Dependability of Model-Driven Executable DSLs
Critical Review and Solutions

Akram Idani(✉)

Univ. Grenoble Alpes, CNRS, LIG, 38000 Grenoble, France
Akram.Idani@univ-grenoble-alpes.fr

Abstract. One of the promising techniques to address the dependability of a system is to apply, at early design stages, domain specific languages (DSLs) with execution semantics. Indeed, an executable DSL would not only represent the expected system's structure but it is intended to itself behave as the system should run. However, in order to make executable DSLs a powerful asset in the development of safety-critical systems, not only a rigorous development process is required but the domain expert should also have confidence in the execution semantics provided by the DSL developer. The challenge addressed in this paper is then to verify whether execution semantics provided by Model-Driven Engineering (MDE) tools comply with the expected behaviour of a given DSL. We experimented existing MDE approaches with associated implementations (QVT, Kermeta, fUML), in order to debug a safety-critical system. This paper presents the lessons learned from this study and provides formal alternatives, based on the B method and CSP process algebra, which are well-established techniques allowing interactive animation on the one hand and reasoning on the behaviour correctness, on the other hand.

Keywords: B Method · Domain specific languages · MDE

1 Introduction

The Model Driven Engineering (MDE) paradigm suggests solutions to the two major problems of software development: (1) the software complexity, and (2) the gap between conceptual models and coding activities. Indeed, on the one hand, MDE advocates for the use of models throughout the engineering life-cycle in order to reduce complexity, and on the other hand, it is assisted by numerous tools (*e.g.* EMF[1], Xtext[2], ATL[3]) dedicated to a clear separation of concerns ranging from requirements to target platforms, and going through several design stages. Interoperability between these tools is favored by the use of standardized meta-modeling formalisms which increases automation especially for developing

[1] https://www.eclipse.org/modeling/emf/.
[2] https://www.eclipse.org/Xtext/.
[3] http://www.eclipse.org/atl/.

domain specific languages (called DSLs). In the last decade, several research works have been devoted in order to enhance DSLs by underlying operational semantics which makes them executable. One of the major advantages of executing a DSL is to provide abstractions of the system's behavior and hence allow the domain expert to perform early analysis of the expected system. Indeed, an executable DSL can be simulated and debugged by existing MDE-based tools (*e.g.* the Gemoc Studio[4]) leading to a better quality than a static DSL. Unfortunately, although these advantages show that executable DSLs are a promising paradigm, several issues related to correctness and the level of trust that one can have in execution engines are still challenges for a rigorous development process.

In this paper, we lead an experimental study built on the Petri-net DSL as it is developed by existing works [1,8,10,11] that applied MDE frameworks such as xMOF-fUML, QVT and Gemoc-Kermeta in order to address operational semantics and corresponding simulation/debugging activities. We tried their Petri-net DSLs to debug a safety-critical system and check their ability to address properties such as: correctness, deadlock-freedom, mutual exclusion and fairness. This paper presents a critical review and lessons learned from this study and provides formal alternatives, based on the B method and CSP[5] process algebra, which are well-established techniques allowing interactive animation on the one hand and reasoning on the behaviour correctness, on the other hand.

Section 2 describes the DSL on which we have built our experimental study and gives an overview about tools of our benchmark. Section 3 applies and compares algorithms as they are encoded in existing works for debugging a safety-critical model from the domain expert point of view. In Sect. 4 we provide a formal solution for the definition of execution semantics. Finally, Sect. 5 draws the conclusions and the perspectives of this work.

2 The Petri-Net DSL

In this paper, our case study is that of running Petri-nets. Petri-net is a visual language used for modeling concurrent systems. Its mathematical foundations inspired by the graph theory allow formal calculus about safety properties. The choice of this DSL is motivated by the fact that it was widely addressed by the research works interested in modeling and debugging techniques. This section presents structural and contextual constraints of this DSL as well as its execution semantics and defines a simple safety-critical Petri-net example.

2.1 Structural and Contextual Semantics

Figure 1 shows the Petri-net meta-model as considered by [1][6]. It is composed of three meta-classes: Net (the root class), Place and Transition. These classes are linked by four relationships: places, transitions, input and output.

[4] http://gemoc.org/.

[5] CSP: Communicating Sequential Processes.

[6] The ecore file can be found at: https://github.com/gemoc/petrinet/blob/master/petrinetv1/fr.inria.diverse.sample.petrinetv1.model/model/petrinetv1.ecore.

