A user-oriented approach to integrate formal verification activity for DSML

Faiez Zalila, Xavier Crégut, Marc Pantel

A user-oriented approach to integrate formal verification activity for DSML

Faiez Zalila, Xavier Crégut, Marc Pantel
Université de Toulouse, IRIT – France
Email: fistname.lastname@enseeiht.fr

***Abstract*—Formal methods based verification activity of safety critical embedded systems has produced very promising results in the industrial context and raised the interest of system designers up to the application of these technologies in real size projects.**

However, these methods usually rely on specific verification oriented formal languages that most designers do not master. It is thus mandatory to embed the associated tools in automated verification toolchains that allow designers to rely on their usual domain-specific modeling languages (DSMLs) while enjoying the benefits of these powerful methods. We propose an approach which introduces different steps to integrate verification tasks for a new DSML and explains the interactions between concerned actors. This work is based on a metamodeling pattern for executable DSML that favors the definition of generative tools and thus eases the integration of tools for new DSMLs.

***Index Terms*—Verification, Formal methods, Domain specific modeling language (DSML), Model checking**

I. Introduction

Domain-Specific Modeling Languages (DSMLs) are a major asset in the development of complex systems. In particular, they are widely used in the early phases of the development of safety critical systems. In this context, model validation and verification (V&V) activities are key features to assess the conformance of the future system to its safety and liveness requirements. They require the introduction of an execution semantics for the DSMLs. It is usually provided as a mapping from the abstract syntax (metamodel) of the DSML to an existing semantic domain, generally a formal language, in order to reuse powerful tools (simulator or model-checker) available for this domain [1], [2].

One key issue is that system designers (DSML end-users) should not be required to have a strong knowledge on formal languages and associated tools. The challenge is thus to leverage formal tools so that the system designer has not to burden with formal aspects and to integrate them in traditional CASE tools, like the Eclipse platform. Model Driven Engineering (MDE) already provides means to define metamodels, static properties, textual and graphical syntaxes. What should be addressed is thus 1) provide the system designer with a user-friendly language to formalize system requirements, 2) define a translational semantics from the DSML to a formal language, 3) translate formal requirements into formal language logic formulae according to the translational semantics, and eventually, 4) bring back formal verification results back at the DSML level so that they are understandable by the system designer.

Our contribution is on the methodological side as we propose an approach to introduce the executability aspect for DSML. We propose different elements to generate a DSML verification framework: the selected DSML and formal language, define a translational semantics to map the DSML abstract syntax to a formal language, introduce a behavioral properties language to express DSML behavioral requirements, map these DSML behavioral properties into formal ones and define backward transformations to feedback verification results into the DSML level.

In addition, we propose different interactions between different actors in order to ease generating a DSML verification framework.

The paper is organized as follows. Section II presents different DSML verification framework elements (a translational semantics, a language to express behavioral properties and a backward transformation) which must be defined. Section III presents the steps to generate this DSML verification framework. Section IV introduces the use of this proposed verification framework by the DSML end-user. Finally, we conclude and presents future work in Section V.

II. Different DSML verification framework elements

When dealing with an executable DSML (xDSML), the usual metamodel generally only allows expressing and verifying static and structural requirements. However, it does not model the notions handled at behavioral verification and at runtime such as behavioral requirements, dynamic information or stimuli that make the model evolve. Figure 2 explains a set of structured elements that must be defined to guarantee the verification activity for a new DSML.

A. *The* DSML

It provides the key concepts of the language (representing the considered domain) and their relationships. It is the usual metamodel used to define the modeling language in standardization organisations. It is usually endowed with structural constraints. To illustrate this paper, we consider as a running example the SPEM process modeling language [3] (SPEM metamodel in Figure 2). It was designed in order to experiment V&V in the TopCased toolkit using an MDE approach. The classical DSML metamodel is shown in Figure 1. For instance, the SPEM metamodel defines the concepts of *Process* composed of (1) *workDefinitions* that model the activities

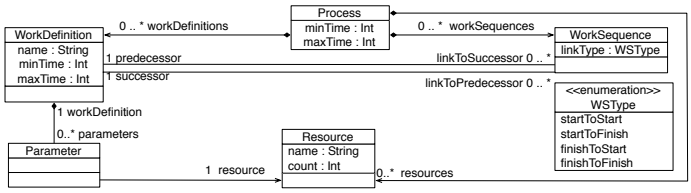

Figure 1. SPEM Metamodel

performed during the process, (2) *workSequences* that define temporal dependency relations (causality constraints) between activities.

B. The formal language

It is mandatory to choose the appropriate target tools in order to perform formal verification. The main advantage is to reuse tools available for this semantic domain like simulators or model-checkers. One common drawback is the semantic gap that may exist between the DSML and the semantics domain. To fill this gap, we target the FIACRE formal language [4] as it provides high level concepts.

