



HAL
open science

A DSL to Feedback Formal Verification Results

Faiez Zalila, Xavier Crégut, Marc Pantel

► **To cite this version:**

Faiez Zalila, Xavier Crégut, Marc Pantel. A DSL to Feedback Formal Verification Results. 13th Model-Driven Engineering, Verification and Validation Workshop at MODELS conference 2016 (MoDeVVa 2016), Oct 2016, Saint Malo, France. pp.30–39. hal-03172263

HAL Id: hal-03172263

<https://hal.science/hal-03172263v1>

Submitted on 18 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

A DSL to feedback formal verification results

Faiez Zalila*, Xavier Crégut†, Marc Pantel†

*IRT Saint-Exupéry, Toulouse, France
{faiez.zalila}@irt-saintexupery.com

†University of Toulouse, IRIT-CNRS
2 Rue Charles Camichel, Toulouse, France
{Xavier.Cregut, Marc.Pantel}@enseeiht.fr

Abstract—The integration of early formal validation and verification (V&V) tools (model checking, static analysis, etc.) in the V&V activities for domain-specific modeling languages (DSMLs) is a key asset to improve safety and reduce development and maintenance costs. However, system designers (DSMLs end-users) expect a seamless approach embedding transparently these tools in automated toolchains while enjoying their benefits. Thus, a mandatory task for DSML designer is to feedback at the DSML level the verification results generated by these tools. This contribution highlights a domain-specific language (DSL) to describe this feedback and the associated tools that helps the DSML designer in integrating the V&V tools. A translational semantics is given — as a higher-order transformation — for this DSL in order to automatically generate a model transformation which builds verification results at the DSML level from the ones produced at the formal level.

I. INTRODUCTION

Domain Specific Modeling Languages (DSMLs) are increasingly used at the early phases of the development of complex systems, in particular, for safety critical systems. The goal is to be able to conduct early verification and validation (V&V) activities to fulfill the associated safety objectives and to reduce the development costs. A widely used technique is model checking that conduct the exhaustive model behavior exploration. It relies on translational semantics to build formal models from DSML ones in order to reuse powerful tools available in the formal domain.

We contributed to MoDELS 2013 [23] a model-based method to integrate formal verification for a DSML composed of three parts: 1) a language to express behavioral properties for DSML models, 2) a translational semantics to map constructs of the DSML to a formal language, 3) an automated translation of the behavioral properties to the formal language properties. The model checkers defined for the formal models can then be used to assess the expected DSML properties and synthesize counter-examples. The proposal was illustrated with the verification of software processes modelled using the SPEM (Software & Systems Process Engineering Metamodel) modeling language [19] with model-checkers for the FIACRE formal language [2]. Its generality has been validated using other DSMLs like Ladder Diagram (LD) and AADL and other formal languages like Timed Petri Net [21].

Its main drawback is that verification results are obtained at the formal language level. For V&V to be more broadly used by system designers, these formal results have to be leveraged

to the DSML level. The challenge is to provide a seamless and transparent integration in an automated tool chain so that system designers can benefit from their power.

Several works have tried to ease the feedback of verification results at the design level but usually these ones are mostly ad-hoc as they rely on the implementation of the translational semantics [14], [15] or are specific to a couple of languages (DSML-formal language) [1]. In previous work [23], we used traceability data and an ad-hoc model transformation to feedback the verification results to the DSML level. However, this was too complex and low level task for the usual DSML designer. These ones should be able to rely on a method and associated tools mostly independent from the DSMLs, formal languages and implementation technologies.

Our contribution provides a domain-specific language (DSL), named FEVEREL (Feedback Verification Results Language) to model the feedback of the formal level verification results to the DSML one. It relies on the Executable DSML pattern [6] which reifies the concerns involved in the definition of DSML semantics. Verification results are traces that combines executed events and intermediate model dynamic states that must both be translated from the formal world to the DSML one.

This paper is organized as follows. Section II gives an overview of our method to integrate the formal verification activity for a DSML. Section III details our proposal to leverage verification results and the associated DSL and tools. Section V gives some related work in the domain of user level verification results. Finally, we conclude and present future works in Section VI.

II. OVERVIEW OF THE APPROACH

Feedback of verification results is done in the context of the definition of a formal verifier for a DSML. In this section, we summarize our method for integrating model-checking for a new DSML (Figure 1) [23]. It relies on a translational semantics from the DSML to a formal language that provides appropriate model checking tools. The main objective is to provide the DSML designers with tools that ease this integration.

The starting point is the abstract syntax (or metamodel) of the DSML and its concrete syntaxes. The main requirements to perform formal verification activities are to:

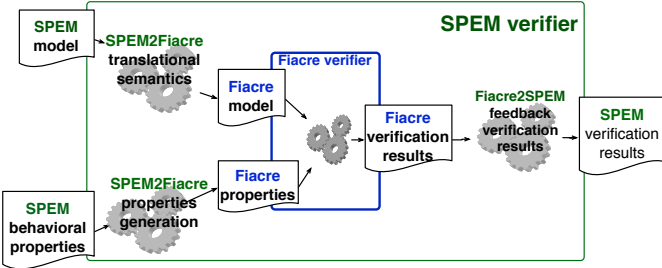


Figure 1. Architecture of a SPEM verifier using FIACRE as a formal language

- 1) empower the DSML end-users with a language to express the behavioral properties they want to assess on their models. At the language level, we rely on the temporal extension of OCL called TOCL [24].
- 2) automatically translate the models and properties to the formal verification domain. The translational semantics must be provided by the DSML designer as well as insights on the properties translation.
- 3) feedback the verification results from the formal domain to the DSML domain. This aspect, the focus of this paper, will be presented in the next section.

In order to fulfill these requirements, we explicitly state concerns that are usually not present in the DSML metamodel. These ones include runtime data and the ability to represent execution scenarios. If formal verification fails, a counter example is usually provided that must be presented to the DSML designer as a scenario composed of the steps that lead to the unwanted state in the model.

A. The SPEM case study

To illustrate the application of this method, we take as running example the xSPEM [19]: an executable extension of the SPEM process modeling language. As shown at the bottom of Figure 3, a SPEM *Process* is composed of activities (*WorkDefinition*) performed during the process, resources (*Resource*) required to run activities (*Parameter*) and temporal dependencies (causality constraints) between activities (*WorkSequence*). The *linkType* attribute from *WorkSequence* specifies the kind of dependency between the source and target activities. It follows the *stateToAction* pattern: the source activity must have reached the state (started or finished) to allow the action (start or finish) on the target activity: *startToFinish* means that the target activity can only finish when the source activity has been started. Figure 2 gives an example composed of four activities drawn with ellipses: *Programming*, *Designing*, *Test case writing* and *Documenting*.

