC - ENSTA Bretagne, France
{papa_issa.diallo, ciprian.teodorov, joel.champeau}@ensta-bretagne.fr
† University of Rennes 1 / INRIA, France
{benoit.combemale}@irisa.fr
‡ University of Nice Sophia Antipolis, France
{julien.deantoni}@polytech.unice.fr

***Abstract*—Concurrency is of primary interest in the development of complex software-intensive systems, as well as the deployment on modern platforms. Furthermore, Domain-Specific Languages (DSLs) are increasingly used in industrial processes to separate and abstract the various concerns of complex systems. However, reifying the definition of the DSL concurrency remains a challenge. This not only prevents leveraging the concurrency concern of a particular domain or platform, but it also hinders: *a)* the development of a complete understanding of the DSL semantics; *b)* the effectiveness of concurrency-aware analysis techniques; *c)* the analysis of the deployment on parallel architectures. In this paper, we introduce the key ideas leading toward MoCCML, a dedicated meta-language for formally specifying the concurrency concern within the definition of a DSL. The concurrency constraints can reflect the knowledge in a particular domain, but also the constraints of a particular platform. MoCCML comes with a complete language workbench to help a DSL designer in the definition of the concurrency directly within the concepts of the DSL itself, and a generic workbench to simulate and analyze any model conforming to this DSL. MoCCML is illustrated on the definition of an lightweight extension of SDF (Synchronous Data Flow [1]).**

I. Introduction

Concurrency is at the heart of modern software-intensive systems and platforms. Complex systems such as the Internet of things and cyber-physical systems are highly concurrent systems per se. Moreover, modern platforms like many-core, GPGPU and distributed platforms are providing more and more parallelism. In the development of such complex software-intensive systems, the correct specification of concurrency is central for leveraging the unique characteristics of these systems and their deployment on the platforms.

In the last decades, dedicated languages like π-calculus [2], CCS [3] or even data-flow languages were developed to express the concurrency of a given system. These languages provide interesting features like verification and validation facilities. However, there is a gap between the concepts offered by these languages and the concepts needed by designers to capture their domain-specific problems.

To bridge this gap, Domain-Specific Languages (DSLs) are increasingly used in industrial processes. They provide concepts and expressiveness tailored to the domain, usually making the designer more efficient [4]. However, the specification and reification of the concurrency constraints remains difficult within the definition of a DSL. Most of the time the concurrency model remains implicit and ad-hoc, embedded in the underlying execution environment. The lack of an explicit concurrency model prevents: *1)* the complete understanding of the DSL's semantics, *2)* the effective usage of concurrency-aware analysis techniques, and *3)* the exploitation of the concurrency model during the system refinement (*e.g.,* during its allocation on a specific platform).

There exist state of the art approaches like [5]–[7], which propose to explicit the model of concurrency and communication (MoCC[1]) of a language. Making a MoCC explicit enables fine tuning of the computational semantics and usually offers simulations facilities. However, these approaches provide either the capacity to adapt to a DSL (Ptolemy [5] or Modhel'X [6]) or the capacity to drive formal refinement and reasoning on the system (Forsyde [7]). Hence, forcing the designer to choose between domain adequacy and analysis power. To the best of our knowledge there is no approach providing a formal and explicit concurrency model that can be embedded in the definition of a the domain specific concepts to create an executable DSL.

In this paper, we present MoCCML, a dedicated declarative meta-language for formally and explicitly specifying the concurrency concern within the definition of a DSL. In the same way BNF [8] or MOF [9] are meta-languages dedicated to the specification of the syntax, MoCCML is a meta-language dedicated to the definition of the concurrency concern (*i.e.,*the MoCC). Moreover, MoCCML can take into account the unavoidable impacts introduced by the choice of a deployment platform on concurrency and timing. Based on its formal semantics, MoCCML and the associated tooling offer a generic workbench for simulation and exploration of any model conforming to a DSL and whose concurrency semantics is defined in MoCCML.