FIACRE is a front end language to several verification toolboxes (TINA [5] and CADP [6] currently). This work focuses on the TINA toolbox. FIACRE [4] is a french acronym for an Intermediate Format for Embedded Distributed Components Architectures. It was designed as the target language for model transformations from different DSMLs such as AADL [7] or PLC [8].

FIACRE is a formal language (see Figure 2) to represent both the behavioral and timing aspects of systems, in particular embedded and distributed systems, for formal verification and simulation purposes. Fiacre is built around two notions:

- Processes describe the behavior of sequential components. A process is defined by a set of control states, each associated with a piece of program built from deterministic constructs available in classical programming languages (assignments, if-then-else conditionals, while loops, and sequential compositions), non deterministic constructs (non deterministic choice and non deterministic assignments), communication events on ports, and jumps to next state.
- Components describe the composition of processes, possibly in a hierarchical manner. A component is defined as a parallel composition of instantiated components and/or processes communicating through ports and shared variables. The notion of component also allows to restrict the access mode and visibility of shared variables and ports, to associate timing constraints with communications, and to define priority between communication events.

C. The translational semantics

One common way to verify a DSML consists in mapping its abstract syntax, defined by a metamodel, to a semantic domain [2]. It is named *Translational semantics* in Figure 2. It consists in generating from a DSML model (SPEM *Model* in

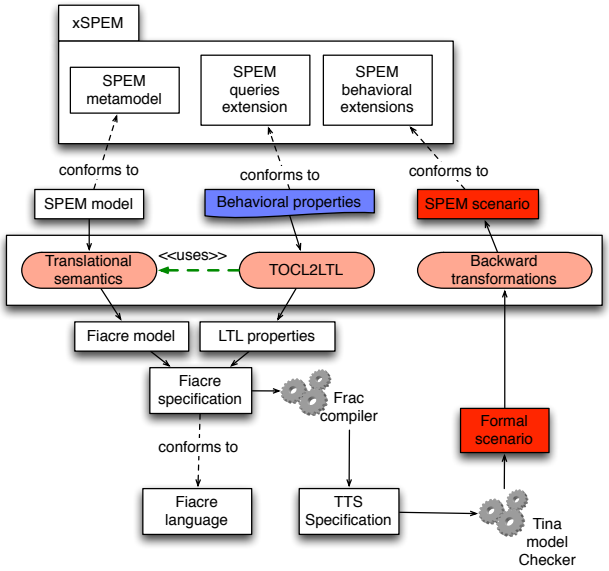

Figure 2. Different DSML verification framework elements

Figure 2) a corresponding formal model (FIACRE *Model*). For XSPEM, the translational semantics consists in transforming a XSPEM model into a FIACRE specification. It is performed with a model-to-model (M2M) transformation expressed in ATL [9] and then based on a defined XTEXT grammar, the FIACRE textual model is generated. Some rationale behind this translational semantics are explained in [10].

D. Behavioral properties language

The properties of interest for the XDSML end-user are behavioral properties relying on temporal operators. We have chosen to reuse the TOCL language [11]. TOCL is an extension of OCL that introduces usual future-oriented temporal operators such as *always*, *sometimes*, *next*, *existsNext* as well as their past-oriented duals.

One first step to formalize the *Behavioral properties* of interest to the DSML end-user is to analyze the properties in order to identify the queries of interest. It consists in introducing DSML requirements as queries in order to be used to query the model. This extension is shown as SPEM *queries extension* in Figure 2.

The key point is then to translate the properties as formulae on the formal model. Obviously, this translation is done at the metamodel level and thus has only to be written once for every DSML. As our purpose is to ease the development of CASE tools for new DSML, we focus on generic and generative approaches advocated by MDE.

We have written a generic tool, *TOCL2LTL*, to translate a TOCL property expressed on the XDSML level to a Linear Temporal Logic (LTL) formulae. Technically, TOCL operators, including OCL ones, are translated in a first transformation that generates a second transformation which uses the produced formal model (from the translational semantics) to

generate the corresponding *LTL Properties*. These transformations have been written using the ATL [9] transformation language. Implementing all these queries is a kind of checklist that ensures that all aspects of interest for the DSML end-user are indeed modeled on the formal side. The FIACRE textual specification is enriched with these generated FIACRE properties.

The complete real-time FIACRE (RT-FIACRE) [12] specification (Fiacre specification in Figure 2) containing both the FIACRE model specification and the properties to check represents the verification entry point. It is translated by the FRAC compiler[1] (the FIACRE compiler for the TINA toolbox) into a Timed Transition Systems (TTS) specification, the accepted input by TINA toolbox.