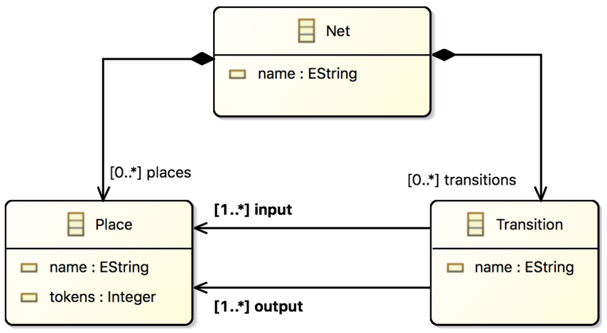


Fig. 1. Petri-nets meta-model

This meta-model defines structural properties of a given Petri-net. For instance, a Transition must be linked to at least one input place and one output place. Attribute tokens represents the number of tokens in a place: it is mono-valuated, optional and without a default initial value. The various references of this meta-model do not admit repetitions. Note that the meta-model is taken from [1] and it is presented without any modification. Furthermore, the DSL must comply with the following contextual invariant written in OCL:

```
context Place inv Token_Is_Natural: self.tokens ≥ 0
```

For illustration we use the simple Petri-net of Fig. 2 which is dedicated to control traffic lights in a crossroads. This model deals with a safety-critical system since failures may lead to loss of life due to accidents that it may cause.

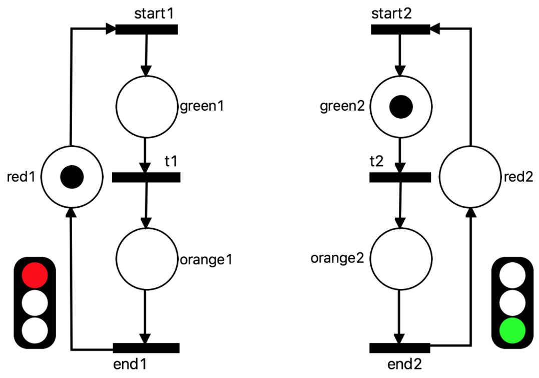


Fig. 2. Traffic light controller in Petri-nets (V1)

The domain expert needs then to have confidence in the provided operational semantics of the Petri-net DSL in order to prove that his model guarantees safety properties such as:

– correctness: asserts that the system does not exhibit bad behaviors, where invariants (structural or contextual) are violated.
– deadlock-freedom: states that the traffic lights can't be blocked in a state in which no progress is possible
– mutual exclusion: states that lights in a road intersection cannot enter simultaneously their critical sections (critical sections are states green and orange in our example).
– fairness: requires that the system gives fair turns to its components (in our example both lights must be able to function).

Model of Fig. 2 deals with two traffic lights (Light A and Light B) which are to be placed in two roads that intersect. Light A and Light B are respectively controlled by the left hand-side and the right hand-side of this figure. Every traffic light sequentially switches from Green to Orange and then to Red, in an infinite loop. This Petri-net model shows concurrent evolutions of traffic lights without any synchronisation between them. Finally, the current state of this model assigns red to Light A and green to Light B.

In this paper we apply existing MDE approaches [1,8,10,11], with associated implementations, in order to debug the traffic light controller especially from the domain expert point of view. The intention is to check the ability of these MDE tools to address safety properties as those mentioned above.

2.2 Execution Semantics

Basic Petri-nets execution semantics are defined by transition firing that holds when a transition satisfies an enabledness property. To check this property, existing MDE techniques call a query defined as:

```
query isEnabled(t : Transition) : Boolean =
    t.input->forAll(p : Place | p.tokens > 0)
```

This query returns true if attribute tokens is greater than 0 for each input place of transition t, false otherwise. Algorithm of Fig. 3, taken from [1], describes how a Petri-net runs. This algorithm chooses non-deterministically a transition t (called $t_{enabled}$) from the set of transitions that satisfy the above property and then calls operation fire(t). As a result, the number of tokens in the input places of t is decreased (operation removeToken) and the number of tokens in the output places is increased (operation addToken). Modifications of tokens, done at every call to operation fire, evolve the set of enabled transitions and then the algorithm may loop or stop when this set becomes empty.

2.3 Benchmark Overview

In order to address safety properties using existing MDE-based Petri-net DSLs, our study applies various approaches which are based on different languages (QVT, Kermeta and fUML). In the remainder, we call these approaches respectively PNet_{QVT}, $\text{PNet}_{\text{Kermeta}}$ and $\text{PNet}_{\text{fUML}}$.

```
Algorithm 1: run
    Input:
    n : the Net object to run
[1] begin
[2]     $t_{\text{enabled}} :\in \{t \in n.\texttt{transitions} \mid \textit{isEnabled}(t)\}$
[3]     while $t_{\text{enabled}} \neq \textit{null}$ do
[4]         $\textit{fire}(t_{\text{enabled}})$
[5]         $t_{\text{enabled}} :\in \{t \in n.\texttt{transitions} \mid \textit{isEnabled}(t)\}$
```

```
Algorithm 2: fire
    Input:
    t : the Transition object to fire
[1] begin
[2]     foreach $p \in t.\texttt{input}$ do
[3]         $\textit{removeToken}(p)$
[4]     foreach $p \in t.\texttt{output}$ do
[5]         $\textit{addToken}(p)$
```

