Formal Verification Integration Approach for DSML

Faiez Zalila, Xavier Crégut, Marc Pantel

Formal Verification Integration Approach for DSML*

Faiez Zalila, Xavier Crégut, and Marc Pantel

Université de Toulouse, IRIT – France
Email: firstname.lastname@enseeiht.fr

Authors' version

Abstract. The application of formal methods (especially, model checking and static analysis techniques) for the verification of safety critical embedded systems has produced very good results 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. More precisely, we propose a language to formally express system requirements and interpret verification results so that system designers (DSML end-users) avoid the burden of learning some formal verification technologies. Formal verification is achieved through translational semantics. 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.

Keywords: Domain specific modeling language, Formal verification, Model checking, Translational semantics, Traceability, Verification feedback

1 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

* This works was funded by the french Ministry of Industry through the ITEA2 project OPEES and the french ANR project GEMOC.

(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 solid 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) providing the system designer with a user-friendly language to formalize system requirements, 2) defining a translational semantics from the DSML to a formal language, 3) translating formal requirements into formal language logic formulae according to the translational semantics, and eventually, 4) bringing back formal verification results back at the DSML level so that they are understandable by the system designer.

Our contribution is on the tooling and methodological side as we propose an approach to integrate formal verification through model-checking for a DSML. We rely on the Executable DSML pattern [3] to define all concerns involved in the definition of DSML semantics. We have fully tooled the Temporal OCL (TOCL) language proposed by Gogolla et al. [4], including the expression of formal properties on a specific model and their translation to the logic formulae of the target language (Linear Temporal Logic (LTL) formulae at the moment). We define guidelines to validate the translational semantics to the formal domain. Finally, the feedback is largely automated thanks to mappings identified while defining the translation semantics.

To illustrate this paper, we consider as a running example the xSPEM executable extension of the SPEM process modeling language [5]. It was designed in order to experiment V&V in the TopCased toolkit using an MDE approach.

The paper is organized as follows. Section 2 presents different manipulated elements by the system designer (models to be verified, verification requests and expected verification results). Section 3 presents the work to be done at DSML level on the running use-case. Section 4 introduces the proposed verification methodology with a translational semantics of xSPEM into the Fiacre formal language [6]. Section 5 explains various steps in order to provide verification results from formal tools to the xSPEM level through Fiacre. Section 6 gives some related work in the

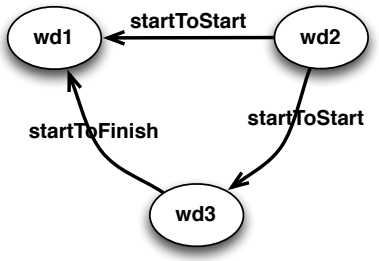

Fig. 1. a xSPEM model

```
Start  wd2
Start  wd1
Finish wd2
Start  wd3
Finish wd1
Finish wd3
```

Listing 1.1. A terminating scenario

domain of user level verification results. Finally, we conclude and presents future work in Section 7.

2 DSML end-user requirements

This section presents the domain – process modeling – considered in the case study and the requirements of system designers, the DSML end-users. We first present the kind of process models the DSML end-users want to build and the properties they want to check on their models. Finally, we describe the feedbacks expected from verification tools in order to get insights on the errors the models may contain.

2.1 DSML end-user models

Fig. 1 shows an example of a process model. It corresponds to a simplified development process composed of three activities, each represented in an ellipse: *wd1*, *wd2* and *wd3*. 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" indicates the state that must have been reached by the source activity in order to perform on the target activity the action, which appears after the "To". For example, the "startToFinish" dependency between *wd3* and *wd1* means that *wd1* can only be finished when *wd3* has been started. To keep this example simple, we have not represented the resources that are required to perform an activity.

2.2 DSML end-user verifications

To validate or to verify a model, the DSML end-user generally checks that properties derived from the system requirements hold on that model.

We focus on *behavioral properties*, that is properties that concern the evolution of the model over time.

The DSML end-user may be interested in general properties not specific to a given process model. For example, he may want to check whether a process model may finish (we call it P_1 requirement). A process finishes if all its activities finish while respecting constraints imposed by dependencies and resource allocation. If this property holds, the DSML end-users may want to get a terminating scenario and use it to pilot the process execution. Listing 1.1 is an example of terminating scenario for the model of Fig. 1.