The overall description of MoCCML is given in Section II. To highlight our approach, we define in Section III the MoCC of a lightweight extension of the Synchronous Data Flow Llanguage (SDF [1]).

[1]In the context of this paper, a MoCC is a focused vision of the model of computation (MoC), which defines the concurrency, the synchronizations and the possibly timed way the element of a program interact during an execution.

978-3-9815370-4-8/DATE15/©2015 EDAA

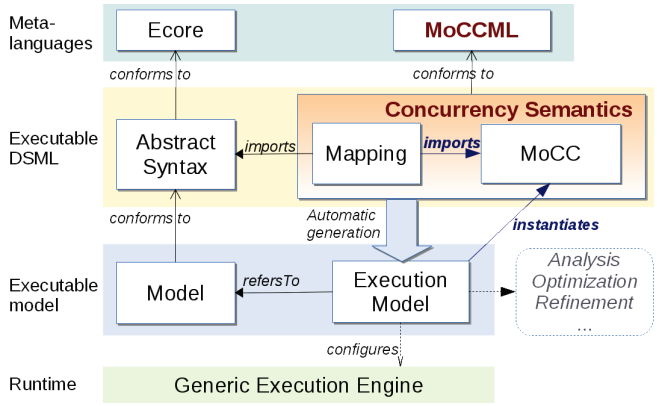

Fig. 1. Big Picture of MoCCML

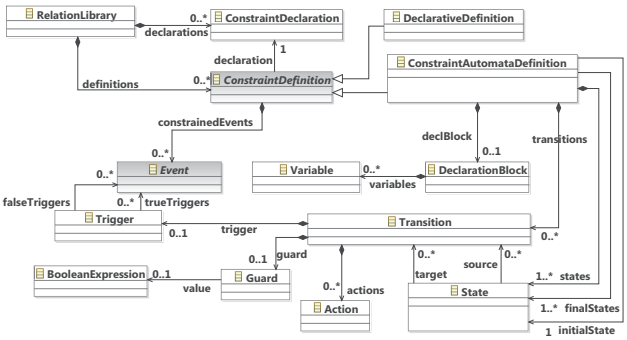

Fig. 2. Excerpt of the MoCCML Metamodel

II. Concurrency Modeling with MoCCML

The meta-language MoCCML tends to crystallize the best practices from the concurrency theory and the model-driven engineering. It leverages experiences on the explicit definition of the valid scheduling of an application through a clock constraint language [10] and an automata-based language [11]. It also reifies the appropriate concepts to enable automated reasoning.

A. MoCCML Overview

MoCCML is a declarative meta-language specifying constraints between the events of a MoCC. At any moment during a run, an event that does not violate the constraints can occurs. The constraints are grouped in libraries that specify MoCC specific constraints (named MoCC on Figure 1 and conforming to MoCCML). These constraints can also be of a different kinds, for instance to express a deadline, a minimal throughput or an hardware deployment. They are eventually instantiated to define the execution model of a specific model (see Figure 1). The execution model is a symbolic representation of all the acceptable schedules for a particular model.

To enable the automatic generation of the execution model, the MoCC is weaved into the context of specific concepts from the abstract syntax of a DSL. This contextualization is defined by a mapping between the elements of the abstract syntax and the constraints of the MoCC (achieved by the box named *Mapping* in Figure 1). The mapping defined in MoCCML is based on the notion of event, inspired by ECL [12], an extension of the Object Constraint Language [13]. The separation of the mapping from the MoCC makes the MoCC independent of the DSL so that it can be reused. From such description, for any instance of the abstract syntax it is possible to automatically generate a dedicated execution model (see "executable model" in Figure 1).

In our approach, this execution model is acting as the configuration of a generic execution engine (see "generic execution engine" in Figure 1), which can be used for simulation or analysis of any model conforming to the abstract syntax of the DSL.

MoCCML is defined by a metamodel (*i.e.,*the abstract syntax) associated to a formal Structural Operational Semantics [14]. MoCCML comes with a model editor combining textual and graphical notations, as well as analysis tools based on the formal semantics for simulation and exhaustive exploration.