Fig. 3. Running a Petri-net [1]

1. PNet$_{\text{QVT}}$ [11]: QVT (Query/View/Transformation) is an OMG[7] standard for model transformations. QVT defines: QVT-Relations and QVT-Core which are declarative languages but at two different levels of abstraction, and QVT-Operational which is an imperative language. In [11], the authors used QVT-Relations which is the high-level language of QVT extending OCL and its semantics with imperative features. Unfortunately, there is a lack of tools supporting QVT-Relations. Indeed, the tools that we found are either out of date (Medini QVT) or proprietary (ModelMorf). Then, for our benchmark needs, we encoded a variant of rules proposed by [11] in QVT-Operational using the Eclipse EMF framework.
2. PNet$_{\text{Kermeta}}$ [1]: Kermeta [7] is a language workbench that involves different meta-languages for abstract syntax (aligned with EMOF [4]), static semantics (aligned with OCL) and behavioural semantics (via an action language also called Kermeta). In [1], the Gemoc studio was applied together with the Kermeta language to define the Petri-net DSL and debug its execution using an animation technique. In our benchmark we used source-code issued from the Gemoc website: https://github.com/gemoc/petrinet/blob/master/petrinetv1/.
3. PNet$_{\text{fUML}}$ [8,10]: fUML is an OMG standard that defines the execution semantics of a subset of UML 2.3. The standard applies, in the form of pseudo Java-code, a basic virtual machine enabling UML models using elements comprised in the fUML subset to be executed. [10] proposes the xMOF tool which integrates fUML with MOF to enable the specification of the behavioural

[7] OMG: Object Management Group (https://www.omg.org).

semantics of DSMLs in terms of fUML activities. For our experiments we used the open-source DSL, provided at: https://modelexecution.org/moliz/xmof/.

The above tools use the Eclipse Modeling Framework (EMF), which makes easy their integration and the analysis of the Petri-net DSLs that they provide within a unified framework. Note that their underlying approaches agree on operations fire, addToken and removeToken. However, they differ from each other by: (1) the level of abstraction depending on (meta-)programming languages, (2) the semantics associated to the non-deterministic choice of enabled transitions, and (3) the execution engine.

3 Debugging the Traffic-Light Model

In this section we apply and compare the works of our benchmark for debugging the traffic-light model from the domain expert point of view.

3.1 Results

Starting from the initial state of Fig. 2, $\mathrm{PNet}_{\mathrm{QVT}}$, $\mathrm{PNet}_{\mathrm{fUML}}$ and $\mathrm{PNet}_{\mathrm{Kermeta}}$ produced the same execution trace (Fig. 4) showing that only Light A is functioning. Curiously the transition firing sequence was: $(start1;\ \ t1;\ \ end1)^{+}$.

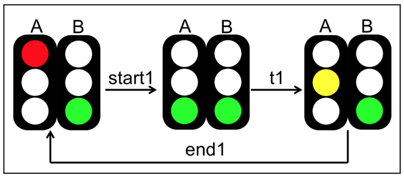


Fig. 4. First execution of the traffic light Petri-net

Often the end user or the domain expert does not have any knowledge about how the Petri-net semantics are encoded, that is why we tried again these tools starting from a more intuitive initial state where lights are set to red. In this second execution, $\mathrm{PNet}_{\mathrm{Kermeta}}$ and $\mathrm{PNet}_{\mathrm{fUML}}$ have had the same behaviour than that they exhibited in the previous case but with Light B left in state Red. $\mathrm{PNet}_{\mathrm{QVT}}$ produced a different trace, presented in Fig. 5:

$$(start1;\ \ t1;\ \ start2;\ \ end1);\ \ (start1;\ \ t1;\ \ end1)^{+}$$

In this behaviour light B is switched to green after Light A passed to orange and then after firing transition $end1$ the system is engaged in a loop similar to that of Fig. 4. Based on these behaviours it is difficult to conclude about safety properties: dead-lock freedom, mutual exclusion and fairness. In the first