Another kind of properties can be targeted which is specific properties. 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 *wd1* is finished before *wd2* is finished (we call it P_2 requirement).

2.3 Verification feedback

Once system designers have defined their models and formalized their requirements through properties, they want to have feedbacks on the assessment of those properties. Obviously, these feedbacks (named also counter-example or scenario) should be expressed at the domain-specific level.

For instance, using the example shown in Fig. 1, property P_1 holds and the process may finish. The DSML end-user can be provided with a scenario that describes a possible execution which leads to a finished process. Listing 1.1 is an example of such a terminating scenario. It lists actions (start or finish) applied on activities.

The DSML end-user will be able to play those scenarios using a model animator like the one developed in the TopCased project [7].

3 MDE for V&V CASE tools

MDE provides powerful techniques and tools to define a metamodel for the considered domain (using Ecore for example), completed with static properties (e.g. OCL) and to generate either textual syntactic editors (e.g. Xtext) or graphical editors (e.g. GMF). The metamodel of xSPEM is shown in Fig. 2. It defines the concepts of *Process* composed of (1) *WorkDefinition*s that model the activities performed during the process, (2) *WorkSequence*s that define temporal dependency relations (causality

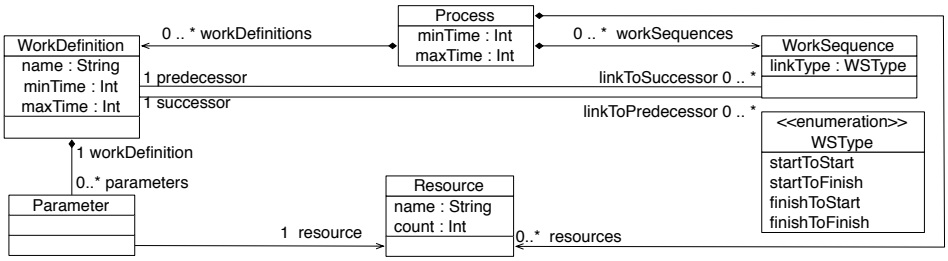

Fig. 2. An extract of the xSPEM Metamodel

constraints) between activities and (3) *Resource*s allocated to activities (*Parameter*).

The DSML end-user is thus able to design models and check whether static properties hold or not. Nevertheless, expressing properties which deal with the evolution of the model over time is not that easy because the metamodel does not usually provide all the required information. For instance, the xSPEM end-user wants to check whether workdefinitions may finish or not but the concept of "finished workdefinition" is not part of the xSPEM metamodel.

3.1 The *Executable DSML pattern*

As part of the TopCased [8] project, Combemale et al. have defined a metamodeling pattern called the *Executable DSML pattern* [3] that describes a way to define and structure the concerns required to make a DSML executable. The original metamodel, called the DDMM (Domain Definition MetaModel) is extended with three other metamodels (Fig. 3). The first metamodel describes stimuli that make the model evolve. They are modeled as events. *Start a WorkDefinition* or *Finish a WorkDefinition* are examples of xSPEM events. These events are modeled in the EDMM (Event Definition MetaModel), top left of Fig. 3. A second 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 Fig. 3. 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. The third metamodel defines the runtime information, that is data that model the state of the model at runtime and that are not part of the DDMM. This metamodel