The “finishToStart” dependency between *Designing* and *Programming* means that this one can only be started when *Designing* has been finished. *Documenting* and *TestCaseWriting* can start once *Designing* is started (*startToStart*) and *Documenting* cannot finish if *Designing* is not finished (*finishToFinish*). The dependencies between *Programming* and *TestCaseWriting* enforce a test driven development: *Programming* can only start when *TestCaseWriting* has started and,

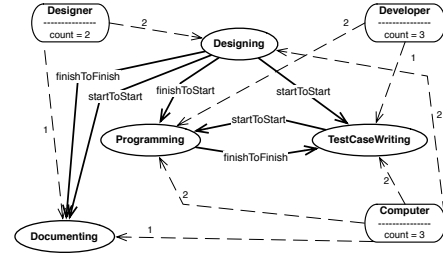


Figure 2. A SPEM development process

obviously, *TestCaseWriting* can only be finished when *Programming* is finished.

Rounded rectangles represent resources with their amounts (2 *Designers*, 3 *Developers* and 3 *Computers*). Dashed arrows indicate how many occurrences of a resource an activity requires. On Figure 2, *Programming* needs two developers and two computers. Resources are allocated when an activity starts and freed when it finishes.

B. Expliciting verification concerns for a DSML

Model executability is a key feature to introduce behavioral V&V for DSMLs. It specifies how the model evolves over time. Defining the DSML execution semantics requires extending its metamodel with concepts that capture the additional dynamic data that represent the execution. In this purpose, we use the *Executable DSML pattern* proposed in [6] that defines and structures the concerns required to make a DSML executable. Figure 3 applies this pattern on the SPEM example.

The original metamodel is called the DDMM (Domain Definition MetaModel), bottom of Figure 3. It is extended with three other metamodels. The first one specifies stimuli modeled as events that triggers the model evolutions. *Start a WorkDefinition*, *Finish a WorkDefinition* or *Fire a WorkSequence* are examples of xSPEM events. This first extension is called the EDMM (Event Definition MetaModel), top left of Figure 3. The second one models 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 3. TM3 is generic as it only relies on the abstract *Event* concept common to all DSML. These two extensions allow to build a DSML scenario as a sequence of DSML events. The third one specifies the runtime data that model the state of the model at runtime and that are not part of the DDMM. This extension is called SDMM (State Definition MetaModel), middle of Figure 3. On the xSPEM example, the SDMM includes the work definition progress state which can be *not started*, *running* or *finished*.

The *Executable DSML pattern* provides the different concerns for graphical model animation since SDMM and EDMM introduce the data that represents the execution semantics [8]. But it lacks a more abstract definition for a model state needed to write behavioral requirements. We proposed in [23] another metamodel extension to ease behavioral properties writing and favor a *Property-Driven Approach*

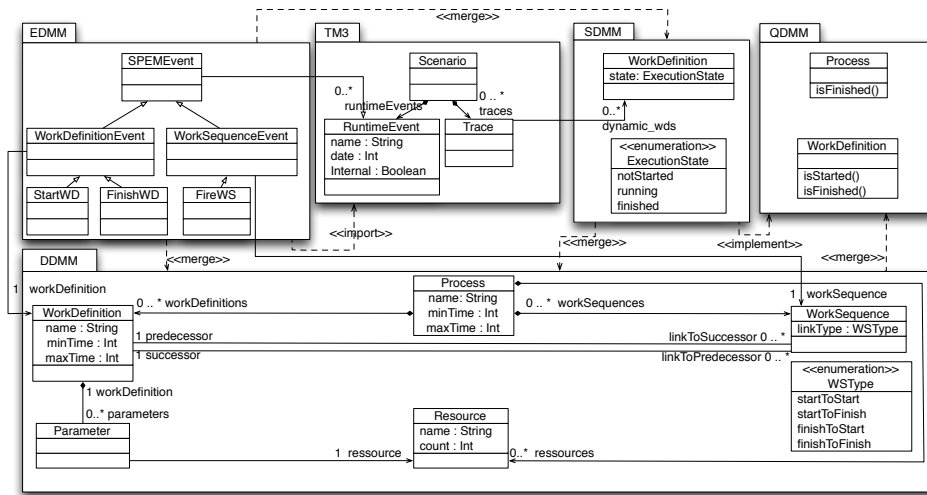


Figure 3. The Executable DSML pattern applied into the SPEM metamodel

as defined in [5] and experimented in [12]. This approach defines an abstract dynamic semantics as properties expressed at the metamodel level. This last extension allows to capture different queries that can be asked on DSML conforming models during their execution. It is called the *Query Definition MetaModel* (QDMM). On Figure 3, two queries are defined on *WorkDefinition*. They are used to check if an activity has started or finished. The QDMM is a kind of abstract view of the SDMM. SDMM may be seen as a way to implement the QDMM by choosing a set of attributes (like a Java class implements a Java interface). As an example, the queries *isFinished()* and *isStarted()* of *WorkDefinition* can be implemented thanks to the attribute *state* defined in SDMM: *isFinished()* is equivalent to $(state = \#finished)$ and *isStarted()* to $(state = \#running)$ or $(state = \#finished)$.

C. Translational semantics and related elements

In our example, the translational semantics defines a mapping from the SPEM DSML to FIACRE. FIACRE is a formal specification language to represent both the behavioral and timing aspects of real-time systems [2]. The FIACRE language is composed of two syntactical constructs, processes and components. A process describes the behavior of sequential components. It is defined by a set of control states, each associated with an expression that specifies state transitions. A component describes the composition of processes, possibly in a hierarchical manner. It is defined as a parallel composition of components and processes communicating through ports and shared variables. Here is some rationale behind the SPEM to FIACRE mapping illustrated with elements from the FIACRE model corresponding to the SPEM example from Figure 2.

Each work definition is translated to one FIACRE process with the same name. This process is composed of three states (*notStarted*, *running* and *finished*) and two transitions (from *notStarted* to *running* and from *running* to *finished*). Transition between the states are derived from the work sequences and thus depends on the state of the predecessor work definition.

Thus, it is necessary to store the current states of different work definitions.

