Innovations in Systems and Software Engineering
https://doi.org/10.1007/s11334-021-00408-4

S.I. : DETECT 2020

Formal model-driven executable DSLs

Application to Petri-nets

Akram Idani[1]

Received: 18 December 2020 / Accepted: 21 June 2021

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. 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. To this aim, we recently developed the Meeduse tool and showed how to bridge the gap between MDE and a proof-based formal approach. In this work, we apply our approach to the Petri-net DSL and we present MeeNET, a proved Petri-net designer and animator powered by Meeduse. MeeNET is built on top of PNML (Petri-Net Markup Language), the international standard ISO/IEC 15909 for Petri-nets, and provides underlying formal static and dynamic semantics that are verified by automated reasoning tools. This paper first presents simplified MDE implementations of Petri-nets applying Java, QVT, Kermeta and fUML that we experimented in order to debug a safety-critical system and summarises the lessons learned from this study. Then, it provides formal alternatives, based on the B method and process algebra, which are well-established techniques allowing interactive animation on the one hand and reasoning about the behaviour correctness, on the other hand.

Keywords DSL · Formal methods · B Specification · Animation · Proofs

1 Introduction

In the last decade, several research works [1–3] 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 behaviour 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 language workbenches (e.g. the Gemoc Studio [4]) leading to a better quality than a static DSL. The execution of DSLs is still an active research area in model-driven engineering (MDE) with several perspectives: debugging, trace analysis, visualisation, etc.

Nowadays, developers can find a plethora of tools and approaches for building DSLs. Unfortunately, several issues related to correctness and the level of trust that one can have in these tools are still challenges for a rigorous development process. Indeed, [5] has done a survey about DSLs from 2006 to 2012, and observed that only 5.7% of the underlying techniques applied a formal analysis approach. The authors attest that "*there is an urgent need in DSL research for identifying the reasons for lack of using formal methods within domain analysis and possible solutions for improvement*". Later, [6] has done an exhaustive empirical study of DSL frameworks that are referenced in publications between 2012 and 2019, and does not report on a better situation. In fact, this study shows that among the 59 discussed tools, none of them applies formal tools, and only 9 provide supports for testing. This is an important shortcoming because it weakens the applicability of DSLs, especially for safety-critical systems. In these systems, correctness is a strong requirement and it is often addressed by the application of formal methods. To cope with this limitation, we developed Mee-

✉ Akram Idani
akram.idani@univ-grenoble-alpes.fr

1 University Grenoble Alpes, Grenoble INP, CNRS, LIG, F-38000 Grenoble, France

Published online: 11 July 2021

duse [7–9], a MDE tool built on the formal B method [10]. The tool is currently the only available tool that applies theorem proving to DSLs and jointly animates their execution. The tool embeds the model-checker and animator ProB [11] within language workbenches, which allows one to formally verify the correctness of DSLs. This paper provides more evidence about this claim by applying Meeduse and existing MDE approaches to the Petri-net DSL. This DSL is an interesting use case because on the one hand it is often used as an illustrative example of tools dealing with executable DSLs (such as the ones discussed in this paper), and on the other hand it is widely applied to develop and verify safety-critical systems.

In [12], we led an experimental study built on existing implementations of the Petri-net DSL [13–16] in QVT, Kermeta and fUML. We tried them to debug a safety-critical system and check their ability to address properties such as: non-determinism, deadlock freedom and mutual exclusion. This paper extends our previous work with two major points. First, we identify and analyse several other reasons that may lead to failures, such as: execution fairness, modelling operations, partial invariant preservation, (un)bounded types, etc., which are important when dealing with the correctness of an executable DSL. The intention is to show by practice how the limitations of the aforementioned approaches can be circumvented when the MDE approach is strengthened with well-established formal approaches such as B [10] and CSP [17] (Communicating Sequential Processes). The second contribution of this paper is the application to the reference Petri-net DSL. The paper presents MeeNET, a proved Petri-net designer and animator that is built on our previous works. MeeNET deals with PNML (Petri-Net Markup Language), the international standard ISO/IEC 15909 for Petri-nets, and provides underlying formal static and dynamic semantics that are verified by automated reasoning tools.

Section 2 describes the DSL on which we have built our experimental study. Section 3 gives our ascertainments and discusses the spectrum of this study. Section 4 applies and compares algorithms of our benchmark as they are encoded in existing works in order to debug a safety-critical model from the domain expert point of view. In Sect. 5, we exhibit several unsafe behaviours from the correctness point of view. Section 6 provides a formal alternative model, based on the B method and CSP process algebra to the Petri-net DSL. Section 7 shows how our formal alternative can be used to debug a Petri-net model. Section 8 presents MeeNET a proved Petri-net designer and animator powered by Meeduse and built on top of PNML. Finally, Sect. 9 draws the conclusions and the perspectives of this work.

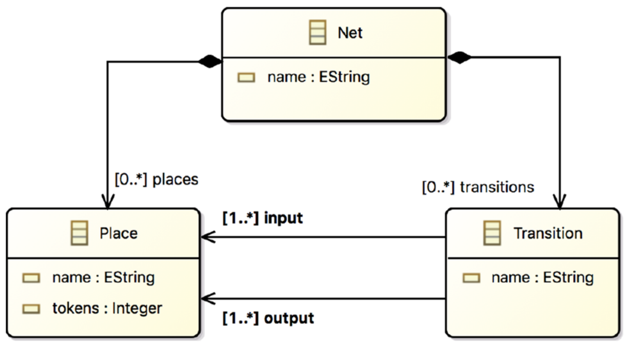


Fig. 1 Petri-nets meta-model [19]

2 A simplified Petri-net DSL

Petri-net [18] is a visual language used for modelling 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 is often used as an illustrative case by the research works and tools interested in modelling and debugging techniques, and also because it has had a wide range of applications in safety critical systems. This section builds on a simplified version of this DSL and defines a Petri-net based safety-critical example.

2.1 Structural and contextual semantics

Figure 1 shows the simplified Petri-net meta-model as considered by [13,14]. 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.

This meta-model defines structural features 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 single-valued, 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 [14][1] and it is presented without any modification. There exist variations of this meta-model where transitions admit both 0 inputs and 0 outputs. Furthermore, the Petri-net 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 Petri-net of Fig. 2 that controls 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. This model represents two traffic lights (Light 1 and Light 2) that are to be placed in two roads

[1] The corresponding ECore file can be found at [19].

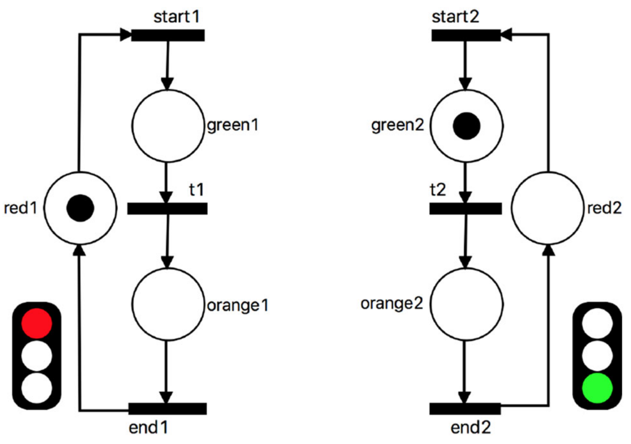


Fig. 2 Traffic light controller in Petri-nets (V1)

that intersect. These traffic lights are, respectively, controlled by the left side and the right 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 1 and green to Light 2.