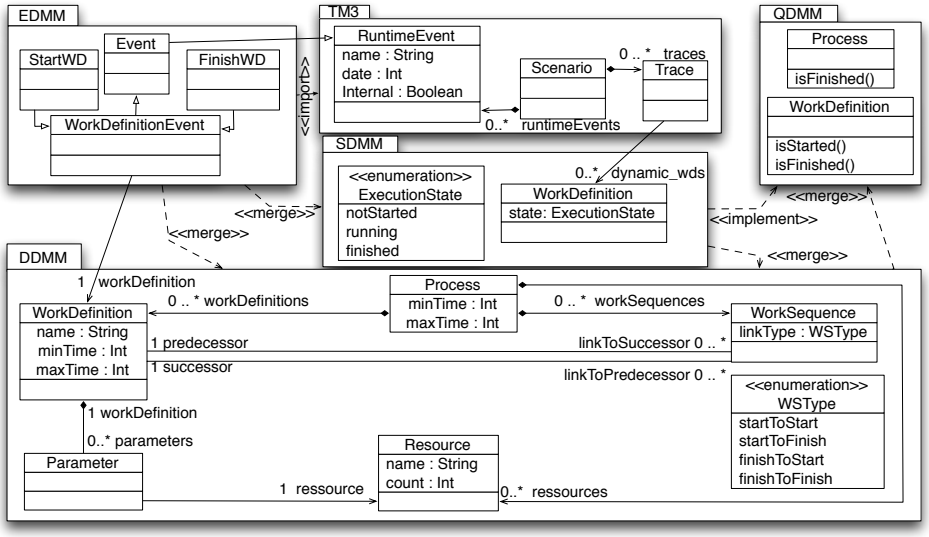

Fig. 3. xSPEM Metamodel

is called SDMM (State Definition MetaModel), middle of Fig. 3. On the xSPEM example, the SDMM includes the achievement state of a workdefinition which is either *not started*, *running* or *finished*.

Fig 3 shows a fourth metamodel aside the three metamodels obtained by applying the *Executable DSML pattern* to xSPEM. This additional metamodel is called QDMM (Query Definition Metamodel), top right of Fig. 3. It is a kind of an abstract view of the SDMM: it defines queries that may be asked on the model. SDMM may be seen as a way to implement the QDMM by choosing a set of attributes (like a Java class implements a Java inteface). For example, on Fig. 3, the SDMM of WorkDefinition defines an attribute *state* that can be used to implement the queries *isStarted* and *isFinished* from QDMM. Obviously, several SDMM are possible for one QDMM.

3.2 Formalizing behavioral properties

The properties of interest for the xSPEM end-user are behavioral properties relying on temporal operators. We have chosen to reuse the TOCL language [4]. 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 properties of interest to the DSML end-user is to analyze the properties in order to identify the queries of interest. The QDMM can then be defined. Considering the properties the DSML end-user wants to assess on xSPEM models, we have identified three queries *isStarted* and *isFinished* on WorkDefinition and *isFinished* on Process. The queries on WorkDefinition are primitive (as we are not able to evaluate them at the moment) whereas *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 property states that a process can never finish (it is the negation of the P_1 property):

```
context Process   -- negation of P1 requirement
inv isNeverFinished:
  always (not self.isFinished())
```

If this condition 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 would be obtained on the formal language used by the model checker and would have to be leveraged to the DSML end-user's domain.

We have built a TOCL syntactical editor integrated to the Eclipse platform. It has been defined using the Xtext tool[1].

4 Verification methodology

One common way to verify a DSML consists in mapping its abstract syntax, defined by a metamodel, to a semantic domain [2]. It is called a translational semantics. The main advantage is to reuse tools available on this semantic domain like simulators or model-checkers. One common drawback is the semantic gap that may exists between the DSML and the semantics domain. To fill this gap, we target the Fiacre formal language [6] because of its high level concepts. Fiacre is a front end language to several verification toolboxes (Tina [9] and Cadp [10] currently). This work focuses on the Tina toolbox.

Fig. 4 depicts the main steps and resources implied in the formal V&V of a DSML's model. The yellow part (top of the figure) shows resources manipulated by the DSML end-user: the model conforming to

[1] `http://www.eclipse.org/Xtext/`

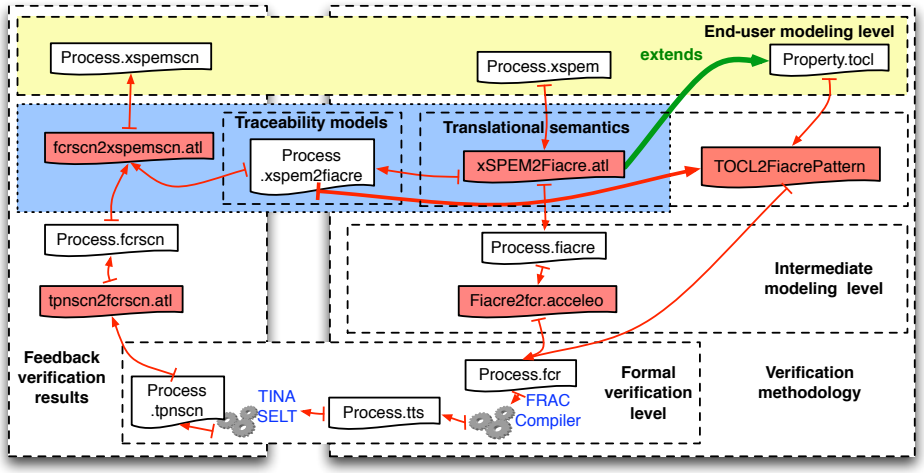

Fig. 4. General approach of DSML V&V

the DSML (Process.xspem), the behavioral requirements formalized using the TOCL editor (Property.tocl) as well as the scenario obtained when one property is not satisfied (Process.xspemscn).