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

WDsQueries defines an array of *WDQueries* storing the state of all work definitions of an xSPEM process. It is a parameter for every work definition process used to implement dependencies as a FIACRE process cannot inspect the current state of other processes.

```

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

type WDsQueries is array 4 of WDQueries end

```

Named constants are defined to ease the reading of the FIACRE model by avoiding the use of meaningless integers to encode a work definition.

```

const DesigningWD : int is 0
const ProgrammingWD : int is 1
const DocumentingWD : int is 2
const TestCaseWritingWD : int is 3

```

The *WDsQueries* variable is updated when a transition from a work definition process is fired. For example, on the transition from the *notStarted* state to the *running* state, the *isStarted* variable is set to *true*.

Furthermore, one work definition can only be started when the required resources are available. As for work definitions, we have modeled resources queries as an array. Array elements are integers as there is only one query on *Resource* meta-class.

ResourceTab defines an array of integer storing the available count of each resource.

```

type ResourceTab is array 3 of int

```

As for work definitions, named constants are defined to ease resource identification.

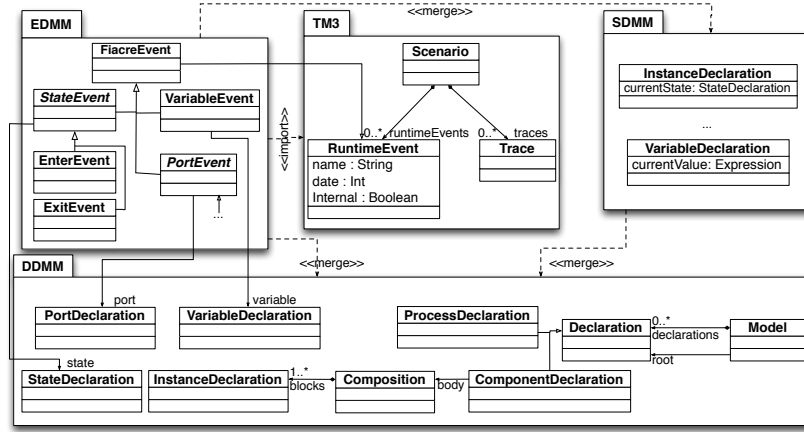


Figure 4. The Executable DSML pattern applied on the FIACRE language

```
const DesignerR : int is 0
const DeveloperR : int is 1
const ComputerR : int is 2
```

xSPEM causality constraints are mapped as a FIACRE conditional statement that checks whether the FIACRE processes corresponding to the previous work definitions have reached the expected states.

For example, the *startToStart* constraint between *Designing* and *Documenting*, leads to a conditional statement that checks whether work definition *Designing* is started. It also verifies whether the required amount for each required resource is available to run this work definition. If the condition evaluates to false, nothing happens else the current state becomes *running*, the state of this work definition is updated, and the available resources are decreased. The following process is the translation of the *Programming* work definition in FIACRE.

```
process Programming(&WorkDefinition: ProcessWDQueries,
                  &Ressource: RessourceTab) is
states notStarted, running, finished
from notStarted
if ( WorkDefinition[$(DesigningWD)].isFinished and
    WorkDefinition[$(TestCaseWritingWD)].isStarted and
    Ressource[$(DeveloperR)]>=2 and
    Ressource[$(ComputerR)]>=2 )
then
    Ressource[$(DeveloperR)] := Ressource[$(DeveloperR)] - 2;
    Ressource[$(ComputerR)] := Ressource[$(ComputerR)] - 2;
    WorkDefinition[$(ProgrammingWD)].isStarted:= true;
to running
else
loop
end if
from running
WorkDefinition[$(ProgrammingWD)].isFinished:= true;
Ressource[$(DeveloperR)] := Ressource[$(DeveloperR)] + 2;
Ressource[$(ComputerR)] := Ressource[$(ComputerR)] + 2;
to finished
```

The FIACRE *Main* component instantiates one FIACRE process for each work definition in the xSPEM process (here four processes for *Designing*, *Programming*, *Documenting* and *TestCaseWriting*) with the arrays that store work definitions' states (initially all work definitions are not started and not finished) and resources available amounts.

```
component Main is
var
WorkDefinition: ProcessWDQueries :=
[ { isStarted = false, isFinished = false },
  { isStarted = false, isFinished = false },
  { isStarted = false, isFinished = false },
  { isStarted = false, isFinished = false } ],
```

```
Ressource : RessourceTab := [2,3,4]
par
Designing (&WorkDefinition,&Ressource)
||
Programming (&WorkDefinition,&Ressource)
||
Documenting (&WorkDefinition,&Ressource)
||
TestCaseWriting (&WorkDefinition,&Ressource)
end
```

This translational semantics is defined as a model-to-model (M2M) transformation expressed in ATL [16]. Once it is defined, the DSML designer implements the primitive queries based on the formal elements created by the translational semantics. These queries guide the automated generation of behavioral properties in the formal domain with a higher order transformation. The following list first gives two properties expressed in TOCL and the result of the translation of the first one in FIACRE.

```
1 context SPEM!Process inv willNeverFinish: always not self.isFinished()
2 context SPEM!Process inv willEventuallyFinish: eventually self.isFinished()
```

```
property willNeverFinish is ltl
([ ( not (Main/1/value WorkDefinition[$(DesigningWD)].isFinished
and Main/1/value WorkDefinition[$(ProgrammingWD)].isFinished
and Main/1/value WorkDefinition[$(DocumentingWD)].isFinished
and Main/1/value WorkDefinition[$(TestCaseWritingWD)].isFinished )))
```

D. Formal verification process

Formal verification can be performed once the full FIACRE model has been generated by the translational semantics including the properties generation. The model-checker of the TINA toolbox[3] (SELT) performs the formal verification relying on a translation to Time Petri Nets and produces verification results in this low-level formal language. In [22], we have proceeded to feedback verification results from the formal low-level, to the formal intermediate language, FIACRE, using the traceability data generated during the translation to Time Petri Nets and the use of the *Executable DSML pattern* on the FIACRE metamodel.

