Leveraging Formal Verification Tools for DSML Users: A Process Modeling Case Study

Faiez Zalila, Xavier Crégut, and Marc Pantel

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

Abstract. In the last decade, Model Driven Engineering (MDE) has been used to improve the development of safety critical systems by providing early Validation and Verification (V&V) tools for Domain Specific Modeling Languages (DSML). Verification of behavioral models is mainly addressed by translating domain specific models to formal verification dedicated languages in order to use the sophisticated associated tools such as model-checkers. This approach has been successfully applied in many different contexts, but it has a major drawback: the user has to interact with the formal tools. In this paper, we present an illustrated approach that allows the designer to formally express the expected behavioral properties using a user oriented language — a temporal extension of OCL —, that is automatically translated into the formal language; and then to get feedback from the assessment of these properties using its domain language without having to deal with the formal verification language nor with the underlying translational semantics. This work is based on the metamodeling pattern for executable DSML that extends the DSML metamodel to integrate concerns related to execution and behavior.

Keywords: Domain specific modeling languages, Model formal verification, Behavioral properties, Translational semantics, Verification feedback.

1 Introduction

TOPCASED[1] is a project[2] started in 2005 in the French "Aerospace Valley" cluster that gathers academic and industrial partners [1]. TOPCASED is dedicated to the development of an open source Computer Assisted Software Engineering (CASE) tool for the development of safety critical aeronautics, automotive and space embedded systems. Such developments will range from system and architecture specifications to software and hardware implementation through equipment definition.

TOPCASED provides modeling languages, both domain specific (SAM, EAST-ADL, SAE AADL, SDL[3] and XSPEM[4]) and general purpose (SYSML, UML, etc.) and

[1] *Toolkit In OPen source for Critical Applications & SystEms Development*, `www.topcased.org`

[2] This work was funded by the French ministries of Industry and Research and the Midi-Pyrénées regional authorities through the FUI TOPCASED, ANR OpenEmbedd, ITEA SPICES and ITEA2 OPEES projects.

[3] *Specification and Description Language: is an object-oriented formal language developed and standardized by The International Telecommunication Standardization Sector (ITU-T).*

[4] *OMG SPEM extended for execution.*

associated tools like graphical and textual editors, documentation generators, validation through model animation, verification through model checking, version management, traceability, etc. TOPCASED relies on MDE generative technologies to define the languages and build all these tools for all these languages. It is thus an MDE platform both for building system models and for building the platform itself. MDE technologies used in TOPCASED for defining and tooling languages are centered around Ecore[5] and configuration models taken as inputs by generative or interpretative tools.

Because the TOPCASED toolkit addresses safety critical systems, Validation and Verification activities are of primary importance and should be performed as early as possible at design time on the various models, both to reduce the development costs and to provide higher quality systems.

Validation is performed through model animation [2]: the designer builds a model using a graphical editor and can execute it according to scenarios. The runtime data produced by these executions is displayed as decorations of the graphical representation of the model or thanks to a dedicated view. Model animation is thus very similar to source level debugging for software. Scenario driven model execution runs through a single path in the set of all possible executions for the model. The use of several scenarios provides a coverage of the various possible executions but this validation is usually not exhaustive.

On the contrary, verification aims to check whether a property holds for all possible executions of the model. Model-checkers are dedicated tools for that purpose. These tools usually rely on two formal verification languages: one to model the behavior of the system and one to express the properties to check. For example, the TINA toolbox [3], available in TOPCASED, relies on Time Petri nets (TPN) for the behavior and State-Event Linear Temporal Logic (SE-LTL) for the properties. Thus, the use of such model checking tools requires to translate the system business domain model into an equivalent behavior model in the considered formal verification language and to express the system requirements as properties. Furthermore, results are obtained on the formal side as execution traces and have to be translated back into the system domain. This is a well-known technique called *translational semantics*. Nevertheless, even if the translations are automated, they are often defined in an ad hoc way, specific to the considered business domain. Furthermore, system requirements are most of the time directly written as formal properties, in the verification tool domain and not in the system domain. Thus, the designer must have a good understanding of: a) the various domain languages; b) the behavior and property languages from the various tools; and c) on the translation scheme used to go from one to the other in both directions. Verification is thus a difficult activity requiring many abilities that are generally not available to the casual business domain designer. Our purpose is to provide methods and tools in order to ease the integration of model checking in MDE toolchain. This integration will provide seamless verification facilities to the business domain designer without requiring him to deal with target verification language and associated model-checkers. We will describe a partly automated MDE driven tool chain for expressing the system requirements in the business domain language, translating the requirements to model checking tools property languages, and translating the failure execution traces back to the designer's world.