The blue part depicts the DSML designer task. It consists of implementing a translational semantics from the DSML to the FIACRE formal language and, based on this semantics, a backward transformation in order to feedback verification results.

4.1 Fiacre Formal Language

FIACRE [6] 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 [11] or PLC [12].

FIACRE is a formal language 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 behavioral 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.

4.2 Translational semantics xSPEM2Fiacre

Translational semantics consists in defining the mapping from the DSML to the formal language.

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 [13] (xSPEM2FIACRE.atl at center of Fig. 4) and then an Acceleo [14] module generates the Fiacre textual syntax (named Fiacre2fcr.acceleo).

Here are some rationale behind this translational semantics. We illustrate it with some elements in the Fiacre program corresponding to the xSPEM example of Fig. 3.

Based on the QDMM, a Fiacre type called WDQueries was defined to represent the two queries on *WorkDefinition* of interest for the xSPEM end-user and also for causality constraints. It is a record type composed of the two boolean fields isStarted and isFinished.

```
type WDQueries is record        // from QDMM
        isStarted: bool,
        isFinished: bool
end
```

WDsQueries defines an array of WDQueries storing the state of all workdefinitions of an xSPEM process. It is an argument for every workdefinition process.

```
type WDsQueries is array 3 of WDQueries end
```

Named constants are defined to ease the reading of the Fiacre program by avoiding the use of meaningless integers to identify a workdefinition.

```
const wd1Id: int is 0
const wd2Id: int is 1
const wd3Id: int is 2
```

Each workdefinition is translated into one FIACRE process with the same name. Such a process is composed of three states (*notStarted*, *running* and *finished*) and two transitions (from *notStarted* to *running* and then from *running* to *finished*). It is parametrized by two ports (*Start* and *Finish*). They are mainly used to synchronize with resources used by the workdefinition (not presented in this paper) but also ease the identification of xSPEM events for the feedback.

Each transition includes an assignment to update variables which store the state of the activities. They were necessary to implement dependencies because a FIACRE process cannot inspect the current state of other processes.

xSPEM causality constraints are thus mapped into a FIACRE conditional statement that checks whether the FIACRE processes corresponding to the previous activities have reached the expected state. For example, because of the *start2Start* constraint between *wd2* and *wd1*, conditional statement checks whether activity *wd2* is started. If true the current state becomes *running* and it is recorded that this activity has been updated (was updated). Otherwise, nothing happens (loop statement). The following process shows the *wd1* workdefinition translated into FIACRE specification.

```
process wd1 [Start:sync, Finish:sync] (& wds: WDsQueries) is
    states notStarted, Running, finished
    from notStarted
    if (wds[$(wd2Id)].isStarted) then
        Start;
        wds[$(wd1Id)].isStarted := true;
        to Running
    else loop
    end if
    from Running
    if (wds[$(wd3Id)].isStarted) then
        Finish;
        wds[$(wd1Id)].isFinished := true;
        to finished
    else loop
    end if
```

The FIACRE component Process consists in instantiating the three processes *wd1*, *wd2* and *wd3* with the actual ports and the array that stores activities' states (initially all activities are not started and not finished):

```
component Process is
var
    wds: WDsQueries :=
        [ {isStarted=false, isFinished=false},
          {isStarted=false, isFinished=false},
          {isStarted=false, isFinished=false} ]
port
    wd1Start, wd1Finish: sync,
    wd2Start, wd2Finish: sync,
```

```
    wd3Start, wd3Finish: sync,
par * in
    wd1[wd1Start, wd1Finish](&wds)
    || wd2[wd2Start, wd2Finish](&wds)
    || wd3[wd3Start, wd3Finish](&wds)
end
```

4.3 Translating TOCL properties

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 facilitate the development of CASE tools for new DSML, we focus on generic and generative approaches advocated by MDE.