In the following, we suppose that by using a Petri-net debugger, a domain expert would like to check safety properties, such as:

- correctness: asserts that the system does not exhibit bad behaviours, where invariants (structural or contextual) are violated.
- deadlock-freedom: states that the traffic lights cannot 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).

In order to debug the traffic light controller especially from the domain expert point of view, we start by applying existing MDE implementations [13–16,20,21] that used the simplified Petri-net as a use case of DSL execution and debugging.

2.2 Execution semantics

Basic execution semantics of the Petri-net DSL are defined by means of transition firing which holds when a transition satisfies an enabledness property. To check this property, we require to call a query defined as:

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

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

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

Fig. 3 Running a Petri-net [14]

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 [14], describes how a Petri-net runs.

This algorithm chooses non-deterministically (operator $:\in$) a transition t (called $t_{enabled}$) from the set of transitions that satisfy the enabledness 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

In order to debug the aforementioned safety properties using various MDE approaches and then observe their strengths and limitations, our experimental study is built on approaches that are widely accepted in the MDE literature: Java, QVT, Kermeta and fUML. In the remainder, we refer to them, respectively, as: PNet$_{\text{Java}}$, PNet$_{\text{QVT}}$, PNet$_{\text{Kermeta}}$ and PNet$_{\text{fUML}}$. Our choice is also motivated by the fact that these approaches are the only ones that we found, where the Petri-net DSL was already implemented, in addition to the availability of the underlying artefacts (source code, publications, tutorials). Furthermore, tools assisting these approaches use the Eclipse Modelling Framework (EMF), which makes easy their integration and the analysis of the Petri-net DSL within a unified MDE framework.

PNet$_{\text{Java}}$ [21]. Java-based semantics of the Petri-net DSL are proposed in [21] and supported by a tool named EProvide [22]. This java implementation is easily reproducible in the Eclipse Modelling Framework. The approach is combined with graphical editor creation in order to support rapid prototyping of animated visual interpreters and debuggers.

PNet$_{\mathrm{QVT}}$ [20]. QVT (Query/View/Transformation) is an OMG 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 [20], 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, tools that we found are either out of date (Medini QVT) or proprietary (ModelMorf). Then, for our experimental needs, we encoded a variant of rules proposed by [20] in QVT-Operational using the Eclipse Modelling Framework.

PNet$_{\mathrm{Kermeta}}$ [13,14]. Kermeta [23] is a language workbench that involves different meta-languages for abstract syntax (aligned with EMOF [24]), static semantics (aligned with OCL) and behavioural semantics. In [13,14], 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 [25].

PNet$_{\mathrm{fUML}}$ [15,16]. fUML [26] 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. [16] proposes the xMOF tool which integrates fUML with MOF to enable the specification of the behavioural semantics of DSLs in terms of fUML activities. For our experiments, we used their open-source DSL, provided at [27].

3 Ascertainment and discussion

3.1 Challenges

There exist several well-established tools [28] that implement the Petri-net theory for verification purposes and apply various programming languages. However, even if most of them are open-source, it remains difficult to update their code in order to experiment various semantics. Not only the developer must have high programming skills, but he/she must also understand the tool logic, which is a very challenging task especially for tools dedicated to formal languages. To this purpose, a MDE solution is much more suitable, because it allows one to reason on the DSL itself, rather than on how the DSL is encoded in a given programming language. Our works focus on MDE tools dedicated to DSL execution and debugging, and apply them to the Petri-net DSL, together with a formal approach. The objective is to benefit from the rich catalogue of MDE tools without losing sight of correctness and rigorous development. Indeed, several MDE tools exist: model-to-model transformation, model-to-code generation, constraint-checkers, graphical concrete syntax representation, bi-directional mappings, etc.

We start our investigation by experimenting MDE tools that present the Petri-net DSL as an application case of their underlying approaches. The approach of PNet$_{\mathrm{Java}}$ [21] proposes to support stepping back in the execution history of a DSL. PNet$_{\mathrm{QVT}}$ [20] investigated model-to-model transformations as a way to define execution semantics of a DSL. PNet$_{\mathrm{Kermeta}}$ [13,14] relies on generic trace management to provide an omniscient debugger thereby allowing developers to “go back in time” [29]. Finally, PNet$_{\mathrm{fUML}}$ [15,16] proposed a new meta-modelling language (xMOF) for specifying the abstract syntax and behavioural semantics of DSLs based exclusively on standardised modelling languages (fUML and MOF). We experimented these tools being guided by two point of views:

- The end-user point of view (Sect. 4): the end-user of a Petri-net is often either a Petri-net expert who is interested by verification features (such as model checking), or a domain expert who is interested by interactive animation and/or simulation. As mentioned above, the experimented approaches were not interested by verification, but rather by validation; and hence, we will focus on debugging rather than on proofs and model checking.
- The developer point of view (Sect. 5): we assume that the developer of a Petri-net DSL has a good knowledge of the Petri-net theory and his/her intention is to provide a bug-free tool. Thus, we will look at how existing tools encoded the semantics of the Petri-net DSL in order to locate the critical part and find the origin of failures when they happen.

In this work, we propose an alternative MDE solution, to the Petri-net DSL, whose correctness is attested by theorem proving and whose execution is managed by an animator. Section 6 presents our solution and Sect. 7 shows how the B method can help during the verification and validation activities. The challenge of this paper is therefore to show by practice how and why a formal model-driven executable DSL is developed and executed. The Petri-net DSL is widely applied in safety-critical systems for verification and simulation [28], and hence, we believe that the application of a formal method to this DSL is a strong requirement. Moreover, in order to go beyond the simplified illustrative case we applied our solution to PNML (Petri-Net Markup Language), the international standard ISO/IEC 15909 for Petri-nets (Sect. 8), which provides formal and realistic basis that may be useful for possible improvements of other MDE tools.

3.2 Observations

Unfortunately, even if our benchmark tools address several interesting features and challenging directions for DSL development, their application to the Petri-net DSL is limited to the simplified semantics. As they did not go further towards the realistic Petri-net DSL, it is difficult to objectively evaluate the usefulness of the resulting models for verification. Hence, our objective is not to check the correctness of their Petri-net case study—since their underlying approaches did not deal with correctness—but rather to build a formal MDE Petri-net and illustrate its contributions based on existing simplified implementations from which unsafe behaviours may be exhibited. Indeed, our main observation is that even if it seems easy to use MDE tools to develop the semantics of a language, the developer can miss obvious details without a good support for verification. Furthermore, this may weaken the general approach of executable DSLs when applied for debugging complex and safety-critical systems.

The analysis of our benchmark for simplified Petri-nets shows that failures or unsafe behaviours may originate from several artefacts:

- The meta-model and its underlying modelling operations, because often execution semantics apply the modelling operations to update the input model. In order to safely update the model, our approach generates a set of pre-established modelling operations that are correct by construction, meaning that they will never violate the structural features of the meta-model.
- The model itself, because incorrect behaviours may not be exhibited from models, especially when the DSL builder applies internal choices that are not conformant to the standard semantics. To cope with this issue, we have done a systematic analysis of the MOF semantics for all the constructs that are covered by our approach and provided a suitable transformation from ECore to B.
- The implementation of the execution semantics that may be distant or not conformant to the reference specification. This refers to a validation problem that requires the involvement of a domain expert in general. However, in the context of the Petri-net DSL the semantics are mathematically defined and as we apply a mathematical language to specify them, then the resulting formal models are expected to be close to what is required.
- The execution engine of the meta-language. For example, EMF/OCL-based engines wrongly support non-determinism as discussed in [30,31]. In our case, the execution engine is the ProB animator and model-checker [11]. This powerful tool has had several successful industrial applications that address safety-critical systems.