[5] Ecore is the metalanguage of Eclipse Modeling Framework,
`www.eclipse.org/modeling/emf`

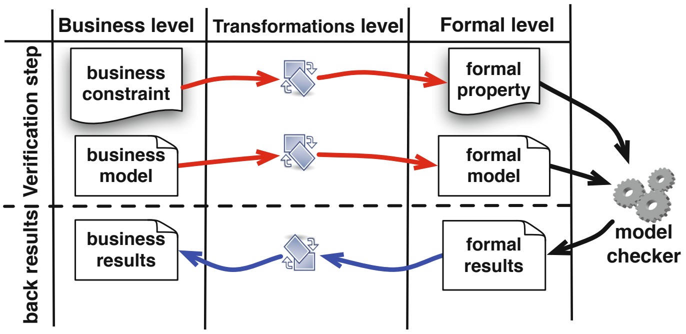

Fig. 1. General approach of a translational semantics with feedbacks

Our contribution consists in reifying elements involved in the semantics of a DSML in order to ease, and partially automate, the different translations that are summarized in Figure 1. More precisely, it includes:

1. the use of MDE technologies on both sides, business and formal verification domains based on a metamodel architecture that combines concerns related to model execution, including runtime information and the stimuli that make the model evolve. This metamodel guides the definition of the translational semantics and simplifies the production of business domain feedback to the end-user.
2. a user dedicated language for the expression of business properties.
3. automatic translation of business properties into formal verification domain properties based on the translational semantics.
4. automatic translation of the verification results obtained in the verification domain to the business domain. When a property does not hold, the obtained counter example is presented to the user either as a business domain scenario or a snapshot of the model completed with runtime information.

The approach is illustrated on a case study which concerns modeling of process using a process description language derived from the SPEM OMG Standard [4].

The paper is organized as follows. Section 2 presents the case study from the end-user viewpoint. It defines some constraints to assess and the expected feedback. Section 3 describes the formalism used for modeling processes, the language of expression of temporal constraints and extensions made on the DSML to be able to capture verification results. Section 4 presents the formal language and tools. Section 5 describes all required transformations for process verification and verification feedbacks. Section 6 considers related works and the last section concludes.

2 End-User Concerns

This section presents the business domain – process modelling – considered in the case study and the concerns of end-users. We first present the kind of process models the end-user wants to build. Then we explain the kind of properties he wants to check on his models. Finally we describe the feedback the end-user expects from verification tools in order to get insight on the errors the models may contain.

2.1 Business Models

Figure 2 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* 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.

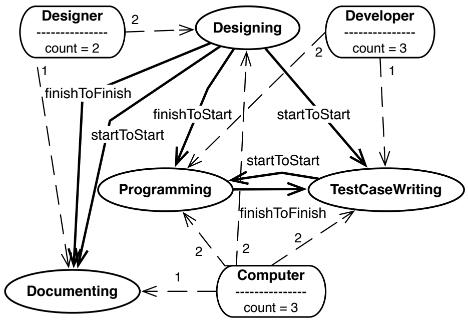

Fig. 2. A business development process

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

These processes are deliberately simplified to avoid overloading this presentation but time constraints or hierarchical decomposition of activities could be added.

2.2 User Verifications

To validate or to verify a model, the user may check that properties derived from the system requirements hold on that model. We focus on *behavioral properties*, properties that concern the evolution of the model over time. Static properties are also important for the end user but they can easily be included in the editing tool using for example an OCL checker.

The 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 or not (P_1). A process finishes if all its activities finish while respecting constraints imposed by dependencies and resource allocation. If these properties hold, the user may want to get a terminating scenario and use it to pilot the process execution.

The user may also want to verify properties that are specific to a particular process model. As an example, he might want to know if in all cases *Documenting* is finished before *Designing* is finished (P_2).

2.3 Verification Feedback

Once the end user has defined his model and expressed his requirements through properties, he wants to have feedback on the assessment of those properties. If a property evaluates to true, then the requirement is fulfilled. But if a property evaluates to false, the user expects to have feedback in order to understand why the property does not hold. For example, a counter example may be exhibited. Obviously, this counter example should be expressed at the business domain level.

For instance, using the example shown in Figure 2, property P_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 Figure 3 which lists the actions (start or finish) applied on activities. The deadlock is due to the fact that *Programming* cannot be started because a *Computer* is missing. If a computer was added, then the P_1 requirement would hold. The property P_2 does not hold. Indeed, it is possible to finish *Designing* before *Documenting* is finished. A possible scenario is shown on Figure 4 (counter-example). The user may want to play those scenarios using a model animator like the one developed in the TopCased project [5].