We have written a generic tool to translate a TOCL property expressed on the *xDSML* (using QDMM queries) to a LTL formulae on the formal language. Technically, TOCL operators, including OCL ones, are translated in a first transformation that generates a second transformation which handles queries from QDMM. These transformations have been written using the ATL transformation language. The second transformation only depends on the way primitive queries from QDMM are evaluated on the formal language. An ATL module must be provided to describe the LTL fragments that corresponds to the primitive queries of QDMM. According to the formal language, it may correspond to a process' state in a FIACRE model. Each query appears in that module as a helper method that returns the corresponding LTL fragment as a string. 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.

Here is the helper that corresponds to the primitive query *isFinished* identified on WorkDefinition in the context of xSPEM to FIACRE transformation.

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

The property body is built according to FIACRE properties [15]. A FIACRE property is composed of two elements[2]: a path and an observable. A path defines the context of applying the observable. For example, the *"Process/2/1"* path identifies the *first* instance in the *second* composition in the main component named *Process*. Observables play the role of

[2] `http://projects.laas.fr/fiacre/properties.html`

atomic proposition in the properties. It can be an instance state change, a communication through a port, a communication through shared variables or the execution of a transition.

The operation getFiacreId() is a helper method which consists of identifying the FIACRE instance – generated by the transformation – corresponding to the current workdefinition (*self*).

The second part in this query corresponds to the predicate to be verified, that is the observable. In the *isFinished()* definition, we check the shared variable wds that stores the state of each WorkDefinition instance.

Based on the translational semantics defined in section 4.2, the property P_1 applied on the the the xSPEM model of Fig. 1 generates the following FIACRE property.

```
property isNeverFinished is  ltl
( [] ( not (      Process/1/value wds[ $(wd1Id) ].isFinished
             and Process/2/value wds[ $(wd2Id) ].isFinished
             and Process/3/value wds[ $(wd3Id) ].isFinished )))
```

4.4 Guidelines for validating the translation semantics

Defining a translational semantics is a highly creative activity which requires high skills both in the formal language and in the DSML to find an efficient mapping between both languages as well as in transformation techniques. We thus only provide guidelines to favor the definition of a correct transformation.

The first guideline is the obligation to define for each QDMM primitive query the corresponding LTL fragment. QDMM queries are thus a kind of checklist that ensures that all aspects of interest for the DSML end-user have indeed been modeled on the formal side.

A second way to validate the translational semantics consists in formalizing invariants on the DSML using TOCL and then automatically translating them on the formal side. If they fail, an error is detected (either in the translation, the invariants or the queries implementations).

4.5 Formal verification

An ACCELEO[3] module generator (named Fiacre2fcr.acceleo) produces the FIACRE specification enriched with generated FIACRE properties.

The complete FIACRE specification (Process.fcr in the Fig. 4) containing both the FIACRE model specification and the properties to check represents the verification entry point shown in the **Formal verification**

[3] `http://www.acceleo.org/pages/home/en`

level part of Fig. 4. It is translated by the FRAC compiler[4] (the FIACRE compiler for the TINA toolbox) into a Timed Transition Systems (TTS) specification, the accepted input by TINA toolbox (Process.tts in Fig. 4).

This TTS specification is verified using SELT[5], the TINA model-checker for a State-Event version of LTL. When the property fails, SELT generates a counter-example as a succession of Petri net transitions. The generated counter-example — also named scenario and verifications results — is not easy to understand for the DSML end-user. So, we have to feedback it at the FIACRE level so that the DSML designer can use them to generate DSML verification results.

5 Feedback verification results

Verification results are obtained at the formal level and must be leveraged at the DSML level. This feedback is made easier thanks to the *Executable DSML pattern* [3] applied not only at the DSML level but also at the formal one. Results at the FIACRE level are obtained by analysing textual outputs of the TINA toolbox [16]. Xtext is used to parse textual outputs and model transformations generate the corresponding FIACRE events and scenarios.

FIACRE EDMM contains specific events [17]: an instance of a process entering or leaving a state, a variable changing value, a communication through a port.

In a previous work [18], we relied on the naming convention used when transforming the domain model to the formal one to translate verification results up to the DSML level. String analysis and parsing were used. However this method is tricky and cannot be applied on more complex DSMLs and cannot be generalized.

A more general solution consists in relying on a traceability metamodel which connects both metamodels (the DSML and the formal level). It corresponds to the traceability approach defined in [19]: trace information is considered as an additional model generated when the translational semantics is run.