E. The backward transformations

The *TTS Specification* is verified using SELT, the TINA model checker[2]. When the properties fail, SELT generates a *Formal Scenario*: a succession of Petri net transitions. The generated scenario — also named counter-example and verifications results — is not easy to understand for the DSML end-user. So, *Backward transformations* are defined to generate DSML verification results. They are composed of two transformations: the first one consists of translating the PETRINET scenario into a FIACRE one. It is illustrated in [13]. The second transformation should be implemented in order to feedback the FIACRE scenario into the DSML scenario. Different ways can be found: in [14], a novel approach based on Higher-Order transformations is defined. It consists in analysing and instrumenting the translational semantics expressed as a M2M transformation in order to produce another model transformation which automatizes the back propagation of verification results to the DSML end-user. A second proposed approach [10] based on traceability information consists of defining previously a traceability metamodel. A traceability model is generated during running the translational semantics. It will be used after by the backward transformation to generate the DSML scenario.

III. GENERATING THE DSML VERIFICATION FRAMEWORK

Performing verification activities for a DSML requires the collaboration of different actors. Figure 3 presents the use case diagram which defines interactions between them.

The DSML *Expert* chooses the DSML which meets his needs. In addition, the DSML *Expert* explains his needs for the verification activity. Which queries should be asked on models ? During the execution of a model, what additional data are needed for expressing the execution itself ? In coordination with the *Formal methods Expert*, they choose the appropriate formal methods and tools.

Therefore, the DSML *Designer* implements verification activity for this DSML. First, he defines the translational semantics which allows to map the abstract syntax of the DSML into the chosen formal language. Based on the proposed translational semantics, he implements DSML extensions and builds the DSML verification framework.

To assist the DSML designer in his work, we propose the use of the *Executable DSML pattern* which is extended in order to explain different behavioral verification elements.

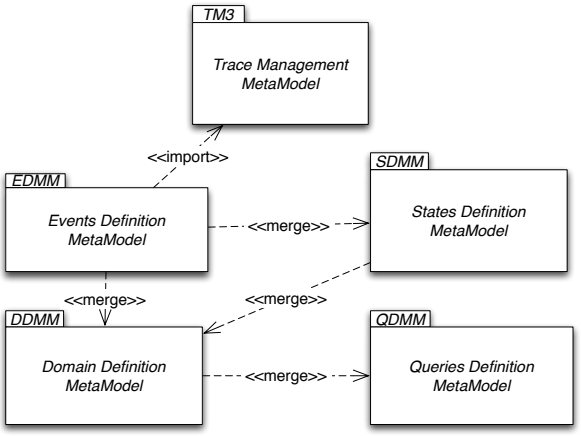

Figure 4. An abstract view of the *Executable DSML pattern* and its extension

A. The Executable DSML pattern and its extension

The *Executable DSML pattern* was proposed in [15] in order to define the behavioral verification and the related tools for a DSML. Figure 4 shows the abstract structure of a xDSML. The original metamodel of the DSML, called the DDMM (Domain Definition MetaModel) is extended with three other metamodels (Figure 4).

The first metamodel describes stimuli that make the model evolve. They are modeled as events in the EDMM (Event Definition MetaModel), middle left of Figure 4. The EDMM of a given DSML specifies the concrete stimuli (called runtime events) that drive the execution of a model that conforms to this DSML.

A second metamodel defines the runtime information, that is data that model the state of the model during the execution of the model and that are not part of the DDMM. This metamodel is called SDMM (State Definition MetaModel), middle right of Figure 4.

The third metamodel defines elements to model a scenario (either an input scenario or the trace of a particular execution) as a sequence of event occurrences. It is called TM3 (Trace Management MetaModel), top middle of Figure 4.

TM3 is not specific to one particular DSML as it only relies on the abstract *Event* concept. These two extensions allow to generate the scenario, which is a succession of events, that we want to feedback.

Figure 4 shows a fourth metamodel aside the three metamodels obtained by applying the *Executable DSML pattern* to a DSML. This additional metamodel is called QDMM (Queries Definition Metamodel), bottom right of Figure 4. It is a kind of an abstract view of the SDMM: it defines queries that may be asked on the model. SDMM and EDMM may be seen as a way to implement the QDMM by choosing a set

[1]http://projects.laas.fr/fiacre/manuals/frac.html
[2]http://projects.laas.fr/tina/manuals/selt.html