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

Fig. 3. A scenario from P_1

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

Fig. 4. A scenario from P_2

3 Business Metamodeling

Metamodels generally focus on business domain concerns and do not take into account other elements required to execute a model. As model execution may be of interest for most of the modeling languages, especially in the context of safety critical systems, we have defined a general solution to describe all the data required to define an execution semantics for any executable modeling language [6]. It is a kind of *metamodeling pattern* that may be used from design time to run time. This pattern has been applied in the TopCased project to build animators [2] that allow validation of SysML/UML State Machine and Activity diagrams or SAM (an automate-based language used by Airbus) models. It is also helpful to ease the definition of forward and backward transformations toward verification languages in order to get back failure scenarios from model checkers.

The pattern advocates to structure an executable DSML metamodel in such a way that the different concerns are stressed: the business domain, the queries a user may ask on a model to assess it satisfies its requirements (i.e. the model business properties), the stimuli that make the model evolve. The corresponding xSPEM metamodel is shown on Figure 5 and detailed in the next paragraphs.

3.1 xSPEM Domain Definition Metamodel (DDMM)

A metamodel defines the concepts (metaclasses) of the business domain addressed by the DSML and the relationships between them (references). In the executable metamodel pattern defined in TopCased, this reference metamodel is called the *Domain Definition MetaModel*, DDMM . The DDMM of xSPEM is shown on Figure 5 (package named DDMM at the bottom). It defines the concepts of process (*Process*) composed of a set of (1) workdefinitions (*WorkDefinition*) that model the activities (described in section 2.1) performed during the process, (2) worksequences (*WorkSequence*) that define dependency relationships between workdefinitions and (3) resources (*Resource*) allocated to activities (*Parameter*).

Obviously, this metamodel could be extended with well-formedness rules for example using OCL to express constraints not captured by the metamodel definition (names of workdefinitions have to be unique, worksequences should not be reflexive, resources counts should be positive, etc.). This aspect related to the static semantics of the DSML is not in the scope of this paper.

3.2 xSPEM Query Definition Metamodel (QDMM) and Formal Expression of Requirements

End users' behavioral properties usually rely on information that is not directly available in the DDMM because they only exist when the model is executed (runtime information). For example, the previous properties rely on the state of a workdefinition: started or finished. As usual, this information has to be reified. Thus, we have chosen to extend the DDMM with a new metamodel which describes the queries the user can conduct on his models. We call it QDMM (Query Definition MetaModel). For xSPEM, queries such as *isStarted* and *isFinished* can be applied on a *WorkDefinition* (see top right of Figure 5). A query *isFinished* is defined on *Process* in order to model the end-user business requirement, (P_1), defined in section 2.2.

To be checked, requirements of section 2.2 have to be formally expressed. OCL is not well suited for that purpose because it only allows the specification of structural properties and some Floyd-Hoare behavioral properties for methods. Nevertheless, it is now a widely known language and a few temporal extensions of OCL have been proposed in order to specify event-based behavioral properties whereas OCL only targets function-based properties. We have chosen to rely on Temporal OCL and especially on the proposal from [7] as the syntax of this extension is quite natural for OCL users. It introduces usual future-oriented temporal operators such as *always*, *sometimes*, *next*, *existsNext* as well as their past-oriented duals.

Here-after are the expression of the P_1 and P_2 requirements identified in section 2.2. They rely on the DDMM but also on the queries defined in the QDMM.

```
context Process      -- P1 requirement
inv isFinished:
    eventually (self.workDefinitions->
        forAll(a: WorkDefinition | a.isFinished())))
context Process      -- P2 requirement
    inv: always self.(getWD("Documenting").isFinished()
        precedes self.getWD("Designing").isFinished());
context Process
    def: getWD(WDName: String): WorkDefinition =
        self.workDefinitions
            ->select(wd: WorkDefinition | wd.name = WDName)
            ->asList->first()
```

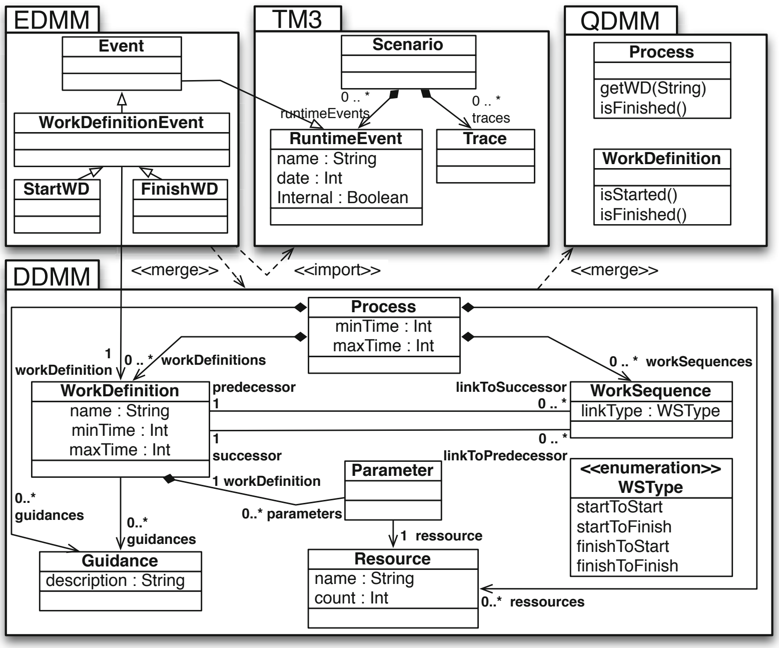

Fig. 5. XSPEM Metamodels

We have build a TOCL text editor dedicated to the end-user thanks to the xText tool from Eclipse Textual Modeling Framework (TMF) and the TOCL grammar of [7].