5.1 DSML-Fiacre traceability links

Based on the *Executable DSML pattern* applied on each DSML and on FIACRE metamodel, the DSML designer is invited to define the trace-

[4] `http://projects.laas.fr/fiacre/manuals/frac.html`
[5] `http://projects.laas.fr/tina/manuals/selt.html`

ability metamodel with the appropriate information in order to capture information required to feedback verification results.

The traceability metamodel depends on the defined translational semantics and what kind of information would be traced back into the DSML level. Typically, this information consists of triggering DSML events into the formal language.

For the xSPEM example, two kinds of events are included *Start a WorkDefinition* and *Finish a WorkDefinition*. As shown in the previous section, the DSML designer has mapped a workdefinition into a FIACRE instance. Events are triggered using port signals (*Start* port and *Finish* port).

Fig. 5 shows the traceability metamodel, xSPEM2FIACRE, inspired from the translational semantics which links xSPEM metamodel (bottom) and FIACRE one (top).

The *xSPEM2Fiacre* model (shown in Fig. 4 as Process.xspem2fiacre) is conforms to this metamodel. To find back xSPEM events from FIACRE ones, we have defined two metaclasses *WDStart2Fiacre* and *WDFinish2Fiacre* that correspond to the two xSPEM events (start a workdefinition and finish a workdefinition). They are respectively linked to the *Start* and *Finish* port signal statements.

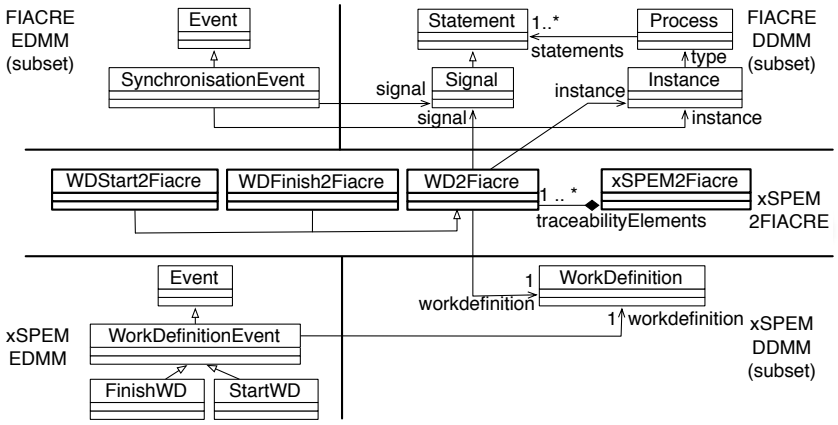

Fig. 5. Defining a traceability meta model

5.2 Feedback verification results at DSML level

The generated FIACRE scenario, Process.fcrscn in Fig. 4, (that only contains FIACRE events) has to be leveraged at the DSML level, xSPEM in our case. An xSPEM scenario only contains events which are instances of the xSPEM's EDMM. Obtaining xSPEM events is done from FIACRE events thanks to the traceability links generated while the translational semantics runs.

Fig. 5 shows the relations between the EDMMs of FIACRE and xSPEM on the one hand (left) and their DDMMs on the other hand (right) through the traceability metamodel (middle). Only the *Synchronisation-Event* is represented because other events are not used for xSPEM. According to the signal and the instance of this event, the corresponding element can be found in the traceability model, and then the workdefinition identified as well as the kind of xSPEM event — either start or finish that workdefinition. Applying our approach on xSPEM model shown in Fig. 1 and TOCL property, negation of P_1 requirement, constructs the scenario presented in the Listing 1.1.

6 Related Work

The problem of integrating formal verification into the design of DSMLs has been widely addressed by the MDE community. In order to tackle property-based verification problem, authors of [20] present the Metropolis design framework for embedded systems.

Their verification approach is based on formal properties specified in Linear Temporal Logic (LTL) and Logic of Constraints (LOC). They have different domains of expressiveness and indeed complement each other quite well. The formal verification methodology of Metropolis consists in translating the Metropolis specification into Promela description, and the LTL properties are checked using the model checker Spin. Translating verification results is done in ad hoc manner.

On the contrary, in our approach, we introduce for the DSML designer a user-friendly tool, TOCL, used to ease the writing of behavioral properties and which is also close to OCL. OCL is widely accepted as the appropriate language to verify structural properties on models.

In [21], authors define an approach named Arcade that uses SPIN model checker for evaluating safety and liveness properties of a Domain Reference Architecture that is translated to Promela language. Arcade interprets SPIN counter-example and generates an Architecture Trace Diagram (ATD).