3.3 Limitations

A major limitation of this work is that the integration of a formal method within MDE tools for DSL execution is not an easy task. Indeed, a major difficulty is that usually a MDE expert does not have knowledge of formal methods; and often, proofs and model checking are not widespread because it seems to create an overhead for the developer. In [32], the authors state that: "*[...] the learning curve of formal methods is steep, whereas the learning curve for drawing diagrams on the black board is very low.*". This observation is true due to the complexity of the mathematical notations that support formal methods.

Nonetheless, when language analysers, such as compilers, are used for safety-critical or high-assurance software, [33] attests that "*validation by testing reaches its limits and needs to be complemented or even replaced by the use of formal methods such as model checking, static analysis, and program proof*". In our proposal, we advocate for collaborations between the formal methods community and the MDE community in order to take benefits of their respective tooling. Our approach [9], and its tool support Meeduse [7,8], favours this collaboration since it makes possible the use of DSL builders and formal methods tools together in one unified framework (*i.e.* EMF). The DSL tool development is the task of MDE experts who have the skills to define meta-models with associated static constraints, and the formal semantics specification is the task of the formal methods experts who are experienced in provers and model checkers.

4 Debugging the traffic light model

In this section, we apply and compare algorithms run and fire as they are encoded in our benchmark for debugging the traffic-light model from the domain expert point of view. In order to have an automated validation of the associated Petri-net controller and avoid manual trace checking, we encoded two drivers that were integrated into tools of our benchmark:

- Trace driver: registers the succession of the traffic light colours that are produced from the Petri-net model after each execution of operation Fire. This trace is then compiled into a graphical file.
- Constraint validation driver: asks EMF to validate OCL constraints after each execution of operation Fire. When activated this driver stops the runner when a critical state is reached (like that where the two traffic lights are set to green together).

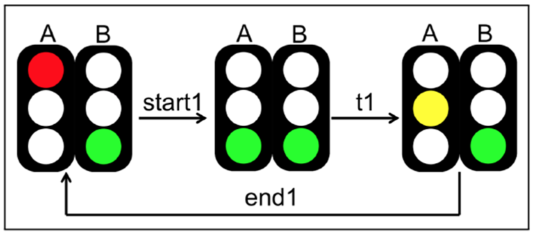


Fig. 4 First execution of the traffic light Petri-net

4.1 Execution results

Starting from the initial state of Fig. 2, $PNet_{QVT}$, $PNet_{fUML}$ and $PNet_{Kermeta}$ produced the same execution trace (Fig. 4) showing that only Light 1 is functioning. The transition firing sequence was:

$$(start1\ ;\ t1\ ;\ end1)^+$$

Regarding $PNet_{Java}$, it randomly executed several sequences and produced after a few moment all colour combinations.

Considering that DSL tools are intended to be used by domain experts, who may be in our case Petri-net experts or traffic light experts, we suppose that it is not required to know how the Petri-net semantics are encoded. In fact, domain experts often would like to debug their models rather than the DSL tool itself. Based on the reference algorithm of Fig. 3, a traffic light expert may conclude that the malfunction observed from this first execution is due to the initial state where one light is green and the other one is red. Then, we tried again these tools starting from a more intuitive initial state where both lights are set to red. In this second execution, $PNet_{Java}$ covered again all the state space and $PNet_{Kermeta}$ and $PNet_{fUML}$ have had the same behaviour that they exhibited in the previous case but with Light 2 left in state Red. $PNet_{QVT}$ produced another trace which is that of Fig. 5:

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

In this behaviour, Light 2 is switched to green after Light 1 passed to orange, and then, after firing transition *end*1 the system is engaged in a loop similar to that of Fig. 4. Having these behaviours, it is difficult to conclude about safety properties: dead-lock freedom, mutual exclusion and fairness. In the first execution of $PNet_{Kermeta}$ and $PNet_{fUML}$ 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 2 did not evolve at all, which is also somehow contradictory.

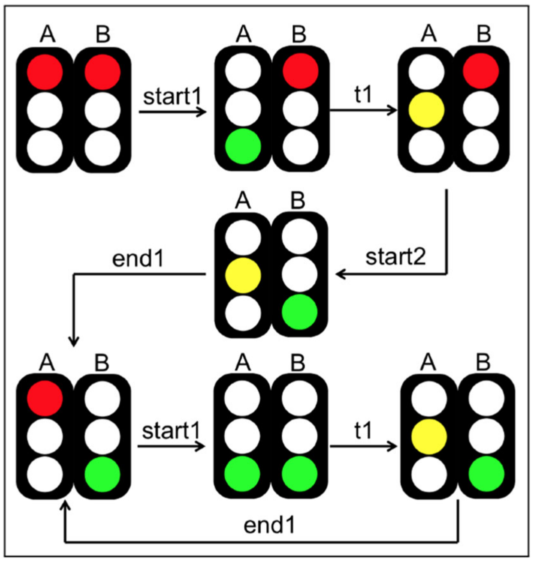


Fig. 5 Second execution of the traffic light Petri-net

These first results show that $PNet_{Kermeta}$, $PNet_{fUML}$ and $PNet_{QVT}$ do not cover all possible behaviours and hence efforts are required to identify the reasons from the implementations of our benchmark. Note that we are aware that these implementations may deliberately simplify their reference algorithm because their initial objective was neither to provide a correct Petri-net DSL nor to address safety properties when debugging a given Petri-net model. We use them mainly to illustrate several obvious points that one should address when dealing with safety and correctness. Nonetheless, for $PNet_{QVT}$ the results seem hazardous and difficult to interpret. Finally, we think that $PNet_{Java}$ has had the more realistic behaviour because it was able to reach all possible states. Its main shortcoming is its performance because it repeated several sequences and required some time to reach the nine possible colour combinations. This behaviour is normal due to non-determinism.

4.2 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 the reference algorithm the choice of the transition to fire is non-deterministic which is not the case for these tools. The closest behaviour to non-determinism is that of $PNet_{Java}$ because it applies the random function (from the java.util.Random package) on the list of enabled transitions. Although randomness and non-determinism are not similar semantically[2] [34], probabilistic algorithms remain one of the most applied techniques to put into practice non-determinism at a low level in concurrent systems.

Non-determinism in PNet$_{\text{fUML}}$ and PNet$_{\text{Kermeta}}$: in PNet$_{\text{fUML}}$ and PNet$_{\text{Kermeta}}$ a deterministic principle is applied to get the first transition satisfying query isEnabled. In PNet$_{\text{fUML}}$ [16] it is stated that:"*The run() operation repeatedly determines a list of enabled Transitions, ..., and calls fire() for the* **first** *Transition in this list.*". 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 modelling elements are created by the designer. In fact, in EMF references are typed by the EList data structure whose semantics are different from those of the Set data structure. Actually for Fig. 2, we created the left hand side (that of Light 1) before the right hand side (that of Light 2), and hence, we get a malfunction of Light 2. We got a different behaviour when we changed the order of transitions in the xmi file. We think that even if the implementation looks obviously deterministic, it remains somehow non-deterministic because one can get two different behaviours from two (visually) identical models.