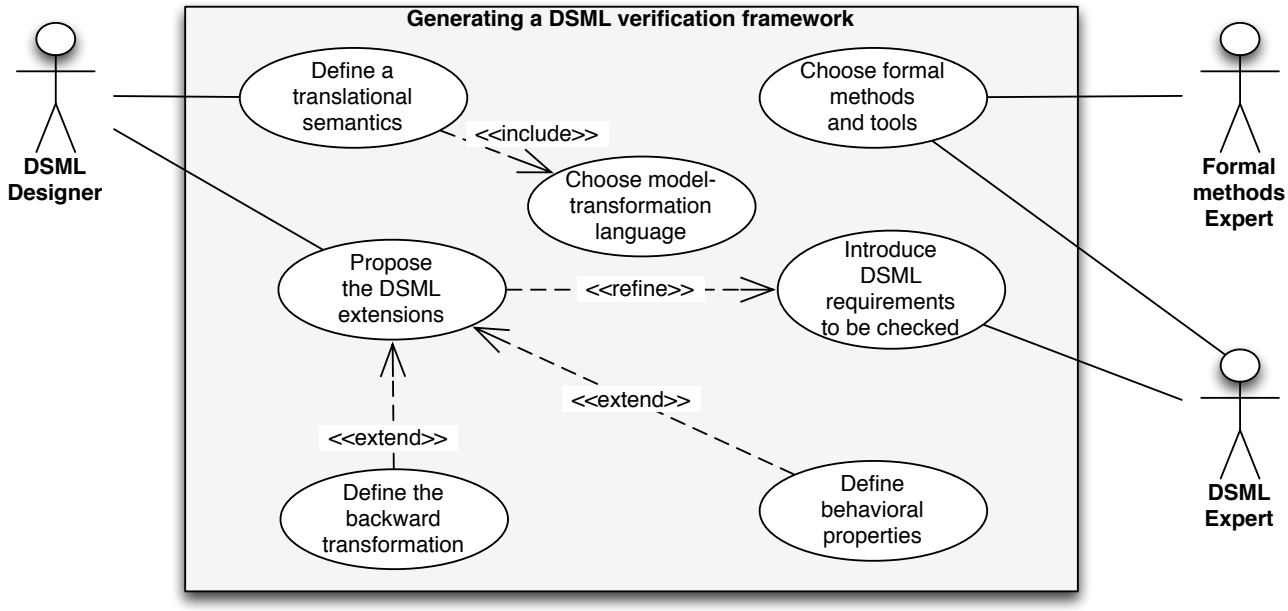

Figure 3. The generation of DSML verification framework

of attributes (like a Java class implements a Java inteface) and the elements that triggers the evolution of these attributes.

B. The process of DSML *verification framework generation*

The use case of DSML verification framework generation is shown in Figure 5. It shows the organisational process to generate the DSML verification framework and interactions between actors.

Once a DSML has been chosen to model a kind of system, the DSML expert expects to verify models; i.e. to check that properties derived from the system requirements hold on that model. He targets behavioral properties that concern the evolution of the model over time.

The DSML Expert may be interested in general properties not specific to a given process model. For SPEM, he may want to check whether a process model may finish. A process finishes if all its activities finish while respecting constraints imposed by dependencies and resource allocation. If this property holds, he may want to get a terminating scenario and use it to pilot the process execution.

In addition, the DSML end-user may also want to verify properties that are specific to a particular process model.

Next, in coordination with a formal methods expert, he chooses the appropriate formal languages and tools to guarantee the verification activity for this DSML. The chosen language should ease requirements expressing in order to verify behavioral properties. Time petri nets are chosen in previous works as the target formal language [16]. However, due to semantic gap between SPEM and time petri nets, the formal methods expert propose the use of FIACRE as the target formal language.

Once these steps are finalized, the DSML designer can start defining the translational semantics. It consists to map the abstract syntax elements of the DSML into FIACRE language elements.

This translational semantics should take in account the DSML expert needs. Different DSML expert requirements may be referenced in the target formal language. For example, in our case, the DSML designer chooses to encode a workdefinition as a FIACRE process with the same name. Based on the QDMM, a FIACRE type called WDQueries was defined to represent the two queries on WorkDefinition of interest for the XSPEM expert and also for causality constraints. It is a record type composed of the two boolean fields isStarted and isFinished.

An array of WDQueries named WDsQueries stores the state of all workdefinitions of an xSPEM process. It is an argument for every workdefinition process.

Based on the defined translational semantics, the DSML designer is able now to formalize DSML requirements. Therefore QDMM is defined.

Considering the properties the DSML expert wants to assess on XSPEM models, three queries *isStarted* and *isFinished* on WorkDefinition and *isFinished* on Process are defined.

isFinished on Process may be defined from the other ones. Here is its TOCL definition.

```
context Process
def: isFinished() : Boolean =
self.workDefinitions
    ->forAll(a:WorkDefinition| a.isFinished())
```

The following invariants state respectively that a process can always finish (Inv_1) and a process can never finish (Inv_2):

```
context Process   -- the invariant Inv1
```