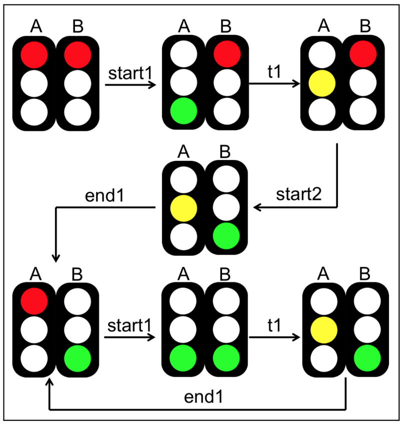


Fig. 5. Second execution of the traffic light Petrinet

execution of $\text{PNet}_{\text{Kermeta}}$ and $\text{PNet}_{\text{fUML}}$ both lights reached their critical sections together (middle state of Fig. 4), which violates the mutual exclusion property. Nonetheless, from the second execution one can conclude that this property is satisfied, which is obviously contradictory with the first execution. In the same sense, these two executions show a dead-lock freedom since the corresponding traces did not reach a blocking state, but they show too that the fairness property is not guaranteed since Light B didn't evolve at all, which is also somehow contradictory. Having these behaviours and considering that the semantics of Petri-nets is well-defined, we believe that it is difficult for the end-user – who should be in our case an expert in Petri-nets and formal methods – to adopt these tools and apply them to model a safety-critical system.

3.2 Analysis

In order to explain these behaviours we analysed the source code of our benchmark tools and we found that they do not choose in the same way the enabled transition. Indeed, in our reference algorithm the choice of the transition to fire is non-deterministic, which is not the case for these tools.

Indeterminism in $\text{PNet}_{\text{fUML}}$ and $\text{PNet}_{\text{Kermeta}}$: $\text{PNet}_{\text{fUML}}$ and $\text{PNet}_{\text{Kermeta}}$ applied a deterministic principle in which the first transition satisfying query isEnabled is fired. In $\text{PNet}_{\text{fUML}}$ [10] it is stated that: "*The run() operation repeatedly determines a list of enabled Transitions, . . . , and calls fire() for the* ***first*** *Transition in this list.*". $\text{PNet}_{\text{Kermeta}}$ source code uses the following instruction in the context of class Net:

```
transitions.findFirst[t|t.isEnabled]
```

The limitation is that collection `transitions` issued from class Net is filled sequentially depending on the order on which the modeling elements are created by the designer. In fact, in EMF references are typed by the EList data structure whose semantics are different from the Set data-structure. Actually for Fig. 2 we created the left hand side (that of Light A) before the right hand side (that of Light B) and hence we get a malfunction of Light B. Based on this observation we changed the order of transitions in the XMI file of the model, and then we get a different behaviour. We think that it is not a judicious choice to condition the DSL behaviour by the order on which modeling elements are created because it may be confusing for the domain expert. Moreover, DSL behaviour variations depending on the XMI file content would not reflect at all the behaviour of the target system, which weakens the debugging functions dedicated to a Petri-net based safety-critical controller.

Indeterminism in PNet$_{\mathrm{QVT}}$: In PNet$_{\mathrm{QVT}}$, the enabled transition is provided by the following OCL-based query:

```
query getActivated(net: Net): Transition {
        net.transitions -> any(trans | isActivated(trans))
}
```

Semantics of the "`any`" construct in OCL [3] (section 11.9, page 177) are defined as: "*Returns any element in the source collection for which body evaluates to true... If there are one or more elements for which body is true, an indeterminate choice of one of them is returned*". In the OCL reference manual the operator "any" is rewritten as follows:

```
Set->any(iterator | body) =
    Set->select(iterator | body)->asSequence()->first()
```

Conversion from Set to Sequence is non-deterministic because type set does not cover ordering. However, the EMF/OCL package uses the java structure HashSet[8] for the OCL type Set. Unfortunately, elements of a HashSet are dispersed by means of a hashing function which is called every time a modification operation (*e.g.* add, remove) is applied to the HashSet. Since in our example, the set of transitions is never modified, then this dispersion is not recomputed and the `asSequence()` operation always produced the same result. The HashSet dispersion produced from our initial Petri-net (Fig. 2) is:

$$[start1, t1, start2, end2, end1, t2]$$

This dispersion allows to understand the weird behaviours of the traffic light. Indeed, in the initial state, the set of enabled transitions gathers $start1$ and $t2$ and hence `asSequence()->first()` gets $start1$. Then, the same algorithm is applied producing a call to $t1$ followed by $end1$. Transition $t2$ would never be fired because in this dispersion it appears after transition $end1$ which brings back the model to the initial state. The similarity between the output of PNet$_{\mathrm{QVT}}$ and that of both PNet$_{\mathrm{fUML}}$ and PNet$_{\mathrm{Kermeta}}$ when Light A is red and Light B is green is hence a pure luck. This behaviour is not only unsuitable towards a

[8] HashSet is an implementation of interface Set in Java.

non-deterministic executable DSL but also dangerous because the failure comes from the execution engine not from the semantics. This failure may reduce the confidence that a domain expert may have in the DSL execution engine. Indeed, besides human errors, it is known that execution engines are the most critical parts in safety-critical systems; that is why several standards exist in order to reduce their capabilities to controllable structures and functions.