As shown in Figure 4, in FIACRE, we can capture events [9] related to a state (*StateEvent*): an instance of a process entering (*EnterEvent*) or leaving a state (*ExitEvent*), a variable changing value (*VariableEvent*), a communication through a port (*PortEvent*). The FIACRE SDMM captures the current

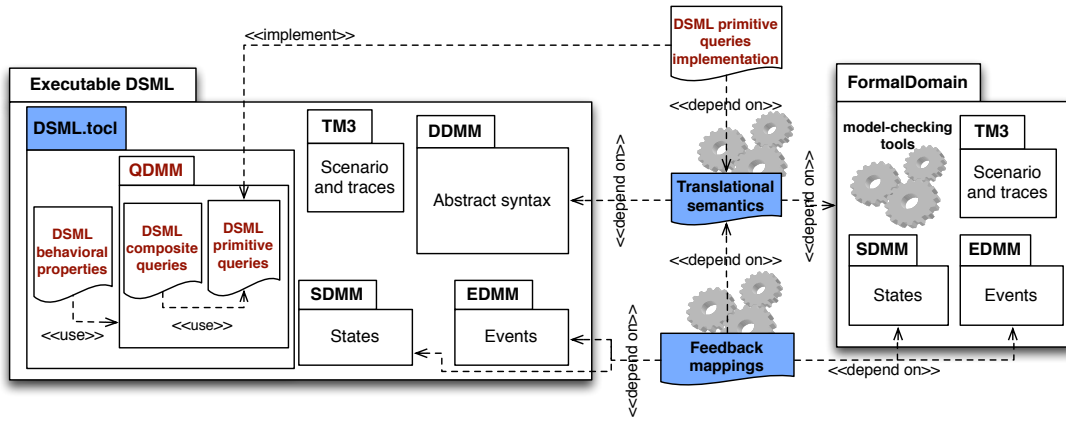


Figure 5. Towards the generation of a DSML verification framework

value of a FIACRE variable (*VariableDeclaration*) and the current state of a FIACRE instance (*InstanceDeclaration*).

The verification output is a trace given as a sequence of FIACRE events (instances of EDMM elements) and states (instances of SDMM elements) during its execution. The following listing gives the first steps of such traces. It shows how a FIACRE model evolves during time. At first, the initial state (lines 1-6) corresponds to the current state of the *WorkDefinition* variable. Then, an *EnterEvent* in the *running* state of the model first instance is triggered (line 8). It triggers the model evolution and, consequently, the corresponding *isStarted* field is updated (line 12) in the next state (lines 10-15).

```

1 state 0:
2   WorkDefinition =
3   [{ isStarted = false , isFinished = false },
4     { isStarted = false , isFinished = false },
5     { isStarted = false , isFinished = false },
6     { isStarted = false , isFinished = false }]
7
8 event: EnterEvent {path: Main/1, state : running}
9
10 state 1:
11   WorkDefinition =
12   [{ isStarted = true , isFinished = false },
13     { isStarted = false , isFinished = false },
14     { isStarted = false , isFinished = false },
15     { isStarted = false , isFinished = false }]
16 ...

```

This step must provide the verification results for the DSML end-user in an appropriate format that corresponds to his knowledge.

III. FEEDBACK VERIFICATION RESULTS LANGUAGE

This section introduces our proposal to ease for the DSML designer the implementation of the verification result feedback from the formal level to the DSML level. It relies on a DSL: the Feedback Verification Results Language (FEVEREL) and associated tools. We first motivate the use of a DSL and then illustrate its use on the xSPeM example. Finally, we discuss adopted patterns to implement this DSL and detail parts of its implementation.

A. Motivations

A key task for ensuring efficient V&V experience for DSMLs end-users is to provide easy to understand verification results extracted from low level formal model checking tools. The DSML designer must provide the service that generates a model execution trace at the DSML level.

Currently, the DSML designer can implement a M2M transformation which imports the formal verification results and produces the DSML ones. But, this can be costly and difficult to maintain as the mapping between runtime concerns (events and states) of both domains can be complex. We propose to provide the DSML designer with a DSL to model the feedback of verification results, and tools to automatically generate these transformations.

Figure 5 shows our vision to transparently empower a DSML with formal verification activity. We target a separation of concerns for the DSML designer (the translational semantics, the properties generation and the feedback of verification results). The implementation of each element in the DSML integrated verification toolchain is assisted by a specific tool. The initial step consists in defining concerns for the executable DSML (left side). It includes the abstract syntax (DDMM), the concerns from the *Executable DSML pattern* (TM3 and SDMM), and the queries (QDMM) used to express behavioral properties. The *Executable DSML pattern* is also applied on the formal domain (right side). The DSML designer is then able to define both the translational semantics and the primitive queries (the green box) for the domain model and properties to be automatically translated to the formal level. The last point is to get back verification results generated in the formal domain (the second blue box), the DSML designer has to define mappings between DSML runtime concerns modeled at the DSML level (EDMM and SDMM) and the corresponding elements in the formal domain.

B. FEVEREL applied on the xSPeM case-study

FEVEREL is a DSL for a DSML designer to ease the implementation of verification results feedback tools. It allows to define how the DSML runtime information (events and

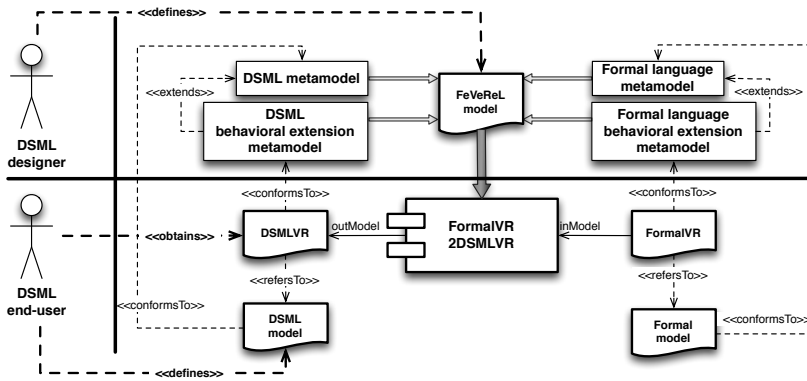


Figure 6. Architecture of FEVEREL

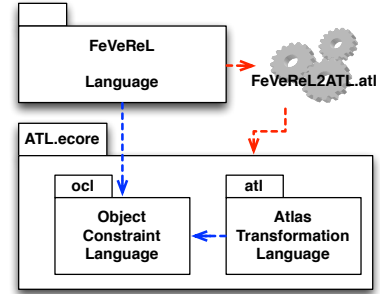


Figure 7. Implementation of FEVEREL DSL

states) can be observed at the formal level. The architecture of FEVEREL is shown in Figure 6. The entry-point is a FEVEREL model defined by the DSML designer. A FEVEREL Xtext editor serves as an interface to ease the DSML designer task.