In the remainder of this section, we present the concepts of MoCCML (*i.e.,*the MoCCML metamodel), its concrete syntax and the semantics behind these concepts.

B. MoCCML Syntax

1) Abstract Syntax: MoCCML is based on the principle of defining constraints on events. In the abstract syntax, there are two categories of constraint definitions: the *Declarative Definitions* and the *Constraint Automata Definitions* (see Figure 2). Each constraint definition has an associated *ConstraintDeclaration* that define the prototype of the constraint. These definitions constraint some *Events*.

A declarative definition is defined as a set of constraint instances. For more details, we refer the reader to [15] that described the declarative part inspired from the CCSL language.

As illustrated in Figure 2, a *Constraint Automata Definition* contains a set of *States* with a single initial state and one or more final states. It also contains *DeclarationBlocks* where local *Variables* can be declared. To ease exhaustive simulations we restricted the types of the variables (and parameters to be Event or Integer).

The constraint automata definition introduces the concept of Transition which links a *source* state and a *target* state. It contains a *Trigger* that defines two sets of events (namely *trueTriggers* and *falseTriggers*). The transition is fired if the events in the *trueTriggers* set are present and the ones in the *falseTriggers* set are absent. A transition can define a *Guard*. A guard is a boolean expression over the local variables or the parameters of the definition. Finally, during the firing of a transition, actions such as integer assignments (possibly with a value resulting from an expression such as the increment of a counter) can operate on the local variables.

2) Concrete Syntax: The concrete syntax of MoCCML is implemented as a combination of graphical and textual syntaxes to provide the most appropriate representation for each part of a MoCC conforming to the aforementioned abstract syntax.

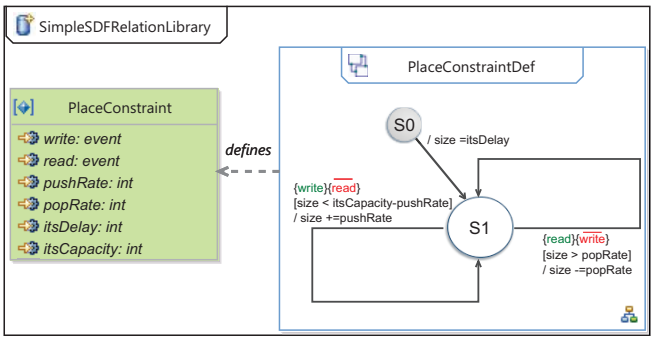

Fig. 3. Screenshot of the MoCCML graphical editor

The graphical model shown in Figure 3 defines a MoCC Constraint Library (*SimpleSDFRelationLibrary*), which contains a constraint declaration named *PlaceConstraint*. The constraint declaration is associated to a constraint automata definition (*PlaceConstraintDef*). In this library, we define a constraint between the $read$ and $write$ events. The automaton operates on 5 integer parameters (one variable: $size$; and 4 constants: $itsCapacity$, $itsDelay$, $pushRate$, $popRate$), which are set during the instantiation process.

C. MoCCML Semantics

This section overviews the operational semantics of MoCCML, which allows the effective construction of the acceptable schedules. The interested reader can refer to [15] for a full definition of the operational semantics. An execution model consists in a finite set of discrete events, constrained by a set of constraints. A *schedule* σ over a set of events E is a possibly infinite sequence of *Steps*, where a step is a set of occurring events. $\sigma : \mathbb{N} \rightarrow 2^E$. For each step, one or several event(s) can occur. The goal of the semantics rules is to specify how to construct the acceptable schedules.

The semantics of a specification expressed in MoCCML is given as a Boolean expression on $\mathscr{E}$, where $\mathscr{E}$ is a set of Boolean variables in bijection with E. For any $e \in \mathscr{E}$, if e is valued to *true* then the corresponding event occurs; if valued to *false* then it does not occur. If no constraints are defined, each boolean variable can be either true or false and there are 2^n possible futures for all steps, where n is the number of events. Consequently, in this case the number of acceptable schedules is infinite.