Nevertheless, the ATD is a graphical representation of the spin counter-example. They do not define a high-level abstraction between model level and formal level. In our work, we separate the two domains (DSML and formal ones) and we hide all formal aspects by translating formal results to domain-specific results.

Hegedüs et al. [22] propose a method to verify BPEL models. It relies on a relation between elements of the source (BPEL) and the target (Petri nets) metamodels, implemented by means of annotations in the transformation's source code. The authors propose a technique for the back-annotation of simulation traces from traces generated by the model checker to the specific animator named BPEL Animation Controller. This approach is based on change-driven model transformations. This choice can be a restriction for DSML designers which are not familiarized with this specific model transformations technique.

In [23], authors introduce an algorithm requiring the DSML's semantics to be defined formally, and a relation R to be defined between states of the DSML and states of the target language. The DSML designer must provide as input a natural-number bound n, which estimates a difference of granularity between the semantics of the DSML and the semantics of the target language.

However, we don't think that DSML designer, for who it it difficult to use formal methods and verification, can define this important information to feedback verification results.

The most important difference between our approach and all the previously quoted approaches is on the fact that we are defining a structured model-based approach allowing to model different steps: defining the model using DDMM, introducing behavioral properties using a TOCL editor and a QDMM extension and capturing runtime information using TM3, EDMM and SDMM extensions.

7 Conclusion

We have presented an approach to integrate verification tools on a DSML in order to assist system designer into the verification of safety and liveness properties on executable models.

It has been illustrated on xSPEM as DSML and Fiacre as the formal language. We introduce a user-friendly language, TOCL, to system designer which allows to specify behavioral properties because 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 more suited user-oriented language for expressing behavioral constraints. 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 and traceability models, we assist DSML designer to define a traceability meta-model used after 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 facilitate 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. 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.
4. 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.
5. *Software & Systems Process Engineering Metamodel (SPEM) 2.0* , Object Management Group, Inc., Oct. 2007.
6. 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.
7. B. Combemale, X. Crégut, J.-P. Giacometti, P. Michel, and M. Pantel, "Introducing Simulation and Model Animation in the MDE TopCased Toolkit," in *Proceedings of the 4th European Congress* Embedded Real Time Software *(ERTS)*, Toulouse, France, Jan. 2008.
8. P. Farail, P. Gaufillet, A. Canals, C. L. Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel, "The TopCased project: a toolkit in open source for critical aeronautic systems design," in *Embedded Real Time Software (ERTS)*, Toulouse, France, January 2006.
9. 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.

10. 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.
11. 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.
12. 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.
13. 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.
14. Eclipse. (2012) Acceleo. [Online]. Available: http://www.eclipse.org/acceleo/
15. N. Abid, S. Dal-Zilio, and D. L. Botlan, "A verified approach for checking real-time specification patterns," *CoRR*, vol. abs/1301.7531, 2013.
16. F. Zalila, X. Crégut, and M. Pantel, "Verification results feedback for FIACRE intermediate language," in *Confrence en Ingnierie du Logiciel (CIEL)*, Jun. 2012. [Online]. Available: http://gpl2012.irisa.fr/?q=node/31
17. N. Abid and S. Dal Zilio, "Real-time Extensions for the FIACRE modeling language," pp. http://automata.rwth–aachen.de/movep2010/index.php?page=about, 2010. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00593958
18. 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
19. F. Jouault, "Loosely coupled traceability for ATL," in *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, 2005.
20. X. Chen, H. Hsieh, and F. Balarin, "Verification approach of metropolis design framework for embedded systems," *International Journal of Parallel Programming*, vol. 34, no. 1, pp. 3–27, 2006.
21. K. S. Barber, T. Graser, and J. Holt, "Providing early feedback in the development cycle through automated application of model checking to software architectures," in *Proceedings of the 16th IEEE international conference on ASE '01*, Washington, DC, USA, 2001.
22. Á. Hegedüs, G. Bergmann, I. Ráth, and D. Varró, "Back-annotation of simulation traces with change-driven model transformations," in *SEFM'10*, 2010, pp. 145–155.
23. B. Combemale, L. Gonnord, and V. Rusu, "A generic tool for tracing executions back to a DSML's operational semantics," in *ECMFA*, 2011, pp. 35–51.