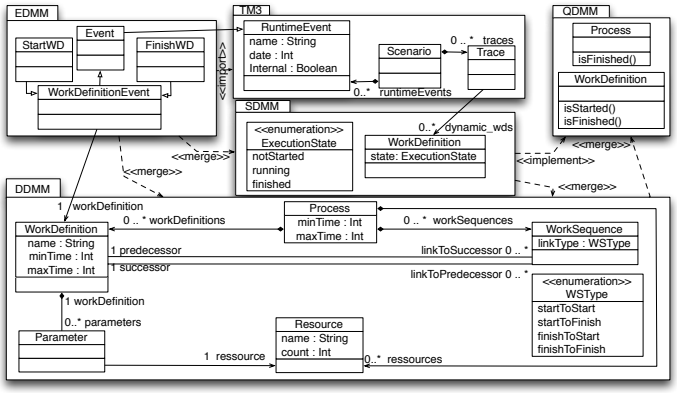

Figure 6. Executable DSML pattern applied into the SPEM metamodel

```
inv isEventuallyFinished:
  eventually (self.isFinished())

context Process  -- the invariant Inv2
inv isNeverFinished:
  always (not self.isFinished())
```

The queries on WorkDefinition are primitive. They should be defined using LTL fragments and based on the proposed translational semantics.

Using our proposed extension of OCL (TOCL) and FIACRE properties [17], primitives queries can be defined as following:

```
context WorkDefinition
def isFinished(): String =
self.getFiacreId() +
"/value wds[(" + self.name + "id)].isFinished"

context WorkDefinition
def isStarted(): String =
self.getFiacreId() +
"/value wds[(" + self.name + "id)].isStarted"
```

These primitive queries can then be used by the DSML end-user to express specific properties related to its model. The DSML designer can also define some additional OCL operations and TOCL queries in order to ease the DSML end-user work. The following OCL operation allows to select from a SPEM process a SPEM workdefinition whose its name is equal to a given name *WDName*:

```
context Process
def: getWD(WDName: String):
WorkDefinition =
self.workDefinitions
->select
(wd: WorkDefinition|wd.name = WDName)
->asList->first()
```

Once this extension is formalized, the DSML designer can complete the application of the *Executable DSML pattern* on its DSML in order to conduct the second part of the verification activity which is feedback verification results. So, SDMM and EDMM should de defined.

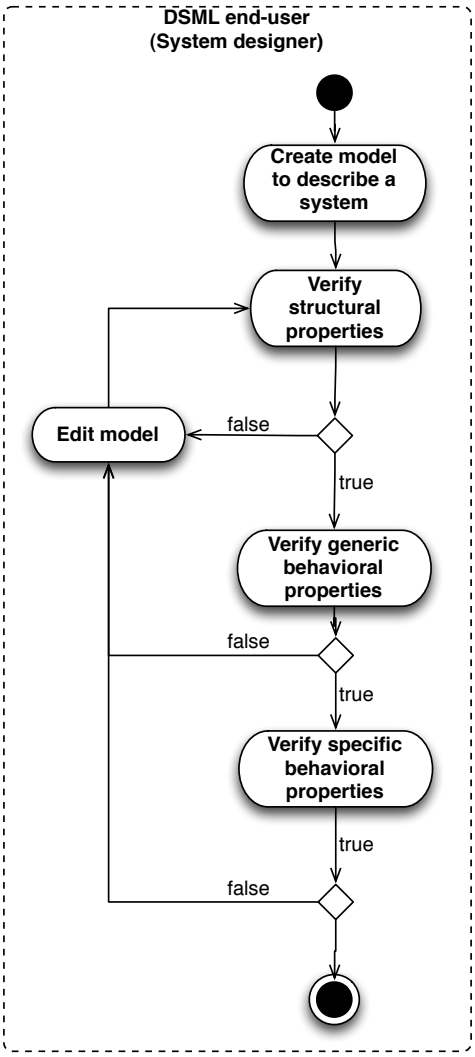

Figure 7. The use of the DSML verification framework

Start a WorkDefinition or *Finish a WorkDefinition* are examples of *Executable* SPEM (xSPEM) events. On the xSPEM example, the SDMM includes the achievement state of a workdefinition which is either *not started*, *running* or *finished*.

Therefore, the DSML *Designer* defines a backward transformation which allows to map formal generated verification results into DSML ones.

To ensure this step, the *Executable DSML pattern* is applied in both Petri nets [16] and FIACRE levels [13]. This feedback is made easier thanks to the *Executable DSML pattern* [15] applied not only at the DSML level but also at the formal one.

Now, we can consider that the DSML verification framework has been produced. It is composed of different tasks which guarantee the verification activity.

IV. THE USE OF THE DSML VERIFICATION FRAMEWORK

Once, the DSML verification framework is built, the DSML end-user is able to use it in order to verify defined