4 Formal DSL Semantics: The Meeduse Technique

The disparity between execution tools leads to behaviours that are conformant to the semantics specified by their execution models but may be far from the expected behaviour in accordance with the domain expertise. This is an important problem since the same model may not be executed in the same way on different tools even for deterministic structures. In fact, when designing a model via a given DSL tool, the domain expert focuses on debugging his model rather than debugging the DSL semantics provided by the MDE expert.

We propose an alternative definition of the Petri-net semantics using Meeduse [6], a tool that we developed in order to mix the formal B method and EMF-based DSLs. The use of a well-established formal approach assisted by provers and model-checkers, guarantees the consistency of the Petri-net DSL and its conformance to the expected behaviour. This formal reference model allows then to establish what goes well and wrong in the considered benchmark and can be useful for further improvements of existing DSL definition tools.

4.1 Functional Model

In order to get a functional B specification conformant to the Petri-net meta-model, Meeduse[9] [6] translates the meta-model into a correct by design B specification. Figure 6 gives the heading part of the generated B machine.

```
1. MACHINE
2.     nets
3. SETS
4.     NET; PLACE; TRANSITION
5. ABSTRACT_VARIABLES
6.     Net, Place, Transition,
7.     places, input,
8.     output, transitions, Place_tokens
```

Fig. 6. Heading part of the Petri-nets machine

Every meta-class leads to an abstract set (*e.g.* TRANSITION) and an abstract variable (*e.g.* Transition) which respectively represent the possible

[9] Meeduse: http://vasco.imag.fr/tools/meeduse/.

instances and the existing instances of the meta-class. Associations and class attributes lead to variables (*e.g.* places, transitions, etc.). The invariant properties generated by Meeduse are provided in Fig. 7.

```
9.  INVARIANT
10.     $Net \in \mathcal{F}\ (NET)$
11.     $\wedge\ Place \in \mathcal{F}\ (PLACE)$
12.     $\wedge\ Transition \in \mathcal{F}\ (TRANSITION)$
13.     $\wedge\ places \in Place \nrightarrow Net$
14.     $\wedge\ input \in Place \leftrightarrow Transition$
15.     $\wedge\ output \in Place \leftrightarrow Transition$
16.     $\wedge\ transitions \in Transition \nrightarrow Net$
17.     $\wedge\ Place\_tokens \in Place \nrightarrow \mathbf{NAT}$
18.     $\wedge\ \mathbf{ran}(input) = Transition$
19.     $\wedge\ \mathbf{ran}(output) = Transition$
20.     $\wedge\ \forall\ transition \cdot (transition \in \mathbf{ran}(input)$
21.         $\Rightarrow input^{-1}\ [\{transition\}] \neq \emptyset\ )$
22.     $\wedge\ \forall\ transition \cdot (transition \in \mathbf{ran}(output)$
23.         $\Rightarrow output^{-1}\ [\{transition\}] \neq \emptyset\ )$
```

Fig. 7. Invariant of the Petri-nets machine

This invariant covers structural properties defined by multiplicities and the optional/mandatory character of attributes, as well as contextual constraints like the `Token_Is_Natural` invariant. For example, predicates from line `(18.)` to line `(23.)` of Fig. 7 translate multiplicities `1..*` associated to references input and output. Attribute tokens, which is single-valued, optional and defined over the set of natural numbers, is translated into a partial function from set Place to the B type NAT (line `(17.)`).

Tools such as those of our benchmark produce an implementation from a meta-model gathering all basic operations (setters, getters, etc.) and Meeduse generates a B machine gathering similar basic operations but which are written in a theory (set theory, first order predicate logic and generalized substitutions) allowing to carry out proof of correctness. Figure 8 shows the basic setter of attribute `tokens`.

For this specification, Meeduse produced 24 operations and the AtelierB (http://www.atelierb.eu/en/) prover generated 74 proof obligations (POs) for which it was able to automatically prove 62. The 12 other POs were proved manually without improvements of the B specifications.

4.2 Execution Operations

Execution semantics often introduces complex modifications of the domain model. They may create or destroy objects, modify relationships between these objects and also update several class attributes. We are then afraid that the

```
**Place_SetTokens**(aPlace, val) =
PRE
    $aPlace \in Place \wedge val \in \mathbf{NAT}$
THEN
    $Place\_tokens :=$
        $(\{aPlace\} \lhd Place\_tokens) \cup \{(aPlace \mapsto val)\}$
END
```

Fig. 8. Basic setter of attribute tokens

difficulty in applying executable DSLs in safety-critical systems goes beyond the problem of indeterminism exhibited from our benchmark. We need a clear separation of concerns regarding properties to verify: (1) that of the meta-model with associated modeling operations, (2) that of the execution utility operations (*e.g.* addToken and removeToken), and (3) that of the coordination mechanism (*e.g.* operations fire and run of Fig. 3).