Based on the DSML metamodel, the formal language metamodel and their behavioral extension metamodels, the DSML designer defines the mappings between DSML runtime information defined in the EDMM and the SDMM of the DSML and their corresponding elements in the formal domain (FEVEREL model in Figure 6).

We illustrate the syntax and semantics of FEVEREL with the xSPeM case-study. We rely on the application of the *Executable DSML pattern* on xSPeM (Figure 3) and on FIACRE metamodels (Figure 4) and on the translational semantics proposed in the subsection II-C.

Listing 1 shows a FEVEREL model defined for the xSPeM running example. It first *imports* the concerned DSML metamodels (lines 1-2). *DSMLMM* represents this metamodel. *DSMLBEMM* extends the first metamodel with the application of the *Executable DSML pattern*. Then, we import formal language metamodels. *FormalMM* is the abstract syntax of the formal language metamodel, and *FormalBEMM* is the behavioral extension applied on the *FormalMM* (lines 3-4).

The FEVEREL model contains mappings (*Mapping*). Two kinds of mappings can be defined: events mappings (*EMapping*) and states mappings (*SMapping*). A *Mapping*, characterized by an identifier, describes a relation between observable elements in the DSML domain (*DSMLStream*) and in the formal domain (*FormalStream*). A *DSMLStream* contains a sequence of elements (*DSMLStreamElement*) which allow to structure an observation in the DSML. Currently, a *DSMLStreamElement* is only one runtime observation (*DSMLRO*). Extensions to more sophisticated runtime observations are part of future work. A *DSMLRO* is characterized by an identifier. It refers to a meta-class in the DSML metamodel or its behavioral extension. In addition, it defines a set of bindings (*Binding*) which specify the initialization of a feature (an attribute or a reference) of a DSML runtime information using an expression (*body*). In the formal domain, as in the DSML one, a *FormalStream* contains only one runtime

observation (*FormalRO*). A *FormalRO* has an identifier, refers to a meta-class in the formal language metamodel or its behavioral extension and contains an OCL expression (*body*). The following listing contains three events mappings: *swd2ee* (lines 6-21), *fwd2ee* (lines 23-38) and *fws2ee* (lines 40-56).

fwd2ee (lines 23-38) is an events mapping (an instance of *EMapping*) that captured the *FinishWD*. It can be observed when the corresponding FIACRE instance enters (*EnterEvent* in line 28) in the *finished* state (line 29). The concerned instance is identified according to its index in the FIACRE component that must corresponds to the index of the work definition in the SPeM process. *FinishWD* features are initialized according to the information observed in the corresponding *EnterEvent* (line 25).

In the following listing, three states mappings (instances of *SMapping*) are defined: *wnotStarted2vd* (lines 58-69), *wdrunning2vd* (lines 71-85) and *wdfinished2vd* (lines 87-98). This last one captures whether a work definition is finished or not. The DSML designer may refer to the FIACRE array defined by the translational semantics. He must identify the shared FIACRE variable that stores the work definitions states (line 93) and verify whether the corresponding *isFinished* field in the array is *True* (lines 95-96).

```

1 import "http://spemDDMM/1.0" as DSMLMM
2 import "http://spemSemantics/1.0" as DSMLBEMM
3 import "http://www.topcased.org/fiacre/xtext/Fiacre" as FormalMM
4 import "http://fiacreSemantics/1.0" as FormalBEMM
5
6 events mapping swd2ee:
7
8   DSMLEvent swd: DSMLBEMM.StartWD (
9     date <- ee.date
10  )
11  observed as
12  FormalEvent ee: FormalBEMM.EnterEvent (
13    ee.state.name = 'running'
14    and
15    FormalMM!Modedatel.allInstances()->first().root.body.blocks
16    ->indexOf(ee.path.instances->first())
17    =
18    DSMLMM!Process.allInstances()->first().processElements
19    ->select(e|e.ocIsTypeOf(DSMLMM!WorkDefinition))
20    ->indexOf(swd.workdefinition)
21  )
22 end events mapping
23

```

```

24 events mapping fwd2ee:
25
26 DSMLEvent fwd: DSMLBEMM.FinishWD (
27   date <- ee.date
28 )
29 observed as
30 FormalEvent ee: FormalBEMM.EnterEvent (
31   ee.state.name = 'finished'
32   and
33   FormalMM!Model.allInstances()->first().root.body.blocks
34     ->indexOf(ee.path.instances->first())
35   =
36   DSMLMM!Process.allInstances()->first().processElements
37     ->select(e|e.ocIsTypeOf(DSMLMM!WorkDefinition))
38     ->indexOf(fwd.workdefinition)
39 )
40 end events mapping
41
42 events mapping fws2ee:
43
44 DSMLEvent fws:DSMLBEMM.FireWS (
45   date <- ee.date
46 )
47 observed as
48 FormalEvent ee: FormalBEMM.EnterEvent (
49   self.state.name =
50   if (fws.worksequence.linkType=# startToStart
51     or fws.worksequence.linkType =# startToFinish )
52   then
53     'Running'
54   else
55     'finished'
56   endif
57   and
58   self.instance.instance.component.name
59   =
60   fws.worksequence.predecessor.name
61 )
62 end events mapping
63
64 states mapping wdnnotStarted2vd:
65
66 DSMLState wd:DSMLMM.WorkDefinition (
67   state <- #notStarted
68 )
69 observed as
70 FormalState vd: FormalMM.VariableDeclaration (
71   vd.name= 'WorkDefinition'
72   and
73   vd.value.values->at(wd.getIndex()).fields
74     ->at(0).value.ocIsTypeOf(FormalMM!FalseLiteral)
75 )
76 end states mapping
77
78 states mapping wdrunning2vd:
79
80 DSMLState wd:DSMLMM.WorkDefinition (
81   state <- #running
82 )
83 observed as
84 FormalState vd: FormalMM.VariableDeclaration (
85   vd.name= 'WorkDefinition'
86   and
87   vd.value.values->at(wd.getIndex()).fields
88     ->at(0).value.ocIsTypeOf(FormalMM!TrueLiteral)
89   and
90   vd.value.values->at(wd.getIndex()).fields
91     ->at(1).currentValue.ocIsTypeOf(FormalMM!FalseLiteral)
92 )
93 end states mapping
94
95 states mapping wdfinished2vd:
96
97 DSMLState wd:DSMLMM.WorkDefinition (

```

```

98   state <- #finished
99 )
100 observed as
101 FormalState vd: FormalMM.VariableDeclaration (
102   vd.name= 'WorkDefinition'
103   and
104   vd.value.values->at(wd.getIndex()).fields
105     ->at(1).currentValue.ocIsTypeOf(FormalMM!TrueLiteral)
106 )
107 end states mapping

```

Listing 1. SPEM/FIACRE FEVEREL mappings

The use of FEVEREL bestows the DSML designer with a structured specification of how verification results should be brought back to the DSML level. Based on the translational semantics and both extensions, he models a mapping using OCL to identify which formal events corresponds to DSML ones. Then, the M2M transformation (*FormalVR2DSMLVR* from Figure 6) is automatically generated. This transformation translates verification results (*FormalVR* in Figure 6) generated by model checking tools into DSML verification results easier to understand by the DSML end-user.