3.3 XSPEM Event Definition and Trace Management Metamodels (EDMM & TM3)

The DDMM and QDMM allow to express the requirements but we also need to show to the user the results obtained in the formal verification domain. A Snapshot can be expressed using QDMM because it represents all runtime information of interest to the user. To express the scenario corresponding to a counter example (like the one shown in Figure 4) we define two other metamodels that also extend the DDMM. The first one is the Trace Management Metamodel (TM3). It allows definition of a scenario as a sequence of runtime events — a stimulus that makes the model evolve. The TM3 is independent of any DSML. On the contrary, the Event Definition Metamodel (EDMM) is specific to a DSML and defines its runtime events. For instance, runtime events for XSPEM include “start a workdefinition” and “finish a workdefinition”.

4 Formal Level Metamodeling

In order to represent the semantic data and ease the exchange of verification results with business domain models, the metamodeling pattern is also applied on the formal language, TPN in the case of the TINA toolbox [3]. Like XSPEM, the TPN metamodel is composed of several parts (figure 6). The DDMM describes a Petri net (*PetriNet*) composed of nodes (*Node*) that denote places (*Place*) or transitions (*Transition*). Nodes are linked together by arcs (*Arc*). Arcs can be normal ones or read-arcs (*ArcKind*). The attribute *initialtokenCount* specifies the number of tokens consumed in the source node or produced in the target one (in case of a read-arc, it is only used to check whether the source place contains at least the specified number of tokens). Finally, a time interval can be expressed on transitions.

The QDMM defines only one query corresponding to the number of token stored in a place (`tokenCount`). We can define other queries like for example *fireableTransition* corresponding to the set of fireable transitions in a petri net. The SDMM (State Definition Metamodel) is an implementation of the QDMM, in this case a trivial implementation that defines an attribute for each query.

Finally, the EDMM defines only one event *FireTransitionEvent* and, obviously, the TM3 is the same as the one presented for XSPEM, as it is DSML-independent.

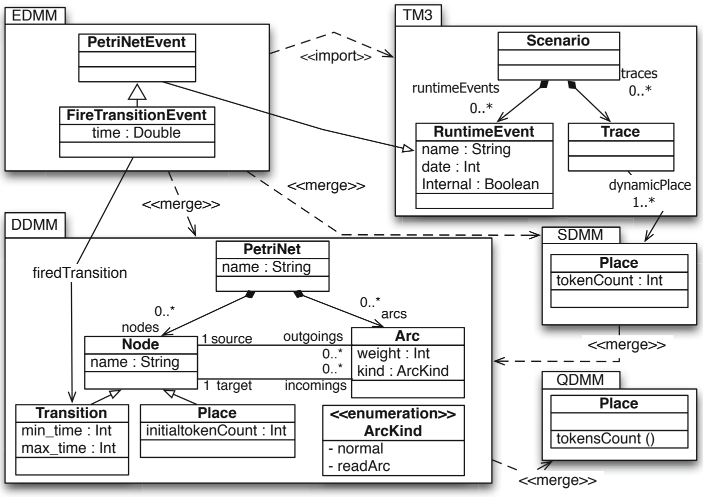

Fig. 6. PETRINET metamodels

5 The Transformation Level

The last steps concern the transformation level of Figure 1. It consists first in defining the translational semantics, that is translating an XSPEM model into a Petri net one. Then, TOCL properties have to be translated into LTL formulae so that they can be checked by the TINA toolbox. Finally, results obtained on the formal verification domain have to be translated back into the business domain. One can notice that business metamodel (or model) TOCL invariants may be expressed to assert that they are preserved by the translational semantics.

All these transformations are defined at the metamodel level. They have thus only to be defined once by DSML and formal domain experts and the end-user can use them for any of the XSPEM models.