We introduce the execution semantics of the Petri-net DSL by a set of B operations shared in a machine that includes the functional machine. As the Petri-net running algorithm iterates over input and output places of a transition, we add operation getPlaces (Fig. 9) in order to return these sets given a transition *tt*. Operation getEnabled is a formalisation of query isEnabled presented in Sect. 2.2. The enabledness property of a transition tt should not only be based on the positive value of tokens (relation Place_tokens) for all input places ($\mathsf{input}^{-1}[\{\mathrm{tt}\}]$) but must also take into account the upper limit of this attribute for all output places ($\mathsf{output}^{-1}[\{\mathrm{tt}\}]$):

$$\text{(P1)}\quad \mathsf{Place\_Tokens}[\mathsf{input}^{-1}[\{\mathrm{tt}\}]] \cap \{0\} = \emptyset$$
$$\text{(P2)}\quad \mathsf{Place\_Tokens}[\mathsf{output}^{-1}[\{\mathrm{t}\}]] \cap \{\mathrm{MAXINT}\} = \emptyset$$

Precondition (P1) is not sufficient because we would like to safely increase the number of tokens in output places. Without precondition (P2), the Petri-net controller may then reach a state in which a transition is enabled, and the tokens in its input places are consumed without producing tokens in the output

```
$tEnabled \leftarrow$ **getEnabled** =
ANY $tt$ WHERE
    $tt \in Transition \wedge$
    $\{0\} \cap Place\_tokens[input^{-1}[\{tt\}]] = \emptyset \wedge$
    $\{\mathbf{MAXINT}\} \cap Place\_tokens[output^{-1}[\{tt\}]] = \emptyset$
THEN
    $tEnabled := tt$
END;
```

```
$src, trg \leftarrow getPlaces(tt) =$
PRE
    $tt \in Transition$
THEN
    $src := input^{-1}[\{tt\}]$
      $\| \; trg := output^{-1}[\{tt\}]$
END;
```

Fig. 9. Operations getEnabled and getPlaces

places. This would lead to an inconsistent Petri-net because consumption and production of tokens should not be dissociated. Both preconditions are then required in order to be able to call both addToken and removeToken when a transition is enabled.

Figure 10 gives the B specification of operation addToken (operation removeToken is somehow similar). Note that AtelierB discharged four proof obligations from this machine (two POs for the setter call, and two additional POs for the well-definedness of $Place\_Tokens(pp)$) and it was able to prove them automatically.

```
addToken(pp) =
PRE
    $pp \in Place \wedge pp \in \mathbf{dom}(Place\_tokens) \wedge$
    $Place\_tokens(pp) < \mathbf{MAXINT}$
THEN
    $\mathbf{Place\_SetTokens}(pp, Place\_tokens(pp) + 1)$
END ;
```

Fig. 10. Operation addToken

4.3 Semantics Coordination

In order to keep reasoning at a high abstraction level, operations run and fire presented as algorithms in Fig. 3, are defined as CSP[10] processes that coordinate the operations of the execution semantics. The process algebra CSP is an event-based formalism that enables description of patterns of system behaviour. In [2] combination of CSP and the B method is defined and integrated within the model-checker ProB [9]. This formalism is then useful for executable DSLs due to its abstraction capabilities and also thanks to the tool availability.

Figure 11 shows the CSP specification of the Petri-net running algorithm. This algorithm is composed of four processes: RUN, FIRE, CONSUME and PRODUCE. Process RUN (line 1.) is a recursion defined by a sequential composition with the prefixed process FIRE. In this sequence channel getEnabled?*trans* is a call to the B operation getEnabled whose output value is registered in variable *trans*. The variable is then transmitted to process FIRE. Concretely, variable *trans* represents an enabled transition provided non-deterministically by operation getEnabled. The simulation of process RUN continues indefinitely or stops when the system reaches a deadlock.

Process FIRE applied to a transition *trans* is a sequencing of processes CONSUME and PRODUCE preceded by the simple action prefix:

getPlaces!*trans*?*input*?*output*

[10] CSP: Communicating Sequential Processes [5].

```
MAIN = RUN
RUN = getEnabled?$trans$ → FIRE($trans$) ; RUN
FIRE($trans$) =
        getPlaces!$trans$?$input$?$output$ → (
                CONSUME($input$) ; PRODUCE($output$)
        )
CONSUME($input$) = $|||_{[x \in input]}$removeToken!$x$ → SKIP
PRODUCE($output$) = $|||_{[x \in output]}$addToken!$x$ → SKIP
```

Fig. 11. CSP formalisation of run and fire

This action is a call to the B operation getPlaces on transition *trans* in order to get its *input* and *output* places, which are further transmitted to processes CONSUME and PRODUCE. The objective is to apply operations removeToken and addToken to all elements of sets *input* and *output*. Notation $|||_{[x \in S]}\mathrm{Op}!x$ represents a replicated interleaving which applies all possible combinations of Op having the various valuations of parameter x taken from set S.