For the xSPEM example, once the formal verification process is performed, the toolchain generates verification results at the xSPEM level as shown below.

```

1 state: Designing notStarted Documenting notStarted
2   Programming notStarted TestCaseWriting notStarted
3 event: StartWD Designing
4 state: Designing running Documenting notStarted
5   Programming notStarted TestCaseWriting notStarted
6 event: FinishWD Designing
7 state: Designing finished Documenting notStarted
8   Programming notStarted TestCaseWriting notStarted
9 event: StartWD Documenting
10 state: Designing finished Documenting running
11   Programming notStarted TestCaseWriting notStarted
12 event: FinishWD Documenting
13 state: Designing finished Documenting finished
14   Programming notStarted TestCaseWriting notStarted
15 event: StartWD TestCaseWriting
16 state: Designing finished Documenting finished
17   Programming notStarted TestCaseWriting running

```

This process shows that the second behavioral property *willEventuallyFinish* does not hold as a deadlock prevents the xSPEM process to finish. After analyzing these results, the xSPEM end-user should understand that the xSPEM process cannot finish because a *Computer* is missing.

With an additional *Computer*, the first property fails and the second property holds. For the first one, the xSPEM verification toolchain produces a terminating scenario where the process and all the activities finishes.

Thanks to this architecture, the DSML designer obtains a suitable tool to define mapping between runtime information. He does not have to deal with technical aspects of the model transformation. The DSML designer focuses on the modeling of how a DSML behavioral element can be observed in the formal domain relying on the behavioral extensions of both domains and the defined translational semantics which maps the abstract syntax of the DSML into the formal ones.

C. Implementation of FEVEREL

A DSL design should be suitable for its end user, that is corresponds to his knowledge and abilities. Designing a new DSL

is one of the main challenges of modern software engineering as it is an error-prone and time consuming task [17]. Thus, it is mandatory to adopt DSL design strategies to ease that process. [20] introduces eight DSL design patterns in that purpose. The design of FEVEREL relies on two patterns: **piggyback** and **source to source transformation**. The first one uses an existing language as a host for the new DSL. This one can be a general-purpose language which offers standardization and expressiveness and makes it more user-friendly. The DSL then share common syntactical elements such as expression handling, operations, arithmetic and logic operators, etc. The second one allows an efficient implementation of the DSL by leveraging the facilities provided by existing tools for other languages. The DSL source code is translated to the source code of existing languages and then existing tools are used to host the generated code. For the DSML designer, the new tool must be user-friendly, close in syntax and semantics to his skills and capabilities like metamodeling with ECORE, expressing constraints with OCL.

Figure 7 shows the implementation of the FEVEREL language. It implements the **piggyback** pattern with OCL as base language (the blue dashed arrow). The second part of the implementation of FEVEREL language concerns the translation part (the red dashed arrows). The **source to source transformation** pattern has been used to ease the burden of implementation. We have chosen the ATL transformation language as a host language (ATL.ecore) because we aim to automatically generate an ATL model transformation from a FEVEREL model. The pattern intersects with an useful MDE technique: higher-order transformations. The FEVEREL translation is thus implemented as a higher-order model transformation `FeVeReL2ATL` using ATL.

Each *Mapping* is translated in:

- 1) an ATL helper without parameters whose context is the super type of all kind of events in the formal metamodel extensions (EDMM and SDMM). Its return type corresponds to a set of elements in the DDMM on which the *DSMLRO* is instantiated. Its body selects a subset of all instances of these elements from the DDMM where the body of *FormalRO* evaluates to true.
- 2) a lazy rule whose source pattern is the element in the DDMM on which the *DSMLRO* is instantiated and its different features (*features*). The target pattern creates an instance of the *DSMLRO* (an event or a state) with different features declared in the source pattern.

The FEVEREL *Model* is translated to ATL rules whose source pattern is a formal scenario (an instance of the *Scenario* meta-class in the TM3 of the formal language). Its target pattern is a DSML scenario (an instance of the *Scenario* meta-class in the TM3 of the DSML). It aims to produce DSML verification results from the formal ones. A formal scenario is iterated using an `iterate` expression. The iterated variable which is the current instance that the *FormalRO* will check, by calling the helper generated previously, if there are elements in the DDMM which satisfy the body of the helper. According

to this result, the lazy rule will be called for each element of the returned subset and with its corresponding features.

We follow the same method used for the translation of behavioral properties at the DSML level to the formal level in [23] relying on DSLs and higher order transformations. The key point is that we help the DSML designer in the generation of a "DSML verification framework" for each DSML. This one is structured by the use of the *Executable DSML pattern*. Combining the **piggyback** and the **source to source transformation** patterns shows two advantages. First, a big part of the transformation is the identity. Indeed, we use OCL as a base language to implement our DSML and ATL as a host language to apply source to source transformation pattern where ATL also relies on OCL. The translation is thus restricted to a limited number of elements. Second, often DSMLs evolve and can be extended. So, an eventual extension of FEVEREL will be easily adopted by a DSML designer because he only needs to update the semantics with new domain-specific elements by extending the higher-order transformation while the OCL part is unchanged.