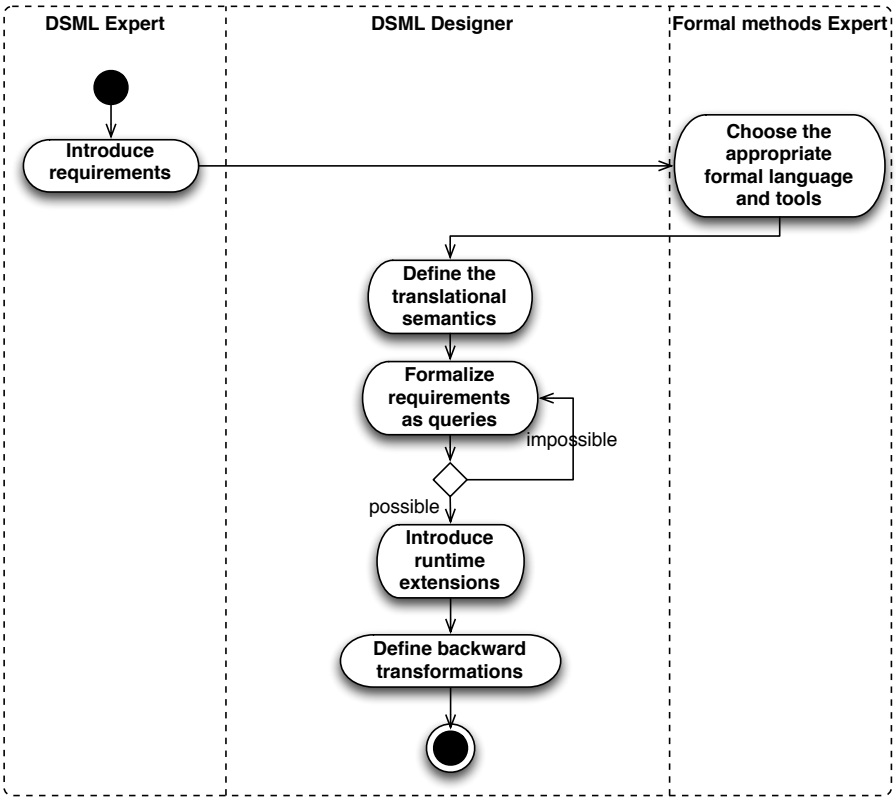

Figure 5. DSML verification framework generation process

models. Figure 8 explains different DSML end-user tasks to use the DSML verification framework. First, he proceeds to static properties which can be verified using an OCL checker. Next, he can verify generic behavioral properties and obtain user-friendly verification results. Another kind of properties can be targeted: model specific properties. i.e. properties specific to a given process model.

Based on Figure 7, we illustrate the use of our DSML verification framework.

First, the DSML end-user defines a SPEM model which is conform to our SPEM metamodel.

Figure 9 shows an example of a process model. It corresponds to a simplified development process composed of four activities, each represented in an ellipse: *Programming*, *Designing*, *Test case writing* and *Documenting*. Arrows between activities indicate dependencies: the target activity depends on the source activity. The label specifies the kind of dependency. The word before the "To" is the state that should have been reached by the source activity in order to perform the action on the target activity, action which appears after the "To". For example, the "finishToStart" dependency between *Designing* and *Programming* means that *Programming* can only be started when *Designing* has been finished. *Documenting* and *TestCaseWriting* can start once *Designing* is started (*startToStart*) but *Documenting* cannot finish if *Designing* is not finished (*finishToFinish*). The dependencies put between *Programming*

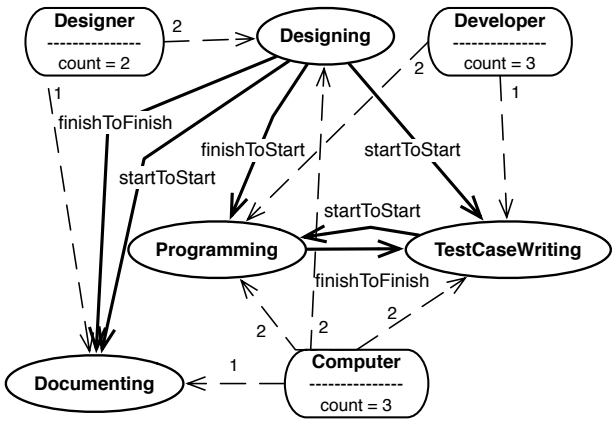

Figure 9. A SPEM development process

and *TestCaseWriting* enforces a test driven development: programming can only starts when test cases are already started and, obviously, test case writing can only be finished when programming is finished in order to take into account test coverage.

Rounded rectangles represent the number of available resources (2 *Designers*, 3 *Developers* and 3 *Computers*). Dashed arrows indicate how many resources an activity requires. *Programming* needs two developers and two computers. Re-