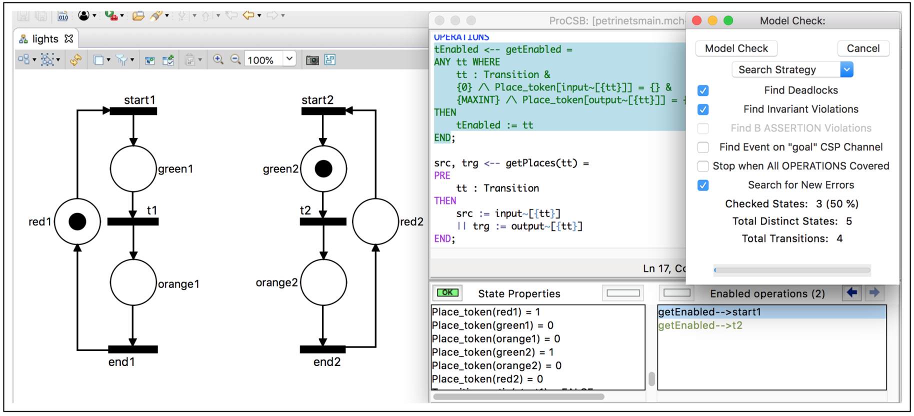
4.4 Debugging the Traffic Light

In order to debug the traffic light via our formal semantics we have two possibilities using Meeduse: (1) interactive animation, and/or (2) model-checking. Meeduse integrates ProB and EMF together in order to take benefit of the visualisation capabilities of MDE tools such as Sirius and GMF for DSLs, and the animation and model-checking functions of ProB. In Meeduse, EMF and ProB are continuously synchronised during the animation process.

The right hand side of Fig. 12 provides the ProB view and the left hand side our EMF/Sirius modeler. The ProB view shows CSP guided animation. In the current state of the model two operations are enabled: start1 and t2. In interactive animation, depending on the choice done by the user, the tool fires the selected transition and then changes the model according to the formal B specification. For every animation step, Meeduse gets the B machine state from ProB and translates it back to the EMF model in order to update the graphical view. As presented in Fig. 12, ProB offers model-checking functions allowing to find deadlocks, invariant violations and reachability of CSP goals.

Mutual Exclusion: A traffic light enters its critical section after enabling transition start and it leaves it by transition end meaning that the critical section includes states Green and Orange. In order to check this property for our petri-net model, we add the following invariant to our B specification and we ask ProB to find invariant violations and produce the corresponding transition sequences. Contrary to the observation issued from our benchmark, ProB quickly found the invariant violation showing that this property is not respected.

$$\boxed{\begin{array}{l} 1 \in \mathit{Place\_tokens}[\{\mathit{green1},\ \mathit{orange1}\}] \\ \qquad \Rightarrow \mathit{Place\_tokens}[\{\mathit{green2},\ \mathit{orange2}\}] = \{0\} \\ \wedge\ 1 \in \mathit{Place\_tokens}[\{\mathit{green2},\ \mathit{orange2}\}] \\ \qquad \Rightarrow \mathit{Place\_tokens}[\{\mathit{green1},\ \mathit{orange1}\}] = \{0\} \end{array}}$$



```
OPERATIONS
tEnabled <-- getEnabled =
ANY tt WHERE
    tt : Transition &
    {0} /\ Place_token[input~[{tt}]] = {} &
    {MAXINT} /\ Place_token[output~[{tt}]] = {
THEN
    tEnabled := tt
END;

src, trg <-- getPlaces(tt) =
PRE
    tt : Transition
THEN
    src := input~[{tt}]
    || trg := output~[{tt}]
END;
```

[✓] Find Deadlocks
[✓] Find Invariant Violations
[ ] Find B ASSERTION Violations
[ ] Find Event on "goal" CSP Channel
[ ] Stop when All OPERATIONS Covered
[✓] Search for New Errors

Checked States: 3 (50 %)
Total Distinct States: 5
Total Transitions: 4

```
Place_token(red1) = 1
Place_token(green1) = 0
Place_token(orange1) = 0
Place_token(green2) = 1
Place_token(orange2) = 0
Place_token(red2) = 0
```

```
getEnabled-->start1
getEnabled-->t2
```