Non-determinism in PNet$_{\text{QVT}}$: In PNet$_{\text{QVT}}$, 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 [35] (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[3] 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 (figure 2) is:

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

This dispersion explains the hazardous 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$_{\text{QVT}}$ and that of both PNet$_{\text{fUML}}$ and PNet$_{\text{Kermeta}}$ when Light 1 is red and Light 2 is green is hence, pure luck. Despite a non-deterministic construct, the resulting behaviour remains somehow deterministic because we always got the same result from the same model. Furthermore, even if we changed the order of the modelling elements in the xmi file, the observed behaviour remained unchanged.

4.3 Discussion

Based on our observations, we believe that the first question a DSL developer must address is: how to check that the observed behaviour is conformant to the required one? From PNet$_{\text{fUML}}$ and PNet$_{\text{Kermeta}}$, we exhibited a non-deterministic behaviour from a deterministic specification. In general, we believe that it is not a judicious choice to condition a DSL behaviour by the order on which modelling 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 the behaviour of the target system, which may weaken the debugging functions dedicated to a Petri-net-based safety-critical controller. One possible solution for PNet$_{\text{fUML}}$ and PNet$_{\text{Kermeta}}$ in order to get the expected deterministic behaviour is to introduce into the meta-model an explicit index representing transition orders, and apply an algorithm that gets the lowest index at every animation step.

The second question, in relation to the previous one, is: how to be sure that constructs of meta-languages, are executed as expected in the target platform? The problem of non-determinism is a specific problem that is naturally exhibited from our experiments because the existing solutions [30,31] are not generalised and not yet integrated within EMF. We observed that the EMF/OCL engine is unlikely to misbehave, whereas the operational semantics are enough correct. In practice, it is possible to solve this issue by using endogenous solutions such as rewriting the random function for OCL as defined in [31]. More generally, we believe

[2] A random algorithm is called probabilistic algorithm because its behaviour depends on an established number generation technique.

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

that failures produced by execution engines are dangerous. 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.

Tools of our benchmark can be improved since solutions to the limitations exhibited by this analysis exist. However, as a Petri-net DSL is intended to be applied for verification purposes in safety-critical systems, we believe that the application of a well-established formal approach is justified. In fact, the lack of support for verification is an important shortcoming of the experimented tools even for the simple Petri-net DSL. Most of the formal tools and languages that are successfully applied to the safety-critical industry have well-established semantics and carefully deal with non-determinism and its execution/simulation. Their integration within MDE tools provides a well-established paradigm for correct DSL creation.

5 Correctness

In MDE tools, correctness is often addressed by feeding various input models to the DSL runner and verifying the output behaviour. However, this empirical method does not actually guarantee that the behaviour is correct [36,37]. The only thing one can prove with this approach is that the runner is not correct because a single example of incorrect behaviour suffices. The absence of an incorrect behaviour does not allow to know whether the operational semantics are correct, or whether we have just not tested them with an input model that would trigger an error.

5.1 Correctness — the modelling point of view

As stated in [14], *operational semantics modify a given model conforming to the execution meta-model by changing values of its fields and by creating/destroying instances of meta-classes*. Consequently, to prove whether operational semantics of an executable DSL preserve the model's properties we start by checking the correctness of basic operations produced from the meta-model.

5.1.1 Basic EMF operations

In the algorithm of Fig. 3, only attribute tokens is updated. Let us then take a look at how this attribute is managed by the generated EMF java source code. Figure 6 gives a part of the java class produced by EMF from meta-class Place. First, this java setter assigns in an uncontrolled way any integer value to the class field named tokens. Second, this field is declared as protected which means that even if we correct the java setter method in order to address the OCL invariant

```java
protected static final int TOKENS_EDEFAULT = 0;
protected int tokens = TOKENS_EDEFAULT;

public void setTokens(int newTokens) {
    tokens = newTokens;
}
```

Fig. 6 Extract of the java code generated by EMF

previously called `Token_Is_Natural`, this field can be changed anyway by classes inside the same package.

This piece of code seems error-prone and tools built around it should then be carefully used. For example, the java-based editor plugin generated by EMF from the meta-model uses this basic setter and hence it allows the modeller to set any value (even negative) to this attribute. After manual checking of the java source code generated from the meta-model, we found that several other methods are worth reviewing such as the constructor of a transition that may produce a transition with neither an input nor an output which may violate multiplicities `1..*` of references input and output.

EMF provides a validation mechanism allowing to check if a given model violates or not the meta-model structural properties and the additional OCL constraints. However, the major drawback of this approach is that the semantics of a model are constrained by the java language limits. Thus, EMF may apply validation choices which are not realistic regarding the OMG specification manual [24]. For example, EMF translates in the same way optional and mandatory attributes. Indeed, attribute tokens which is single-valued and optional (*i.e.* upper-bound is equal to 1 and lower-bound is equal to 0) is not associated to a default value in the meta-model but by default EMF assigns to it value 0. We get the same java code as Fig. 6 even when we defined attribute tokens as mandatory (both upper-bound and lower-bound are equal to 1). Neither the MOF specification [24] nor the UML specification [38] advocate for an artificial choice of default values. Only the java specification [39][4] defines default values for fields and variables which are not initialised by the programmer.

5.1.2 Impact on the V&V activities

Our observations are not claimed to be limitations of EMF because the source code is standardised and follows an automatic generation mechanism. The question of initial values, for example, in programming languages is outdated and well-known today. However, building operational semantics on top of this standard source code requires some precautions in order to be conformant to the semantics. For example, having the meta-model of Fig. 1, one interesting test case for running

[4] Section 4.12.5, page 89.

a Petri-net (at the modelling level) is a model gathering one or several places without default values. While our benchmark tools do not detect any problem because a default value is internally assigned to these places, a formal validation tool would detect either an inconsistency between the meta-model and the OCL constraint `Token_Is_Natural` (because the OCL constraint requires that all values are defined) or a deficiency of the `isEnabled` query (because the query may read undefined values). Our claim is that since MDE promotes the use of models in place of coding, domain-specific tools should abstract away the implementation choices (such as those done in EMF) when dealing with models.

We believe that this issue may be highly critical especially for Petri-net models from which a controller has to be synthesised. Indeed, the target system may apply a different programming language. For example, the value of an uninitialised variable in C is indeterminate and reading or updating it can invoke undefined behaviour. In ADA, integer variables get value 0 by default (like in java), but record fields with the same type get a nondescript value. Several embedded safety-critical systems are not encoded using the Java language (*e.g.* Meteor subway [40] was developed in ADA), and hence, it may be a little dangerous to let the debugging tools make internal choices, not conformant to the semantics, and which may not reflect the final system. The practical contribution of an executable DSL is its ability to behave as the final system should run. Failures must then be exhibited or taken into account from the early abstraction level because the dependability of a system is close to that of its underlying DSL. If the designer misses some relevant aspects at the modelling stage because of a disparity between the MOF and its implementations, then the target system may produce unexplained crashes.

The permissiveness of MDE tools and the downstream validation they provide with associated programming choices may be acceptable from the modelling point of view. Indeed, in general domain experts know how to draw a correct model: they would never assign a negative value to attribute tokens, and they would never create a transition without at least one input place and one output place; even if this is possible by modelling tools. Often modelling tools allow inconsistencies and provide means to detect them. Indeed, when the model is being constructed, it may be inconsistent at some stages. However, these principles can be unsafe for execution semantics because on the one hand execution semantics are intended to reflect the system's behaviour and on the other hand in safety-critical systems when an error is identified after it happens, the consequences can be tragic. For these kinds of systems, the system (and then the model) must evolve in a safe state space which requires less permissiveness than the MDE tools and preferably an upstream validation mechanism like theorem proving.