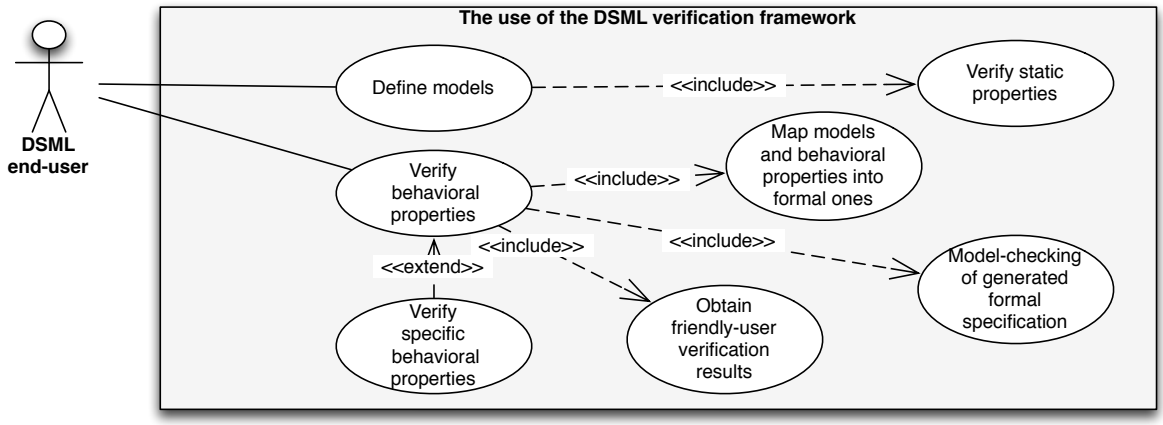

Figure 8. Different DSML verification framework uses

sources are allocated when an activity starts and freed when it finishes.

Once the model is defined, the DSML end-user can verify structural properties. They are defined to overcome the lack of expressivity of the graphical representation of metamodeling languages, like MOF. OCL is the OMG standard defined to request this kind of properties on models. For example, in SPEM metamodel, Listing 1 introduces an OCL property which verifies whether the workdefinitions names are unique on a process. In addition, the OCL property shown in the Listing 2 verifies the non-reflexivity of a worksequence.

```
context Process:
self.processElements
  ->select(pe|pe.oclIsTypeOf(SPEM!
      WorkDefinition)
    ->forAll(wd1, wd2|wd1 <> wd2
        implies wd1.name <> wd2.name)
```

Listing 1. OCL property to verify the uniqueness of workdefinitions names

```
context WorkDefinition:
self.predecessor <> self.successor
```

Listing 2. OCL property verifying the non-reflexivity of a worksequence

To assess these properties, the DSML end-user can use OCL checkers like for example TOPCASED[3] OCL checker.

For our SPEM model shown in Figure 9, all structural properties are verified. Therefore, the DSML end-user can assess the behavioral properties using the predefined invariants (Inv_1 and Inv_2).

So, If the Inv_2 (a process is never finished) is satisfied, it means the process cannot finish.

For our SPEM model shown in Figure 9, the Inv_2 is satisfied. Therefore, he can now refer to the Inv_1 which verifies whether the process can always be finished. The invariant Inv_1 does not hold and there is indeed a deadlock during process execution. The user can be provided with a counter-example that explains the deadlock as a scenario like the one of Listing 3 which lists the actions (start or finish) applied on activities.

```
Start Designing
Finish Designing
Start Documenting
Finish Documenting
Start TestCaseWriting
```

Listing 3. A scenario from Inv_1

The deadlock is due to the fact that *Programming* cannot be started because a *Computer* is missing. So, the SPEM model should be corrected. An occurrence of the *Computer* resource should be added.

Now, four occurrences of *Computer* resource are available. The DSML end-user will restart the verification activity. The verification of Inv_2 is not satisfied, it means the process can finish and the DSML end-user expects that a model checker would exhibit a counter example that corresponds to a scenario that finishes the process and thus all its activities. This scenario is shown in Listing 4.

```
Start Designing
Finish Designing
Start Documenting
Finish Documenting
Start TestCaseWriting
Start Programming
Finish Programming
Finish TestCaseWriting
```

Listing 4. A terminating scenario

The DSML end-user may also want to verify properties that are specific to a particular process model. As an example, he might want to check whether it is required that *Documenting* is finished before *TestCaseWriting* is finished. Based on our TOCL tool and using OCL operations and TOCL queries defined by the DSML designer, he can define the following invariant Inv_3:

[3]http://topcased.org/

```
context Process
inv Inv_3:
 always (
 self.getWD("Documenting").isFinished()
 precedes
 self.getWD("TestCaseWriting").isFinished()
);
```

The invariant Inv_3 does not hold and there is indeed a possible execution when *TestCaseWriting* can finish before *Documenting*. The generated counter-example is generated in the following Listing 5:

```
Start Designing
Finish Designing
Start TestCaseWriting
Start Programming
Finish Programming
Finish TestCaseWriting
Start Documenting
Finish Documenting
```

Listing 5. A terminating scenario

The use of the DSML verification framework shows that the DSML end-user performs behavioral verification without having to deal with the formal verification language nor with the underlying translational semantics. Based on the *Executable DSML pattern* and the provided tools, this DSML verification framework will provide seamless verification facilities to the system designer without requiring him to deal with target verification language and associated model-checkers.

V. Conclusion

We have presented an user-oriented approach to integrate verification tools on a new DSML in order to assist the DSML end-user into the verification of safety and liveness properties on executable models.