Fig. 12. Integration of ProB within EMF

Fairness: To check this property we apply a parallel composition of process RUN with the process FAIRNESS defined in Fig. 13, line (12.). This process leads to two possible traces: ($step1$; $step2$; $goal$) and ($step2$; $step1$; $goal$). Channel $step1$ (respectively $step2$) is produced from process FIRE when guard $trans = end1$ (respectively $trans = end2$) holds. The objective of this specification is to reach goal STOP when the system produces a trace where both transitions $end1$ and $end2$ are fired by the RUN process.

```
1.  MAIN = RUN $|[\{step1, step2\}]|$ FAIRNESS
2.  RUN = getEnabled?$trans$ $\rightarrow$ FIRE($trans$) ; RUN
3.  FIRE($trans$) =
4.          getPlaces!$trans$?$input$?$output$ $\rightarrow$ (
5.                  CONSUME($input$) ; PRODUCE($output$)
6.          ) ;
7.          ( ($trans = end1$) : $step1$ $\rightarrow$ SKIP
8.            [] ($trans = end2$) : $step2$ $\rightarrow$ SKIP
9.            [] ($trans \notin \{end1, end2\}$) : SKIP )
10. CONSUME($input$) = $|||_{[x \in input]}$removeToken!$x$ $\rightarrow$ SKIP
11. PRODUCE($output$) = $|||_{[x \in output]}$addToken!$x$ $\rightarrow$ SKIP
12. FAIRNESS = ($step1$ $\rightarrow$ SKIP $|||$ $step2$ $\rightarrow$SKIP) ; $goal$ $\rightarrow$ STOP
```

Fig. 13. Fairness checking with CSP

ProB successfully found the expected sequences leading to the goal and showing that the system gives fair turns to lights A and B. However, given that the running algorithm is non-deterministic, it would be interesting to seek for the existence of loops where only one light runs. For this purpose, we can override the getEnabled operation in process RUN as follows:

$$\text{RUN} = \text{FIRE}(start1);\ \text{FIRE}(t1);\ \text{FIRE}(end1);\ \text{RUN}$$

Given this CSP rule, ProB explored all possible situations without finding goal STOP, which shows that the system may stay running without evolutions of Light B. This proof exhibits a weak fairness from the model.

5 Conclusion

PNet_{QVT} gave the better abstraction level however it suffers from limitations of the misuse of non-determinism. $\text{PNet}_{\text{Kermeta}}$ and $\text{PNet}_{\text{fUML}}$ have had a controllable deterministic behaviour however this choice makes them quite distant from the original Petri-net semantics. The Petri-net DSL is a "tiny" DSL and it does not allow to present all possibilities of a proof-based approach, but it was sufficient to exhibit several failures from our benchmark. Indeed, in addition to the problem of indeterminism, these tools include other unsafe behaviours due to: the implicit initialisation of the optional attribute tokens, and also the uncontrolled incrementation of this attribute that may produce an integer overflow. Similar simple failures in real-life critical systems have had disastrous consequences. To cope with these limitations, our solution applies a formal model in order to debug the DSL using the ProB animator and model-checker.

Often, in classical development processes, the use of a formal method with proofs is not widespread because it seems to create an overhead for the developer. The Meeduse approach described in this paper targets safety-critical systems where formal reasoning is widely applied even if it requires good skills in mathematics. Integration of a DSL-based solution to this field is interesting since it provides a way for rapid-prototyping of a system's behaviour without a loss of formal proofs. The alliance, favored by Meeduse, between executable DSLs and a formal method such as B, allows to reach a high level of abstraction with a good mix between expressiveness and precision. We believe that this is a promising technique to deal with the dependability of safety-critical systems.

References

1. Bousse, E., Leroy, D., Combemale, B., Wimmer, M., Baudry, B.: Omniscient debugging for executable DSLs. J. Syst. Softw. **137**, 261–288 (2018)
2. Butler, M., Leuschel, M.: Combining CSP and B for specification and property verification. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005). https://doi.org/10.1007/11526841_16

3. Object Management Group: Object Constraint Language (OCL) 2.4 Core Specification (2014). https://www.omg.org/spec/OCL/
4. Object Management Group: Meta Object Facility (MOF) 2.5.1 Core Specification (2015). https://www.omg.org/spec/MOF/2.5.1/
5. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall Inc., Upper Saddle River (1985)
6. Idani, A., Ledru, Y., Vega, G.: Alliance of model-driven engineering with a proof-based formal approach. Innov. Syst. Softw. Eng. 1–19 (2020). https://doi.org/10.1007/s11334-020-00366-3
7. Jézéquel, J.M., Combemale, B., Barais, O., Monperrus, M., Fouquet, F.: Mashup of meta-languages and its implementation in the Kermeta language workbench. Softw. Syst. Model. **14**, 905–920 (2015)
8. Langer, P., Mayerhofer, T., Kappel, G.: Semantic model differencing utilizing behavioral semantics specifications. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) MODELS 2014. LNCS, vol. 8767, pp. 116–132. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11653-2_8
9. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
10. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: executable DSMLs based on fUML. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 56–75. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_4
11. Wachsmuth, G.: Modelling the operational semantics of domain-specific modelling languages. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 506–520. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88643-3_16