$\text{PNet}_{\text{Kermeta}}$

```
// Adding a token from each input place
for (Place input : _self.input)
        input.tokens = input.tokens + 1

//Removing a token to each output place
for (Place output : _self.output)
        output.tokens = output.tokens - 1
```

PNet_{QVT}

```
top relation addToken {
    checkonly domain input p : Place {
       src = t : Transition{}, tokens = n : Integer{}
    };
    enforce domain output p : Place { tokens = n + 1 };
    when { fire(t) ; t.src -> excludes(p); }
}
top relation removeToken {
    checkonly domain input p : Place{
       snk = t : Transition{}, tokens = n : Integer{}
    };
    enforce domain output p : Place{ tokens = n - 1 };
    when { fire(t) ; t.snk -> excludes(p); }
}
```

Fig. 7 `addToken` and `removeToken` encoded in PNet_{QVT}

5.2 Correctness: the execution point view

5.2.1 Abstraction levels

The discussed works encoded in the same way operations `addToken` and `removeToken` but at different abstraction levels. The advantage of PNet_{QVT} and $\text{PNet}_{\text{fUML}}$ in comparison with $\text{PNet}_{\text{Kermeta}}$ and $\text{PNet}_{\text{Java}}$ is that they define semantics at a higher level which favours abstraction from implementation details. For example, the foreach loop of the transition firing algorithm (Fig. 3) is not defined as such, but with a collection definition. $\text{PNet}_{\text{Kermeta}}$ and $\text{PNet}_{\text{Java}}$ are quite close to each other with a low abstraction level. Figure 7 gives the Gemoc example of $\text{PNet}_{\text{Kermeta}}$ and the QVT relations[5] of PNet_{QVT}.

While $\text{PNet}_{\text{Kermeta}}$ explicitly defines a loop over input and output places of a transition, in PNet_{QVT} this is left to the QVT transformation engine. Indeed, the QVT engine applies the same transformation to every place p in a source model if the matched transition is fired and if the matched place is not an input place of this transition (a when clause in QVT-R specifies a condition under which a relation applies).

5.2.2 Invariant preservation

In order to check the correctness of operations `addToken` and `removeToken` as defined in the implementations of

[5] In listing of Fig. 7, src and snk correspond to references input and output of our meta-model. In QVT-R, input and output are reserved words. Note that this is the original specification taken from [20], but in our experiments, we have encoded a variant of this listing in QVTo.

our benchmark, we should be able to prove the following formulas:

- PO_1: $tokens \in \mathsf{NAT} \Rightarrow$
  $[tokens := tokens - 1](tokens \in \mathsf{NAT})$
- PO_2: $tokens \in \mathsf{NAT} \Rightarrow$
  $[tokens := tokens + 1](tokens \in \mathsf{NAT})$

These proof obligations mean that starting from a state where the invariant $tokens \in \mathsf{NAT}$ is true, then both substitutions (between brackets) lead to a state establishing this invariant. They can be written as follows by computing their weakest precondition:

- PO_1: $tokens \in \mathsf{NAT} \Rightarrow (tokens - 1) \in \mathsf{NAT}$
- PO_2: $tokens \in \mathsf{NAT} \Rightarrow (tokens + 1) \in \mathsf{NAT}$

PO_1 and PO_2 can be reduced as follows after replacing variable tokens with the limit bounds of type NAT, meaning that they cannot be proved because their weakest preconditions are not established for the lower bound (zero) and the upper bound (maxint) of natural numbers:

- PO_1: $0 \in \mathsf{NAT} \Rightarrow (0 - 1) \in \mathsf{NAT}$
- PO_2: $maxint \in \mathsf{NAT} \Rightarrow (maxint + 1) \in \mathsf{NAT}$

Obviously, in both cases the result is: $true \Rightarrow false$, which is equivalent to *false*. This simplistic analysis shows that in order to have a correct execution of operations `addToken` and `removeToken`, the corresponding assignments $[tokens := tokens - 1]$ and $[tokens := tokens + 1]$ must be encoded under the following preconditions:

- Precondition for `removeToken`: $tokens > 0$
- Precondition for `addToken`: $tokens < maxint$

Indeed, with these preconditions the above proof obligations become:

- PO_1: $tokens > 0 \wedge tokens \in \mathsf{NAT} \Rightarrow$
  $[tokens := tokens - 1](tokens \in \mathsf{NAT})$
- PO_2: $tokens < maxint \wedge tokens \in \mathsf{NAT} \Rightarrow$
  $[tokens := tokens + 1](tokens \in \mathsf{NAT})$

The simplification is then as follows, which is equivalent to true in both cases:

- PO_1: $1 \in \mathsf{NAT} \Rightarrow (1 - 1) \in \mathsf{NAT}$
- PO_2: $maxint - 1 \in \mathsf{NAT} \Rightarrow (maxint - 1 + 1) \in \mathsf{NAT}$

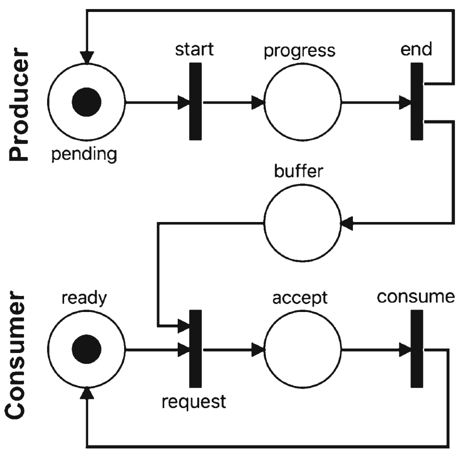


Fig. 8 Unbounded Producer/Consumer Petri-nets

5.2.3 Discussion

Tools of our benchmark implicitly take into account the precondition of `removeToken` since operation fire is called only for enabled transitions which are transitions where the input places contain at least one token. Nonetheless, they encoded the assignment $[tokens := tokens + 1]$ without the precondition that guarantees its correctness. As a result, they all crashed for the example of Fig. 8 which addresses the (un)bounded-buffer problem of the producer/consumer synchronisation [41]. This Petri-net describes two processes, the producer (top side of Fig. 8) and the consumer (bottom side of Fig. 8), who share a common buffer. In this model, sequence $(start; end)*$ increases the number of tokens in place buffer. If the consumer stays in state ready without firing transition request, the system would reach its limit. Then, the normal behaviour is that the producer stops producing and waits for a buffer release by the consumer. The simplest way to reach this limit is to set the number of tokens in place buffer to Integer.MAX_VALUE. In Java, Integer.MAX_VALUE + 1 produces value −2147483648 and depends on the execution environment. The general case in programming languages (like operation Math.addExact(MAX_VALUE, 1)) gives rise to an arithmetic overflow.