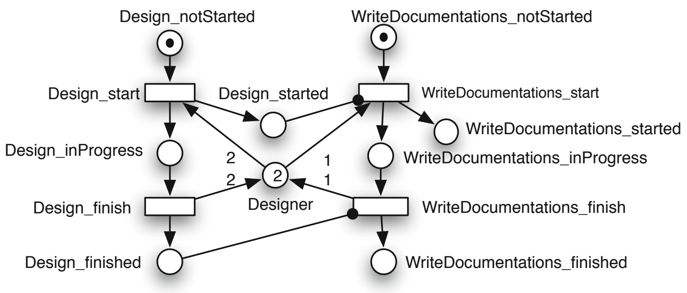
Design_notStarted
WriteDocumentations_notStarted
Design_start
Design_started
WriteDocumentations_start
WriteDocumentations_started
Design_inProgress
2
1
WriteDocumentations_inProgress
2
2
1
Design_finish
Designer
WriteDocumentations_finish
Design_finished
WriteDocumentations_finished

Fig. 7. Generated Petri net for parts of Figure 2

5.1 Translation XSPEM 2petrinet

Several translational semantics may be defined for XSPEM according to the level of details in the execution that we want to model and the kind of properties we want to assess. Thus, we advocate in [8] that defining the translational semantics should be *property-driven* to favor the definition of a *minimal* semantics, that will allow to answer to the questions the user may ask about his models.

As the QDMM metamodel records all the aspects of interest to the end user, the expert has to verify that all the queries may be expressed using the formal language translation. Thus, the QDMM can be used as a guide to write the translation. For example, in the formal language, one should be able to determine if a workdefinition is started or finished as these queries are part of the QDMM. A *WorkDefinition* is thus translated into four places characterizing its state (*notStarted*, *started*, *running* and *finished*) linked by two transitions. These transitions model the actions that we want to observe on a workdefinition: one can *start* a workdefinition and then *finish* it. A workdefinition is considered *started* if it is either running or finished. This is recorded by the place named *started*.

A*WorkSequence* becomes a *read-arc*[6] from one place of the source workdefinition (either *started* or *finished*) to a transition of the target workdefinition (either *start* or *finish*) according to the kind of *WorkSequence* (*linkKind* attribute). A resource becomes a place whose initial marking (*initialtokenCount*) corresponds to its *count*. Each *Parameter* element is translated into two arcs, the first one to take resources when the concerned workdefinition starts and the second one to release them when the workdefinition finishes.

Figure 7 contains the Petri net model resulting from the application of the translational semantics on a part of the XSPEM model from Figure 2 (*Designing* and *Documenting* workdefinitions and worksequences between them as well as the *Developer* resource).

The ATL transformation language [9] has been used to implement this translational semantics. First, an ATL module describes the transformation from an XSPEM model to a Petri net model [10] (not shown here[7]). Then, an ATL query[8] generates the textual

[6] A read-arc only checks that there is enough tokens in the input place but those tokens are not withdrawn when the transition is fired.

[7] `http://combemale.svn.enseeiht.fr/proto/fr.irit.acadie.xspem2tina/`

[8] This query is obviously independent of the translational semantics.

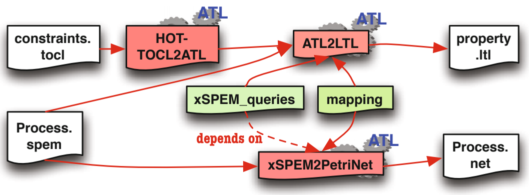

Fig. 8. Architecture of the ATL transformations

syntax used by the TINA tools from a Petri net model. This part could be implemented with any model to text language but it was simpler to rely on the same tool.

5.2 Translating TOCL to SE-LTL

Translating TOCL properties into LTL implies dealing with three main aspects. First, temporal operators have to be translated. It is straightforward since LTL provides the same kind of temporal operators. Second, OCL operators have to be evaluated, especially those that query the model. Finally, queries specified in XSPEM QDMM have to be translated into LTL. The meaning of these queries obviously depends on the previously defined translational semantics. Thus, while defining the translational semantics, the expert has to provide an ATL library called XSPEM_*queries* which defines all queries as helpers returning the appropriate LTL formulae. For example the WorkDefinition query "isFinished" becomes a helper "isFinished()" that computes the string "self.name + "_finished"".

The TOCL to LTL transformation is composed of two stages (top of Figure 8). The first one is a Higher Order Transformation (HOT-TOCL2ATL) that takes as input a TOCL model and generates an ATL transformation, which is the one executed in the second stage. This second transformation takes as input the business domain model (conforming to XSPEM in our case study) upon which the TOCL properties handled in the first stage will be run to generate the LTL formulae.

This transformation strategy results from two points. First, it is not possible in the first stage to use the XSPEM_queries library because there is no reflexivity in ATL. Second, several TOCL operators can be applied to each element of an input model. So, ATL iteration rules must be generated to traverse the model.

One strong point is that the TOCL2LTL transformation is generic and automated. It is only parametrized by the ATL module that provides the definitions of the QDMM queries for the target formal property language.