Each time a constraint is added to the specification, it adds boolean constraints on $\mathscr{E}$. The boolean constraints depends on the definition of the MoCCML constraint and its internal state. When several MoCCML constraints are defined, their boolean expressions are put in conjunction so that each added constraint reduces the set of acceptable schedules. For instance, if the *sub-event* declarative constraint is defined between two events e1 and e2 (*i.e.,*e1 *sub-event* of e2), then the corresponding boolean expression is e1 $\Rightarrow$ e2.

The same principle applies to the constraint automata definitions. The boolean expression associated to a specific constraint automata is obtained according to: 1) the value of the automata local variables; 2) the current state; 3) the evaluation of boolean guards on the output transition of the current state and 4) the triggers ($trueTriggers$ and $falseTriggers$) on the output transitions of the current state.

The semantics of a constraint automata is defined as a *logical disjunction* of the boolean expressions associated to the output transitions of the current state. For a transition t, if its guard is valued to true, the resulting boolean expression is the conjunction of all the events in the $trueTrigger$ set in conjunction with the conjunction of the negation of all the events in the $falseTrigger$ set. For instance, in the constraint automata depicted in Figure 3, the boolean expression when $size$ is lesser than $itsCapacity$ minus $pushRate$ is: $write \wedge \neg read$. In the case where $size$ is also greater than $popRate$ the automata semantics is $(write \wedge \neg read) \vee (read \wedge \neg write)$. If the new computed step is such that the boolean equation of one transition is valued to true, then the transition is fired, meaning that the current state evolves to the target of the fired transition and the actions of this transition are executed.

At this step, we introduced the syntax and the semantics of MoCCML. In the next section we illustrate the use of MoCCML to define a MoCC and its mapping to an illustrative DSL.

III. MoCC Definition of a Signal Processing DSL

In our extension of the SDF syntax, an application is described as a set of *Agent*s. Upon activation, each agent uses the data on its *Input Port*s, executes N processing cycles and produces computed results on its *Output Port*s. Data in transition between *Agent*s are stored in *Place*s with limited capacity.

A. SDF MoCC

Before explicitly introducing the MoCC constraints, the set of relevant SDF events have to be identified. In the context of an *Agent* the relevant events are: *start*, *stop* and *isExecuting* (see Listing 1). Moreover, the ports have also the *read* (input port) and *write* (output ports) events.

The events are part of the mapping since they are defined in the *context* of a concept of the DSL and are used as parameters by MoCCML constraints (see line 7 listing 1).

```
1 context Agent
2   def : start : Event
3   def : stop : Event
4   def : isExecuting : Event
5 context Place
6  inv PlaceLimitation:
7  Relation PlaceConstraint(self.outputPort.write,self
       .inputPort.read, self.outputPort.rate, self.
       inputPort.rate, self.delay, self.capacity)
```

Listing 1. Part of the event and constraint mapping in the context of the SDF concepts

With such a mapping, for a specific model, any instance of the meta class *Agent* is associated to the three events. These events have to be constrained to provide the adequate semantics. For instance in line 6 of Listing 1 the events are used in a constraint (i.e. *PlaceConstraint*) defined in the context of a *Place*. The *PlaceConstraint* automata is shown on Figure 2. It defines a constraint between the *read* of an input

port and the *write* event of an output port linked by a place. This automata imposes that *read* does not occur if there is not enough data in the place and *write* does not occur if there is not enough room in the place. This constraint is instantiated for each instance of *Place* in a *SigPML* model.

The reader should note that this automata could be modified to provide variants of the semantics. For instance, one could add a transition to specify that *read* and *write* can be done simultaneously (as supported by multiport memories).

Another constraint automata, not represented in this paper, has been defined in the context of an Agent. It defines that: *1) read* is simultaneous to *start*, *2) isExecuting* occurs only between *start* and *stop 3) stop* occurs at the N^{th} occurrences of *isExecuting* that follows *start*, and *4) stop* is simultaneous to a *write*. In the case where N equals 0 (*i.e.,*the SDF abstraction), then the *read*, the *start*, the *stop* and the *write* are simultaneous. However, an execution time can be specified, for example according to a deployment on a specific platform.