$PNet_{Kermeta}$ and $PNet_{fUML}$ produced quickly the invariant violation. $PNet_{Java}$ produced several behaviours: sometimes it took a while before reaching the wrong state, sometimes one execution was sufficient and sometimes it moved away from this state. Although the negative value produced by these executions $PNet_{Kermeta}$, $PNet_{fUML}$ and $PNet_{Java}$ did not stop. The running mechanism continues and the producer continues producing until the number of tokens becomes again positive. Indeed, when the number of tokens in place buffer is negative, the enabledness property of transition request is evaluated to false, and then, the consumer is blocked while the producer produces even if the system

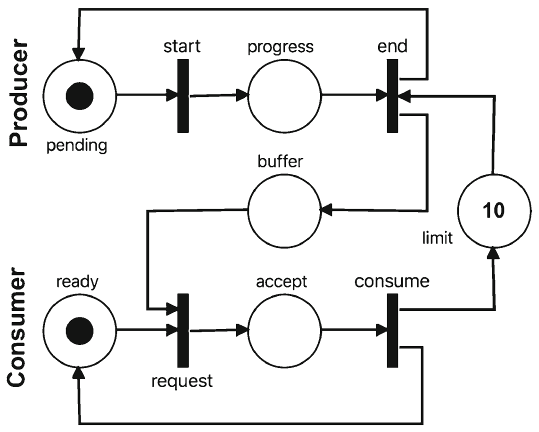


Fig. 9 Bounded Producer/Consumer Petri-nets

capacity is bypassed. PNet_{QVT} was surprising because it only decreased the number of tokens until this number reached value zero by running sequence $(\mathit{request}; \mathit{consume})*$. After that it indefinitely executed sequence:

$$(\mathit{start}; \mathit{end}; \mathit{request}; \mathit{consume})*$$

The Petri-net of Fig. 8 is a limit testing case because the production and the accumulation of tokens in the buffer is unbounded. The overflow bugs may be hard to discover because they may manifest themselves only for large input data sets, which are less likely to be addressed by testing cases. A formal approach assisted by a theorem prover allows the developer to find these kinds of limitations thanks to proof obligations. A more realistic situation can be obtained by considering a buffer of limited capacity such as in Fig. 9 where the buffer limit is fixed to 10. In this model, sequences $(\mathit{start}; \mathit{end})*$ and $(\mathit{request}; \mathit{consume})*$ would be triggered at the most 10 times each before stopping producing or consuming. As expected, for the bounded Producer/Consumer Petri-net and also the traffic light example (Fig. 2), DSLs of our benchmark did not crash because the model does not reach the limit of type Integer (Integer.MAX_VALUE $= 2^{31} - 1$).

In real-life systems, Petri-nets were used to specify embedded controllers for systems like air/rail traffic or nuclear reactors [42] and devices with restricted resources where the execution mechanism does not run in a JVM (such as in the EMF framework). Reaching an exception may then damage the system with disastrous consequences. In fact, several experiences in safety-critical systems showed that typing limitations are not as simple as they sound. For example, Ariane 501 rocket launched by the European Space Agency exploded just forty seconds after its lift-off. The reason was an integer assignment badly managed regarding the devices memory restriction leading to an overflow exception [43].

```
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. 10 Heading part of the Petri-nets machine

6 A formal model-driven Petri-net DSL

The disparity between DSL execution tools leads to behaviours that are conformant to the semantics specified by their execution models but which 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.

In this work, we propose an alternative definition of the Petri-net semantics (that we call $\text{PNet}_{\text{Reference}}$) using Meeduse [8,9], a MDE 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, would at least guarantee the consistency of the Petri-net DSL and its conformance to the expected semantics (other benefits will be discussed later). The proposed formal model-driven Petri-net DSL builds on the B method [10] and the CSP [17] process algebra (Communicating Sequential Processes). It allows us to establish rigorous foundations to further improvements of existing DSL definition tools interested by Petri-nets.

6.1 Functional model

In order to get a functional B specification conformant to the Petri-net meta-model, Meeduse translates the meta-model into a correct by design B specification. Figure 10 gives the heading part of the generated B machine.

Every meta-class leads to an abstract set (*e.g.* TRANSITION) and an abstract variable (*e.g.* Transition) that, respectively, represent the possible instances and the existing instances of the meta-class. Associations and class attributes lead to variables (*e.g.* places, transitions, etc). Finally, the static invariants generated by Meeduse are provided in Fig. 11.

This invariant covers structural properties defined by multiplicities and the optional/mandatory character of attributes,

```
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. 11 Invariant of the Petri-nets machine

as well as contextual constraints like the `Token_Is_Natural` invariant. For example, predicates from line `(18.)` to line `(23.)` of Fig. 11 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** (see line `(17.)`). Note that in EMF this type does not exist, which then requires the additional OCL constraint in the EMF-based tools. In Meeduse, it is possible to mix predefined EMF/UML basic types with those defined in the B language.

6.2 Modelling operations

Our objective is to use Meeduse in order to reproduce the EMF process for editing models, but in the rigorous world of a formal method. Indeed, EMF generates a Java 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 generalised substitutions) allowing to carry out proof of correctness. Figure 12 shows the basic setter of attribute `tokens` produced by Meeduse. By default, this setter preserves invariant properties since it affects a natural number to attribute `tokens` thanks to precondition: $val \in$ **NAT**.

Figure 13 gives two cases of the place creation operation that depend on the optional/mandatory character of attribute tokens. An attribute without an initial value in the meta-model leads to two possibilities: (1) if the attribute is mandatory then the B constructor requires a parameter to assign an initial value to the attribute and (2) if it is

```
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. 12 Basic setter of attribute tokens

optional then the constructor creates a place with an undefined attribute value.

Note that the meta-model used in PNet$_{\text{fUML}}$ defines attribute tokens as single-valued, mandatory with a default value equal to 0. In this case, even if the attribute is optional, our tool produces the following constructor for class Place:

As we will focus in the following on attribute `tokens`, we present only these operations among the 24 operations generated automatically. These B operations are conceptually more accurate than the unique java translation used in EMF-based DSL tools because they guarantee the preservation of the structural features of the meta-model. For this specification, the AtelierB [44] prover generated 74 proof obligations (POs) for which it was able to automatically prove 62. The 12 other POs were proved interactively without any improvement of the B specifications.

6.3 Execution semantics

Tools of our benchmark used (meta-)programming languages with algorithmic facilities in order to define the operational semantics. In our reference model, we use additional B operations to those generated automatically and we propose to coordinate them using CSP process algebra. Besides theorem proving, this formal background, allows one to apply associated animation and model-checking tools in order to rigorously debug and verify the various models.

Execution semantics often introduce more complex modifications to the domain-specific model than those discussed for attribute tokens: they may create or destroy objects, modify relationships between these objects and also update several class attributes at the same time. We are then afraid that the difficulty in applying executable DSLs in safety-critical systems goes beyond the overflow typing problem or the implicit assignment of default values, or even non-determinism. In our opinion, this issue needs a methodological background and a clear separation of concerns regarding the properties to verify:

1. That of the meta-model with associated modelling operations (*e.g.* setToken,…),
2. That of the execution utility operations (*e.g.* addToken, removeToken, and transition enabledness),

Fig. 13 Constructors of class Place

Mandatory $Place \rightarrow \mathbf{NAT}$

```
Place_NEW(aPlace, val) =
PRE
    aPlace ∈ PLACE ∧
    val ∈ NAT ∧
    aPlace ∉ Place
THEN
    Place := Place ∪ {aPlace} ||
    Place_tokens :=
        Place_tokens ∪ {(aPlace ↦ val)}
END;
```

Optional $Place \nrightarrow \mathbf{NAT}$