IV. EVALUATION

This section assesses our proposed DSL from three perspectives: design process, expressiveness and effectiveness.

a) *Design process*: FEVEREL was defined to meet the DSML designer needs for a dedicated tool that focuses on the required concepts while hiding most technical details. The process starts from ad-hoc solutions for distinct examples and then abstracts the required concepts for the new language. We first wrote manually the target ATL transformations and analyzed them to extract the needed concepts to allow their automatic generation. This analysis concluded that the DSML behavioral extensions could be used as pivot to handle the feedback process but that other elements were also needed to build the feedback. Thus, we defined a specific DSL to express the mapping between both behavioral extensions. Consequently, the DSML designer only needs to handle his DSML concepts without dealing with the implantation details.

b) *Expressiveness*: FEVEREL requires sophisticated model querying facilities to select and extract complex data from the formal analysis results. OCL provides this expressiveness commonly used in that purpose in model transformation languages. However, it will only create DSML verification results, that is sequences of events and model states. A full model transformation language would be too expressive on the creation side. Thus, it was defined as a DSL to limit this aspect. At last, the relations between source and target in future version of FEVEREL need to express how the selection of several elements in the source, observed in such chronological order, will allow to create a structured set of elements in the target (m to n mappings), which is not possible easily with all transformation languages. To adapt the FEVEREL language to this approach, it should be necessary to choose the appropriate pattern to capture events. Complex Event Processing (CEP) [4] and the patterns specification

language of Dwyer et al. [10] should be interesting candidates to do that.

The following listing shows a subset of the generated ATL model transformation. The first element corresponds to the entry point rule (lines 1-29) and the two following elements, the *helper* and the *lazy rule*, correspond to the *fwd2ee* mapping (lines 23-38) shown in Listing 1. The helper (lines 31-40) consists in identifying, based on an observed formal runtime information, the DSML elements that verify OCL expression defined in the *body* of the mapping. Then, after calling this helper to verify whether a formal runtime information respects this expression (line 16), the entry-point ATL rule calls the corresponding *lazy rule* for each found element (line 20) to create a DSML runtime information (lines 42-50). So, the FEVEREL interface allows to hide all model transformation noises and gives for the DSML designer a more suitable tool to manage the feedback of verification results.

```

1 helper context FormalBEMM!Event def: getworkdefinitionFinishWD():
2   Sequence(DSMLMM!WorkDefinition) =
3     DSMLMM!WorkDefinition.allInstances()
4     ->select(fwd |self. state .name = 'finished'
5       and
6       FormalMM!Model.allInstances()
7       ->first(). root .body.blocks
8       ->indexOf(self.path.instances ->first())
9     =
10    DSMLMM!Process.allInstances()
11    ->first(). processElements
12    ->select(e |e.ocls!TypeOf(DSMLMM!WorkDefinition))
13    ->indexOf(fwd)
14 );
15
16 lazy rule createFinishWDEvent {
17   from fwd : DSMLMM!WorkDefinition, ee: FormalBEMM!Event
18   to DSML_event : spemSemanticsMetaModel!FinishWD (
19     workdefinition <- fwd,
20     date <- ee.date
21   )
22 }

```

c) *Effectiveness*: Ensuring the correctness of model transformations is a key in the definition of DSML verification toolchain. In [23], we proposed to specify invariants on the DSML models using TOCL and then transform them into LTL properties that must hold on the generated formal model as translation validation. For FEVEREL, a possible alternative consists in defining OCL invariants that must hold on the generated DSML verification results. If they fail, an error is detected in the definition of FEVEREL mappings.

V. RELATED WORKS

The problem of integrating formal verification into the design of DSMLs has been widely addressed by the MDE community. However, the analysis feedback at the DSML level problem is typically either ignored or resolved by defining ad-hoc or hard-coded solutions. For example, [1] proposes the *Metaviz* approach based on the real-time systems specification and validation toolset IFx-OMEGA. It is designed to ease the visualization of the simulation trace by assisting the user in the Interactive Simulation task. It refines this step with a diagnosis process built around visualization concepts. It relies on feedback from the verification results at the OMEGA level. Thus, it can be considered as an ad-hoc approach. The use of

the *Executable DSML pattern* on both domains and FEVEREL could ease the implementation of their work.

Works handling the feedback with general solutions also exists in the literature. [7] introduces an algorithm that requires a formal definition of the DSML's semantics and of a relation between states of the DSML and states of the target language. The DSML designer must also 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. We feel that the DSML designer who hardly knows formal methods will fail in providing these mandatory data required to feedback verification results.

Hegedüs et al. [15] propose a technique for the back propagation of the simulation traces based on change-driven model transformations from traces generated by the SAL model checker to a specific BPEL Animation Controller. The change-driven model transformation consumes changes in the Petri nets simulation run and produces a BPEL process execution using traceability information generated while running the translational semantics. In this case, after defining the runtime extension for both levels (BPEL and Petri nets) and the translational semantics, the DSML designer is invited to define 1) a change command metamodel for Petri nets and BPEL and 2) the backward change-driven transformation. In our approach, we try to provide the DSML designer with a high-level tool to define mapping between events. A variant of FEVEREL could ease their implementation.

Gerkin et al. [13] rely on a similar approach to feedback verification results from the UPPAAL model checker. It also uses an ad-hoc model to model transformation that could benefit from our proposal to focus on the DSML concepts instead of the low level technical details.

In [14], a domain-specific visual language called *BaVeL* is designed. It allows defining how a verification result should be reflected in terms of the original notation. It is based on triple graph patterns. This approach requires an additional information which is the mappings (named also traces) relating the source and target models and created during the execution of the translational semantics. This framework could be implemented using the QVT model transformation language as it creates traces between the source and target models. Usually, DSML designers choose to encode the translational semantics as a code generation process (model-to-text transformation) instead of a model-to-model transformation. So, this information is missing. In our approach this information is optional but not an essential one. The DSML designer decides if it is required to generate model transformation traces to ease the feedback with FEVEREL. He must then import the mappings metamodel in his FEVEREL specification. A variant of FEVEREL could ease their implementation.

Meyers et al. [18] have designed the ProMoBox framework in the same purpose as our work: ease the integration of formal verification for DSML. Their feedback transformation relies on simple event name mappings produced during the DSML to formal model translation. FEVEREL uses OCL to express sophisticated data matching on the states and events attributes

which enable the management of more sophisticated DSML to formal model transformations.

VI. CONCLUSION

This paper defined the FEVEREL DSL to ease the development of formal methods based verification tools that feedback their results at the DSML level. FEVEREL is thus a key element to ease the integration of verification tools on a DSML in order to assist system designers in the early model validation and verification activities.

FEVEREL relies on the *Executable DSML pattern* that favors the definition of generative tools and eases the integration of tools for new DSMLs. It provides the DSML designer with a better modeling interface (that avoids implementing complex model transformations) to describe how different DSML runtime information can be observed in the formal verification domain to be leveraged at the DSML level.