It has been illustrated using SPEM as DSML. We explained different communications between DSML verification framework actors. We introduced a user-friendly language, TOCL, to the DSML designer and the DSML end-user which allows to specify behavioral properties as it is close to OCL. However, the use of OCL and TOCL have shown that it is still not well suited to many system designers. Therefore, we might need to investigate a suited user-oriented language for expressing behavioral constraints like Dwyer patterns [18]. So, TOCL can be considered as an intermediate language between LTL and the high-level property language.

To ease feedback verification results, relying on the executable DSML pattern, we assist the DSML designer to define the backward transformation to feedback verification results at the DSML level. This approach has been designed for domain specific languages. It is currently being experimented for several significantly different DSMLs. But, it is still to be shown if it can scale up to more complex languages or to languages combining different models of computation.

As future works, we propose to further ease the DSML designer task by providing automatically the backward transformation which feedbacks verification results into the DSML level. It can be inspired from the previously defined translational semantics.

References

[1] J. Merilinna and J. Pärssinen, "Verification and validation in the context of domain-specific modelling," in *Proceedings of the 10th Workshop on Domain-Specific Modeling*, ser. DSM '10. New York, NY, USA: ACM, 2010, pp. 9:1–9:6. [Online]. Available: http://doi.acm.org/10.1145/2060329.2060351

[2] D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of "Semantics"?" *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[3] *Software & Systems Process Engineering Metamodel (SPEM) 2.0* , Object Management Group, Inc., Oct. 2007.

[4] B. Berthomieu, J.-P. Bodeveix, M. Filali, P. Farail, P. Gaufillet, H. Garavel, and F. Lang, "FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment," in *ERTS'08*, Jan. 2008.

[5] B. Berthomieu, P.-O. Ribet, and F. Vernadat, "The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets," *Int. Journal of Production Research*, vol. 42, no. 14, pp. 2741–2756, 2004.

[6] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2010: A toolbox for the construction and analysis of distributed processes," in *TACAS*, 2011, pp. 372–387.

[7] T. Correa, L. Becker, J.-M. Farines, J.-P. Bodeveix, M. Filali, and F. Vernadat, "Supporting the Design of Safety Critical Systems Using AADL," in *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, March, pp. 331–336.

[8] J.-M. Farines, M. H. De Queiroz, V. De Rocha, A. M. Carpes, F. Vernadat, and X. Crégut, "A model-driven engineering approach to formal verification of PLC programs (regular paper)," in *Emerging Technologies and Factory Automation (ETFA), Toulouse, France*. IEEE, septembre 2011, pp. 1–8.

[9] F. Jouault and I. Kurtev, "Transforming Models with ATL," in *Proceedings of the Model Transformations in Practice Workshop at MoDELS*, ser. LNCS. Jamaica: Springer, 2005.

[10] F. Zalila, X. Crégut, and M. Pantel, "Formal verification integration approach for DSML," in *The ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*. Miami: Springer-Verlag, Sep. 2013.

[11] P. Ziemann and M. Gogolla, "An Extension of OCL with Temporal Logic," in *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, vol. TUM-I0208, Sep. 2002, pp. 53–62.

[12] S. Dal Zilio and N. Abid, "Real-time Extensions for the FIACRE modeling language," in *MoVep 2010, Summer School on Modelling and Verifying Parallel Processes*, Aachen, Allemagne, Jun. 2010, 6 pages FNRAE Quarteft. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00494617

[13] F. Zalila, X. Crégut, and M. Pantel, "Verification results feedback for FIACRE intermediate language," in *Conférence en Ingénierie du Logiciel (CIEL)*, Jun. 2012. [Online]. Available: http://gpl2012.irisa.fr/?q=node/31

[14] F. Zalila, X. Crégut, and M. Pantel, "A transformation-driven approach to automate feedback verification results," in *MEDI*, ser. Lecture Notes in Computer Science, A. Cuzzocrea and S. Maabout, Eds., vol. 8216. Springer, 2013, pp. 266–277.

[15] B. Combemale, X. Crégut, and M. Pantel, "A Design Pattern to Build Executable DSMLs and associated V&V tools (short paper)," in *Asia-Pacific Software Engineering Conference (APSEC), Hong Kong, China*, 2012.

[16] F. Zalila, X. Crégut, and M. Pantel, "Leveraging formal verification tools for DSML users: a process modeling case study," in *ISoLA*, 2012. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00720917

[17] N. Abid, S. Dal-Zilio, and D. L. Botlan, "A verified approach for checking real-time specification patterns," *CoRR*, vol. abs/1301.7531, 2013.

[18] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *Proceedings of the second workshop on Formal methods in software practice*, ser. FMSP '98. New York, NY, USA: ACM, 1998, pp. 7–15. [Online]. Available: http://doi.acm.org/10.1145/298595.298598