```
Place_NEW(aPlace) =
PRE
    aPlace ∈ PLACE ∧
    aPlace ∉ Place
THEN
    Place := Place ∪ {aPlace}
END;
```

3. That of the coordination mechanism (*e.g.* operations fire and run of Fig. 3).

The first kind of property is covered by the functional model (machine nets of Fig. 10) extracted automatically from the meta-model. The resulting B specification gathers modelling operations and can be enhanced by additional invariants in order to take into account contextual properties and then carry out proofs of correctness for these operations. The two other kinds of properties would be introduced by an expert in formal methods based on this functional model. This is a similar approach to a classical MDE technique where first EMF generates a Java API from a meta-model and then frameworks for executable semantics suggest the use of specific languages (like QVT, Kermeta, etc). In our approach, this specific language is that of the B method.

We introduce the execution semantics of the Petri-net DSL by a set of B operations shared in an additional machine that we call semantics in the following. This machine includes the functional machine nets in order to be able to explicitly call its operations. Contrary to the java protected field produced by EMF from attribute tokens, in B the only way to modify a variable outside the machine in which it is defined, is to use operations provided by the latter. The Petri-net implementations of our benchmark are simple cases because they should address mainly two basic properties: the first one about attribute tokens that belongs to natural numbers was discussed in the previous section, and the second one is about transition enabledness. This property should not only take into account the positive value of tokens (relation Place_tokens) for all input places ($\text{input}^{-1}[\{\text{tt}\}]$) of a transition tt but must also take into account the upper limit of this attribute for all output places ($\text{output}^{-1}[\{\text{tt}\}]$). Operation getEnabled (Fig. 15) is a formalisation of query isEnabled presented in section 2.2. Substitution ANY …WHERE …END gets any transition tt satisfying preconditions:

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

```
Place_NEW(aPlace) =
PRE
    aPlace ∈ PLACE ∧
    aPlace ∉ Place
THEN
    Place := Place ∪ {aPlace} ||
    Place_tokens :=
        Place_tokens ∪ {(aPlace ↦ 0)}
END;
```

Fig. 14 Mandatory/Optional attribute initialised to 0

```
tEnabled ← getEnabled =
ANY tt WHERE
    tt ∈ Transition ∧
    {0} ∩ Place_tokens[input ⁻¹ [{tt}]] = ∅ ∧
    {MAXINT} ∩ Place_tokens[output ⁻¹ [{tt}]] = ∅
THEN
    tEnabled := tt
END;
```

Fig. 15 Operation getEnabled

Since relation Place_tokens is defined over natural numbers (Fig. 11), it is not necessary to use the forAll primitive inside the precondition like in the OCL query. The precondition of operation getEnabled simply verifies if values 0 and MAXINT belong to the sets of tokens issued from the input and output places of tt. Precondition (P1) is not sufficient because we would like to safely increase the number of tokens in the 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 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 16 gives the B specification of operations addToken and removeToken.

```
removeToken(pp) =
PRE
    pp ∈ Place ∧ pp ∈ dom(Place_tokens) ∧
    Place_tokens(pp) > 0
THEN
    Place_SetTokens(pp, Place_tokens(pp) − 1)
END;

addToken(pp) =
PRE
    pp ∈ Place ∧ pp ∈ dom(Place_tokens) ∧
    Place_tokens(pp) < MAXINT
THEN
    Place_SetTokens(pp, Place_tokens(pp) + 1)
END ;
```

Fig. 16 Operations addToken and RemoveToken

```
src, trg ← getPlaces(tt) =
PRE
    tt ∈ Transition
THEN
    src := input ⁻¹ [{tt}]
    || trg := output ⁻¹ [{tt}]
END;
```

Fig. 17 Operation getPlaces

Every operation has three preconditions allowing the success of the POs computed by the AtelierB prover:

- the parameter typing predicate $pp \in Place$,
- the definition domain of the tokens number (greater than zero or less than maxint), and
- predicate $pp \in dom(Place\_Tokens)$ which asserts that attribute tokens is assigned to a value for place pp. This predicate guarantees the well-definedness PO of application $Place\_Tokens(pp)$. Indeed, in order to be able to read the value of attribute tokens from a given place, the program must verify that the value is not undefined for that place as discussed in Sect. 5.1.

Both operations addToken and removeToken call the basic setter Place_SetTokens (Fig. 12) of the functional model nets, and hence, they preserve the meta-model invariant properties. As the Petri-net running algorithm iterates over input and output places of a transition, we enhance machine semantics by an additional getter (Fig. 17) which returns these sets given a transition tt. Finally, 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.

Operator	Syntax
stop	$STOP$
skip	$SKIP$
prefix	$a \rightarrow Q$
conditional prefix	$a?x : C \rightarrow P$
external choice	$P \Box Q$
internal choice	$P \sqcap Q$
interleaving	$P \vert\vert\vert Q$
parallel composition	$P \lbrack\vert A \vert\rbrack Q$
sequential composition	$P; Q$

Fig. 18 Some commonly used CSP operators

6.4 Semantics coordination

Machine semantics provides B operations that are proved correct, and hence, we have the guarantee that any running algorithm based on these operations will not lead to violations of the model's properties. The notion of algorithm in the oxford dictionary [45], is defined as: "*an algorithm is a process or set of rules to be followed in calculations or other problem-solving operations*". In order to keep reasoning at a high abstraction level, and be faithful to this definition, operations run and fire presented as algorithms in Fig. 3, will be defined as CSP[6] processes that coordinate operations of machine semantics. The process algebra CSP is an event-based formalism that enables description of patterns of system behaviour. In [46], combination of CSP and the B method is defined and integrated within the model-checker ProB [11]. This formalism is then useful for executable DSLs due to its abstraction capabilities and also thanks to the tool availability. Figure 18 provides the main CSP constructs. In this paper, we apply the following ones:

- simple action prefix $a \rightarrow P$ where a is a B operation name (called channel in CSP) possibly followed by a sequence of outputs ($!v$) and inputs ($?var$) such that v is a value expression and var is a variable identifier.
- sequential composition ($P \;;\; Q$), meaning that the execution of process Q follows that of process P.
- process interleaving ($P \;|||\; Q$) where the resulting execution traces are the arbitrary interleaving of traces issued from each process.
- guarded process ($g : P$) which is the execution of process P under a condition defined by guard g.

Figure 19 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 2.) is a recursion defined by a sequential composition with the prefixed process FIRE. In this sequence

[6] CSP: Communicating Sequential Processes [17].