FEVEREL was used in several case-studies to feedback verification results at the DSML level: for example, in the transformation chain from [11] that verifies PLC (Programmable Logic Controller) models written in Ladder Diagram (LD) using the FIACRE intermediate language.

Future works will focus on supporting other kinds of mappings and improving the visualisation of the verification results. Currently, FEVEREL support only 1-to-1 mappings between the domains (a DSML event corresponds to a formal event and a DSML state corresponds to a formal state). We plan to integrate 1-to-n and m-to-n mappings that occur when the translational semantics is more complex. A possible solution consists in adopting the Complex Event Processing (CEP) technologies [4] that produce data streams by matching data streams. We also plan to connect the verification results from the DSML domain to animators like the one developed as part of the GEMOC¹ project.

REFERENCES

- [1] ABOUSSOROR, E. A., OBER, I., AND OBER, I. *Seeing Errors: Model Driven Simulation Trace Visualization*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 480–496.
- [2] BERTHOMIEU, B., BODEVEIX, J.-P., FILALI, M., FARAIL, P., GAUFILLET, P., GARAVEL, H., AND LANG, F. FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment. In *4th European Congress ERTS Embedded Real-Time Software (2008)* (Jan. 2008).
- [3] BERTHOMIEU, B., RIBET, P.-O., AND VERNADAT, F. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research* 42, 14 (2004), 2741–2756.
- [4] BUCHMANN, A. P., AND KOLDEHOFE, B. Complex event processing. *it - Information Technology* 51, 5 (2009), 241–242.
- [5] COMBEMALE, B., CRÉGUT, X., GAROCHE, P.-L., THIRIOUX, X., AND VERNADAT, F. A property-driven approach to formal verification of process models. In *9th International Conference Enterprise Information Systems (ICEIS) 2007, Funchal, Madeira, June 12-16, 2007, Revised Selected Papers* (2007), J. Filipe, J. Cordeiro, and J. Cardoso, Eds., vol. 12 of *Lecture Notes in Business Information Processing*, Springer, pp. 286–300.
- [6] COMBEMALE, B., CRÉGUT, X., AND PANTEL, M. A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01* (Washington, DC, USA, 2012), APSEC '12, IEEE Computer Society, pp. 282–287.
- [7] COMBEMALE, B., GONNORD, L., AND RUSU, V. A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics. In *Proceedings of the 7th European Conference on Modelling Foundations and Applications* (Berlin, Heidelberg, 2011), ECMFA'11, Springer-Verlag, pp. 35–51.
- [8] CRÉGUT, X., COMBEMALE, B., PANTEL, M., FAUDOUX, R., AND PAVEL, J. Generative Technologies for Model Animation in the TOPCASED Platform. In *6th European Conference on Modelling Foundations and Applications (ECMFA)* (Paris, France, June 2010), vol. 6138 of *LNCS*, Springer.
- [9] DAL ZILIO, S., AND ABID, N. Real-time Extensions for the FIACRE modeling language. In *MoVep 2010, Summer School on Modelling and Verifying Parallel Processes* (Aachen, Allemagne, June 2010). 6 pages FNRAE Quartef.
- [10] DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering* (1999).
- [11] FARINES, J.-M., DE QUEIROZ, M. H., DE ROCHA, V., CARPES, A. M., VERNADAT, F., AND CRÉGUT, X. A Model-Driven Engineering Approach to Formal Verification of PLC programs (regular paper). In *Emerging Technologies and Factory Automation (ETFA), Toulouse, France* (septembre 2011), IEEE, pp. 1–8.
- [12] GE, N. *Property Driven Verification Framework: Application to Real-Time Property for UML-MARTE Software Designs*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, mai 2014.
- [13] GERKING, C., SCHÄFER, W., DZIWOK, S., AND HEINZEMANN, C. Domain-specific model checking for cyber-physical systems. In *Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, MoDeVVA@MoDELS 2015, Ottawa, Canada, September 29, 2015*. (2015), M. Famelis, D. Ratiu, M. Seidl, and G. M. K. Selim, Eds., vol. 1514 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 18–27.
- [14] GUERRA, E., DE LARA, J., MALIZIA, A., AND DÍAZ, P. Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information & Software Technology* 51, 4 (2009), 769–784.
- [15] HEGEDÜS, Á., BERGMANN, G., RÁTH, I., AND VARRÓ, D. Back-annotation of Simulation Traces with Change-Driven Model Transformations. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods* (Washington, DC, USA, 2010), SEFM '10, IEEE Computer Society, pp. 145–155.
- [16] JOUAULT, F., AND KURTEV, I. *Transforming Models with ATL*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 128–138.
- [17] KARSAI, G., KRAHN, H., PINKERNELL, C., RUMPE, B., SCHNEIDER, M., AND VÖLKEL, S. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)* (2009), M. Rossi, J. Sprinkle, J. Gray, and J.-P. Tolvanen, Eds., pp. 7–13.
- [18] MEYERS, B., DESHAYES, R., LUCIO, L., SYRIANI, E., VANGHELuwe, H., AND WIMMER, M. Promobox: A framework for generating domain-specific property languages. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings* (2014), B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, Eds., vol. 8706 of *Lecture Notes in Computer Science*, Springer, pp. 1–20.
- [19] OMG. *Software Process Engineering Metamodel (SPEM) 2.0*, Mar. 2007.
- [20] SPINELLIS, D. Notable design patterns for domain specific languages. *Journal of Systems and Software* 56, 1 (Feb. 2001), 91–99.
- [21] ZALILA, F. *Methods and Tools for the Integration of Formal Verification in Domain-Specific Languages*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, December 2014.
- [22] ZALILA, F., CRÉGUT, X., AND PANTEL, M. Verification results feedback for FIACRE intermediate language. In *Conférence en Ingénierie du Logiciel (CIEL)* (June 2012).
- [23] ZALILA, F., CRÉGUT, X., AND PANTEL, M. Formal verification integration approach for dsml. In *Model-Driven Engineering Languages and Systems*, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, Eds., vol. 8107 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 336–351.
- [24] ZIEMANN, P., AND GOGOLLA, M. An Extension of OCL with Temporal Logic. In *Critical Systems Development with UML – Proceedings of the UML'02 workshop* (Sept. 2002), vol. TUM-10208, pp. 53–62.

¹<http://gemoc.org/>