Applied on P_1, the TOCL2LTL transformation produces the following formulae:

```
<> (Designing_finished /\ Programming_finished
    /\ TestCaseWriting_finished /\ Documenting_finished)
```

The formulae corresponding to P_2 is:

```
[] ([] (- Designing_finished) U Documenting_finished)
```

5.3 Checking SE-LTL Properties

Once the Petri net and the LTL formulae have been generated, the `selt` model checker from the TINA toolbox is used. If the LTL formulae does not hold, it exhibits a counter-example: a specific execution of the model that leads to a state where the property is not satisfied. Figure 9 shows the counter example corresponding to the P_1 property. It consists of a sequence of states. A state is a snapshot of the model showing the places marking. After each state, there is the transition fired to go to the next state. The example shows a deadlock (last transition).

```
FALSE
  state 0: Programming_notStarted Designing_notStarted Documenting_notStarted TestCaseWriting_notStarted computer*3
        designer*2 developer*3
  -Designing_start->
  state 1: Programming_notStarted Designing_inProgress Designing_started Documenting_notStarted
        TestCaseWriting_notStarted computer developer*3
  -Designing_finish->
  state 2: Programming_notStarted Designing_finished Designing_started Documenting_notStarted
        TestCaseWriting_notStarted computer*3 designer*2 developer*3
  -Documenting_start->
  state 3: Programming_notStarted Designing_finished Designing_started Documenting_inProgress Documenting_started
        TestCaseWriting_notStarted computer*2 designer developer*3
  -Documenting_finish->
  state 4: Programming_notStarted Designing_finished Designing_started Documenting_finished Documenting_started
        TestCaseWriting_notStarted computer*3 designer*2 developer*3
  -TestCaseWriting_start->
  * [accepting] state 5: Programming_notStarted Designing_finished Designing_started Documenting_finished
        Documenting_started TestCaseWriting_inProgress TestCaseWriting_started computer designer*2 developer*2
  -deadlock->
  state 5: Programming_notStarted Designing_finished Designing_started Documenting_finished Documenting_started
        TestCaseWriting_inProgress TestCaseWriting_started computer designer*2 developer*2
  [accepting all]
0.001s
```

Fig. 9. Selt output for P_1 checked on example of Figure 2

5.4 Designer Dedicated Feedback

Model verification based on a translational semantics provides a significant advantage: the reuse of existing sophisticated model checkers. But there is one significant drawback: results are obtained at the verification level and have to be translated back to the business domain level. This section explains this translation.

Generating PETRINET Scenario and Trace. Using xText, we analyze the output of the `selt` model-checker and produce a PETRINET scenario and trace using the PETRINET metamodels and the TM3 presented in sections 3 and 4. The PETRINET scenario corresponding to the counter example of Figure 9 is the following:

```
   FireTransitionEvent Designing_start
   FireTransitionEvent Designing_finish
   FireTransitionEvent Documenting_start
   FireTransitionEvent Documenting_finish
   FireTransitionEvent TestCaseWriting_start
```

The same tool builds the PETRINET models that corresponds to each states of the counter-example (not shown here).

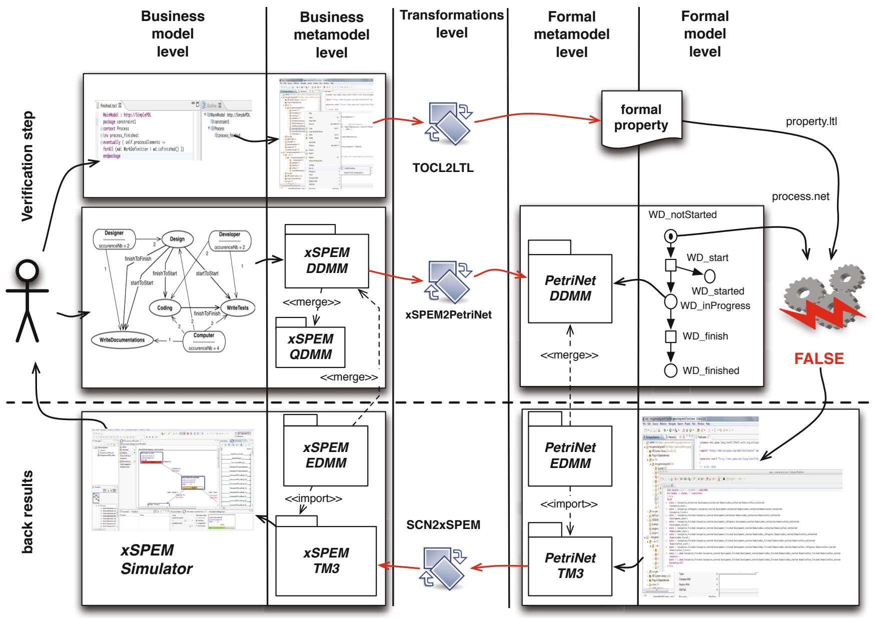

Fig. 10. General approach illustrated with the XSPEM case study

Translating PETRINET Scenario to XSPEM Scenario. The PETRINET Scenario is then transformed to an XSPEM Scenario. This transformation converts transition firing events *FireTransitionEvent* to XSPEM events, either start (*StartWD*) or finish (*FinishWD*) a *WorkDefinition*. The naming convention defined in the *Mapping* ATL library are used to decode the fired transition names and produce the corresponding XSPEM events and their target workdefinitions. The XSPEM scenario corresponding to the previous PETRINET one is the one shown on Figure 3. To obtain the scenario of figure 4, we have to check the negation of P_2 because we want an example which satisfies P_2.