These two constraint automata reproduce the SDF semantics and they are explicitly defined on the concepts of the metamodel.

At this point we note two important characteristics of our approach:

- It offers the ability to define (and to vary) the MoCC of a dedicated modeling language, much as Ptolemy or Modhel'X. However, instead of doing so by implementing a specific API in a general purpose language, we rely on a formal model suitable for analysis;
- We do not enforce the designer to use a proprietary DSML, as opposed to formal framework such as ForSyDe. Instead, we are injecting the MoCC into the designer appropriate language.

IV. Conclusion

In the current state of the art we identified a lack in the possibility to formally define the concurrency concern in the definition of a Domain Specific Language (DSL), *i.e.,*based on the definition of the domain specific concepts. We proposed to fill this lack by defining a dedicated meta-language with a formal semantics. By doing so we enrich the DSL syntax with an explicit MoCC, mapped onto the domain specific concepts. MoCCML is illustrated with the definition of a lightweight extension to SDF. Such an explicit MoCC can be used by a generic execution engine to drive simulation and analysis on any model conforming to the DSL.

We also extended SDF (*i.e.,*the syntax and the MoCC) to define a deployment on a simple platform. Due to place restriction, this is not described in this paper but on a companion website[2]. As preliminary results, we have shown the possibility to extend the SDF MoCC to specify both application and platform constraints. In this context, the SDF extension is used to model and validate an application from the Passive Acoustic Monitoring (PAM) domain. We first model a PAM system under an infinite resource assumption before studying three different deployments on different platforms. The extended MoCC has been used to evaluate, through simulation traces and exhaustive exploration, the impact of the different allocations on the valid scheduling of the application. These experiments shown the ability to understand the impact of the deployment on the actual parallelism of a given system. It also validated the configuration of the generic execution engine by an execution model to offer the possibility to simulate the different models and to obtain by exploration quantitative results on the scheduling state-space.

MoCCML paves the way to various usages of an explicit concurrency model in a DSL specification. While we focused on the use of the concurrency model for validation and verification purposes, it opens the way for, among others, design space exploration, optimizing compiler and adaptation at runtime.

[2]http://gemoc.org/date15/

Acknowledgment

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

References

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
[2] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.
[3] ——, "A calculus of communicating systems," 1980.
[4] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry," in *ICSE*. ACM, 2011, pp. 471–480.
[5] 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*, vol. 91, no. 1, pp. 127–144, 2003.
[6] C. Hardebolle and F. Boulanger, "Modhel'x: A component-oriented approach to multi-formalism modeling," in *Models in Software Engineering*. Springer, 2008, pp. 247–258.
[7] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe [formal system design]," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, no. 1, pp. 17–32, 2004.
[8] D. D. McCracken and E. D. Reilly, "Backus-naur form (bnf)," in *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., 2003, pp. 129–131.
[9] OMG, *Meta Object Facility (MOF)*, 2014, Version 2.4.2.
[10] F. Mallet, J. DeAntoni, C. André, and R. de Simone, "The clock constraint specification language for building timed causality models - application to synchronous data flow graphs," *ISSE*, vol. 6, no. 1-2, pp. 99–106, 2010.
[11] P. I. Diallo, J. Champeau, and V. Leilde, "Model based engineering for the support of models of computation: The cometa approach," in *MPM*, 2011.
[12] J. Deantoni and F. Mallet, "ECL: the Event Constraint Language, an Extension of OCL with Events," INRIA, Tech. Rep. RR-8031, 2012.
[13] OMG, *Object Constraint Language*, 2014, Version 2.4.
[14] G. D. Plotkin, "A structural approach to operational semantics," *Journal of Logic and Algebraic Programming*, 1981.
[15] J. Deantoni, P. Issa Diallo, C. Teodorov, J. Champeau, and B. Combemale, "Operational Semantics of the Model of Concurrency and Communication Language," Tech. Rep.