The same approach is used to translate PETRINET snapshots (that is PETRINET models) into XSPEM ones. It could be automated using traceability data between source and target models generated during the transformation step.

6 Related Work

Translational semantics with feedbacking verification results: The main advantage of translational semantics is the reuse of existing tools from the target technical space like model-checkers. Its major issue is that it provides results in the target space that must be translated back to the business domain space. Hegedus et al. [11] propose a method based on a traceability mechanism of model transformations. It relies on a relation between elements of the source (BPEL) and the target (PETRINET) metamodel, implemented by means of annotations in the transformation's source code. The authors propose a technique for the back-annotation of simulation traces based on change-driven model transformations from traces generated by SPIN model checker to the specific animator named BPEL Animation Controller. However, in our approach, we try to generate a scenario (a set of events) that will be animated by a generic animator. In [12], authors use traceability links of the transformation which generates Alloy models from UML. The back-annotation transformation is automatically generated based on these traceability links using a QVT-based implementation. Here, the back-annotation is supported for static model instances, and not for execution traces of Executable DSML models like in our case. In [13], the 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) that has two dimensions: a vertical dimension that represents time and a horizontal dimension representing SPIN processes. Nevertheless, they do not define a high-level abstraction between business level and formal level. Contrary to our work, we separate the two domains (DSML and formal verification ones) and we hide all formal aspects by translating formal results to business ones. In [14], Pelliccione *et al.* present a software tool platform for the model-based design and validation of software architectures, named CHARMY, that offers an extension called SASIM deriving from Theseus approach [15]. Both translate the violation trace from SPIN model checker on a generated sequence diagram and an animated UML state diagrams. vUML [16] also use the same approach. CHARMY, Theseus and vUML are based on a very ad hoc approach that uses UML diagrams and SPIN model checker. On the contrary, we rely on a generic approach that can be applied to other DSML.

All the above approaches aim at verifying a specific DSML through formal tools by translating business semantics into formal one and by feedbacking formal verification results to the initial business level. However, our work provides a generic approach for the verification of executable DSML. It is based on the explicit definition of the different concerns involved in model execution (runtime information expressed as queries, events) thanks to the executable DSML metamodelling pattern. Based on this pattern and the translation semantics, generic transformations allow to translate user properties to logical formulae and verification results back to business level.

Behavioral property Patterns: To verify BPEL service composition schemas, [17] proposes a property specification language based on ontologies and named PROPOLS which allows composition of the patterns defined in [18]. These patterns are close to TOCL temporal operators and composition corresponds to OCL operators. Rather than relying on a Query Definition MetaModel, a one to one mapping has to be defined for each property item to the corresponding BPEL operation.

In [19], the authors provide a graphical tool named PSC (Property Sequence Chart), to specify temporal properties as an extended notation of a selected subset of the UML 2.0 Interaction Sequence Diagrams. Theseus approach [15] uses SPIDER to translate natural language properties to the property specification language of the targeted analysis tools. Both, PSC and SPIDER are specific approaches used in a specific domain between UML and SPIN. User-oriented property languages, graphical or not, are an important point to make formal verification accessible to end users. TOCL is certainly not the best-suited language despite it is an extension of OCL, a well accepted language in MDE. Nevertheless, we consider it can be used as a pivot language for more user-oriented languages.

7 Conclusion

Using the XSPEM case study, this paper has illustrated a method to ease the integration of verification tools for safety and liveness properties on executable models. It relies on the executable DSML metamodeling pattern using a translation to the Time Petri nets as formal verification language providing the semantics. This could be applied to any other kind of formal language providing automated verification tools. We have recently applied it to the FIACRE intermediate verification language [20], that abstracts several existing verification toolsets such as TINA and CADP in order to factorize common aspects and avoid redefining transformations for all toolsets. This experiment results will be presented in a forthcoming paper. The integration is provided through QDMM extension to the pattern and automated translations on the property side.

This approach has been designed for domain specific languages and this is a key point to keep it simple. It is currently being experimented for several significantly different DSMLs (and sub-languages from general purpose languages) such as data flow models, SAE AADL, SDL, UML and SYSML class, state machine, activity and composite structure diagrams. But, it is still to be shown if it can scale up to more complex languages or to languages that combine different models of computation.

These preliminary experiments allowed a first validation of our proposal for the systematic construction of verification tools for behavioral properties expressed on a DSML. We have chosen to rely on TOCL to express properties at the business domain level because it is close to OCL. However, some early feedback have shown that it is still not well suited to many end users. Therefore, we might need to investigate new user-oriented language for expressing behavioral constraints. It is a problem that has already been identified in [21]: the authors have defined a new dialect of linear temporal logic more suitable for control engineers.

Finally, we propose to ease the feedback of verification results. We currently rely on naming conventions. We are investigating the explicit construction of the links between the business domain and verification models elements during the downward translation so that they can be used during the upward feedback.

References

1. Farail, P., Gaufillet, P., Canals, A., Camus, C.L., Sciamma, D., Michel, P., Crégut, X., Pantel, M.: The TOPCASED project: a toolkit in open source for critical aeronautic systems design. In: Embedded Real Time Software (ERTS), Toulouse, France (January 2006)
2. Crégut, X., Combemale, B., Pantel, M., Faudoux, R., Pavei, J.: Generative Technologies for Model Animation in the TOPCASED Platform. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 90–103. Springer, Heidelberg (2010)

3. Berthomieu, B., Ribet, P.-O., Vernadat, F.: The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. International Journal of Production Research 42(14), 2741–2756 (2004)
4. Software & Systems Process Engineering Metamodel (SPEM) 2.0, Object Management Group, Inc. (October 2007)
5. Combemale, B., Crégut, X., Giacometti, J.-P., Michel, P., Pantel, M.: Introducing Simulation and Model Animation in the MDE TOPCASED Toolkit. In: Proceedings of the 4th European Congress EMBEDDED REAL TIME SOFTWARE (ERTS), Toulouse, France (2008)
6. Combemale, B.: Simulation et vérification de modèle par métamodélisation exécutable, E. U. Européennes, ed. (June 2010)
7. Ziemann, P., Gogolla, M.: An Extension of OCL with Temporal Logic. In: Jürjens, J., Cengarle, M.-V., Fernandez, E., Rumpe, B., Sandner, R. (eds.) Critical Systems Development with UML – Proceedings of the UML 2002 Workshop, vol. TUM-I0208, pp. 53–62, Université Technique de Munich, Institut d'Informatique (September 2002)
8. Combemale, B., Crégut, X., Garoche, P.-L., Thirioux, X., Vernadat, F.: A property-driven approach to formal verification of process models. In: ICEIS, Selected Papers (2007)
9. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
10. Bendraou, R., Combemale, B., Crégut, X., Gervais, M.-P.: Definition of an executable spem 2.0. In: APSEC, pp. 390–397. IEEE Computer Society (2007)
11. Hegedus, A., Bergmann, G., Rath, I., Varro, D.: Back-annotation of simulation traces with change-driven model transformations. In: IEEE International Conference on Software Engineering and Formal Methods, vol. 0, pp. 145–155 (2010)
12. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation (MODEVVA 2009). ACM International Conference Proceeding Series (2009)
13. Barber, K.S., Graser, T., Holt, J.: 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 Automated Software Engineering, ASE 2001 (2001)
14. Pelliccione, P., Inverardi, P., Muccini, H.: Charmy: A framework for designing and verifying architectural specifications. IEEE Trans. Soft. Eng. 35(3), 325–346 (2009)
15. Goldsby, H.J., Cheng, B.H.C., Konrad, S., Kamdoum, S.: A Visualization Framework for the Modeling and Formal Analysis of High Assurance Systems. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 707–721. Springer, Heidelberg (2006)
16. Lilius, J., Paltor, I.: vuml: a tool for verifying uml models. In: 14th IEEE International Conference on Automated Software Engineering, pp. 255–258 (October 1999)
17. Yu, J., Manh, T., Han, J., Jin, Y., Han, Y., Wang, J.: Pattern Based Property Specification and Verification for Service Composition, pp. 156–168 (2006)
18. Dwyer, M., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of the Second Workshop on Formal Methods in Software Practice, pp. 7–15. ACM Press (1998)
19. Autili, M., Inverardi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Automated Software Engg. 14, 293–340 (2007)
20. Berthomieu, B., Bodeveix, J.-P., Filali, M., Farail, P., Gaufillet, P., Garavel, H., Lang, F.: FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment. In: 4th European Congress EMBEDDED REAL TIME SOFTWARE (ERTS) (January 2008)
21. Ljungkrantz, O., Akesson, K., Fabian, M., Yuan, C.: A formal specification language for plc-based control logic. In: 2010 8th IEEE International Conference on Industrial Informatics (INDIN), pp. 1067–1072 (July 2010)