```
1. MAIN = RUN
2. RUN = getEnabled?$trans$ → FIRE($trans$) ; RUN
3. FIRE($trans$) =
4.         getPlaces!$trans$?$input$?$output$ → (
5.                 CONSUME($input$) ; PRODUCE($output$)
6.         )
7. CONSUME($input$) = $|||_{[x \in input]}$removeToken!$x$ → SKIP
8. PRODUCE($output$) = $|||_{[x \in output]}$addToken!$x$ → SKIP
```

Fig. 19 CSP formalisation of run and fire

channel getEnabled?$trans$ is a call to the B operation getEnabled whose output value is registered in variable $trans$ which 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:

$$\text{getPlaces!}trans?input?output$$

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. These two processes apply, respectively, operations removeToken and addToken to all elements of sets $input$ and $output$. Notation $|||_{[x \in S]}\text{Op!}x$ represents a replicated interleaving which applies all possible combinations of Op having the various valuations of parameter x taken from set S.

7 Debugging with PNet$_{\text{Reference}}$

In this work, we use our tool Meeduse [7] in which ProB and EMF are integrated to take benefit of the visualisation capabilities of tools like Sirius and GMF, and the animation and model-checking functions of ProB. In Meeduse, EMF and ProB are synchronised at every animation step using a notification mechanism. For more details about the tool, we refer the reader to [8,9]. Our objective is to debug the traffic light via PNet$_{\text{Reference}}$. We have two possibilities applying the ProB tool:

1. interactive animation, which is useful for domain experts,
2. model checking, which allows sophisticated analysis of reachability properties.

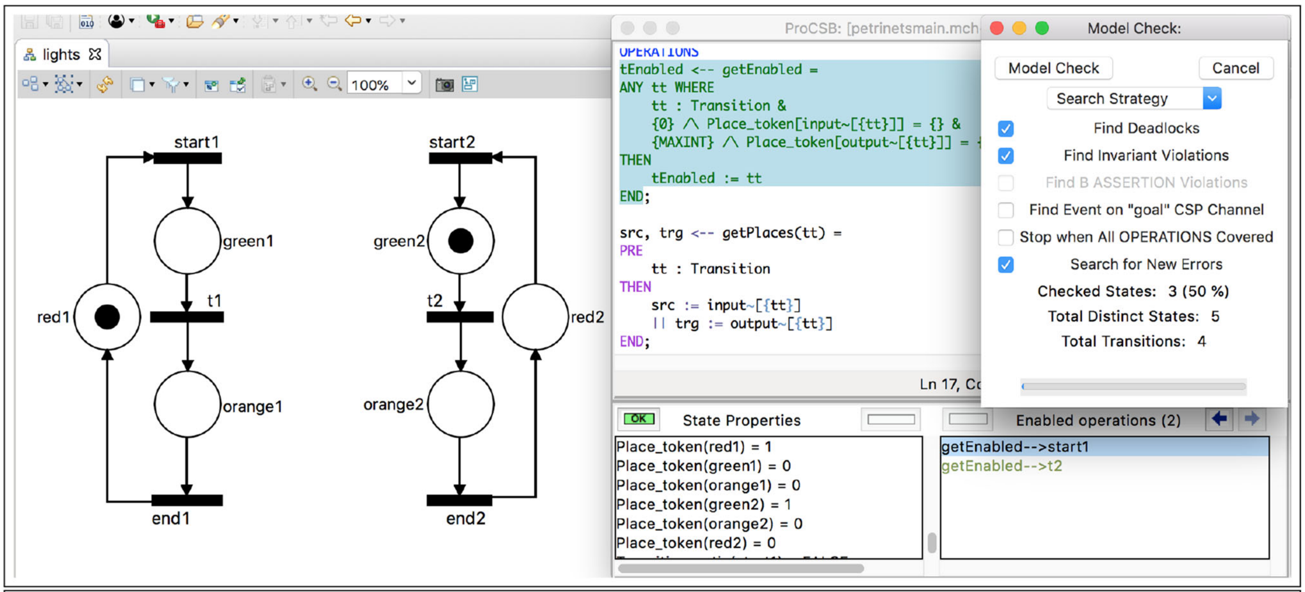
First, Meeduse interprets an input EMF model, such as the traffic light model of Fig. 1 and injects its elements as valuations in the formal specification of the meta-model. These valuations (bottom part of Fig. 20) allow ProB to initialise the B machine and start animation. The right side of Fig. 20 provides the ProB view and the left side shows our Sirius modeller that we developed for the simplified Petri-net case study. The ProB view shows CSP guided animation of machine semantics. 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. 20, ProB offers model-checking functions allowing to find deadlocks, invariant violations and reachability of CSP goals.

Mutual exclusion: The mutual exclusion property can be expressed by the following invariant which excludes a state where traffic lights are simultaneously in their critical sections. 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:

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

In order to check whether this mutual exclusion property holds or not for our Petri-net model, we add this invariant to machine semantics and we ask ProB to find invariant violations starting from several possible initial states. ProB successfully found all the expected invariant violations and also produced the corresponding transition sequences (called counter-examples). ProB provides several interesting facilities to debug a formal specification in order to improve it. Figure 21 shows the improved Petri-net model where place sync is added between transitions $start$ and end. Note that currently Meeduse does not provide means to apply the feedback produced by ProB (such as trace analysis, invariant decomposition, etc.) to the input EMF model. The introduction of place sync is hence done after checking the trace produced by ProB when it found the invariant violation. From the initial state of Fig. 20, both $start1$ and $start2$ can be enabled because their input places have the required number of tokens. If $start1$ (respectively, $start2$) is fired then the token of place sync will be consumed which forbids Light 2 (respectively, Light 1) to enter its critical section. This token will be restored after firing transition $end1$ (respectively, $end2$). From this Petri-net model, ProB did not find any invariant violation after visiting all possible nodes of the state space. This proof guarantees the mutual exclusion property of the improved traffic-light.

Fairness: In order to check this property we apply a parallel composition of process RUN with the process FAIRNESS



```
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
```

Place	{red1,green1,orange1,green2,orange2,red2}
Transition	{start1,t1,end1,start2,end2,t2}
input	{(red1↦start1),(green1↦t1),(orange1↦end1),(green2↦t2),(orange2↦end2),(red2↦start2)}
output	{(red1↦end1),(green1↦start1),(orange1↦t1),(green2↦start2),(orange2↦t2),(red2↦end2)}
transitions	{(start1↦lightSynch),(t1↦lightSynch),(end1↦lightSynch),(start2↦lightSynch),(end2↦lightSynch),(t2↦lightSynch)}
places	{(red1↦lightSynch),(green1↦lightSynch),(orange1↦lightSynch),(green2↦lightSynch),(orange2↦lightSynch),(red2↦lightSynch)}
Place_token	{(red1↦1),(green1↦0),(orange1↦0),(green2↦1),(orange2↦0),(red2↦0)}

Fig. 20 Integration of ProB within EMF

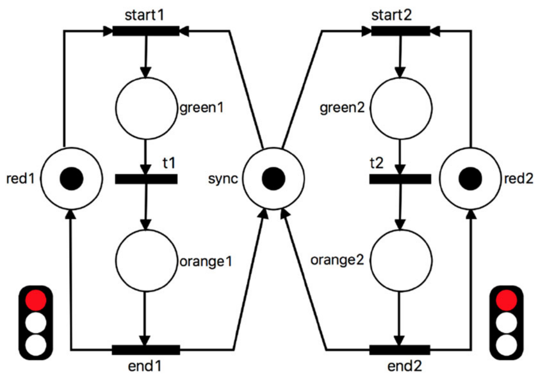


Fig. 21 Improved Petri-net for mutual exclusion (V2)

defined in Fig. 23, 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 stop the running algorithm when goal STOP is reached, which means that the system produced a trace where both transitions $end1$ and $end2$ are fired by the RUN process. For the example of Fig. 21, ProB successfully produced the expected sequences leading to the CSP goal process and showing that the system gives fair turns to lights 1 and 2. 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:

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

From this CSP rule, ProB covered the whole state space and did not find a deadlock showing that the system may stay running without evolutions of Light 2. This proof exhibits a weak fairness from the model. To improve the traffic light Petri-net, the domain expert can introduce a sequencing mechanism (Fig. 22) and replays the fairness checking, first with the CSP specification of Fig. 23 for non-regression and next with the overriding of rule RUN as mentioned above. The latter ends with a deadlock because after FIRE($end1$) it is not possible to go back and run again FIRE($start1$). The fairness checking with the CSP specification of Fig. 23 provides the same result as previously meaning that this Petri-net controller gives fair turns to lights 1 and 2 without any loop where only one light runs.

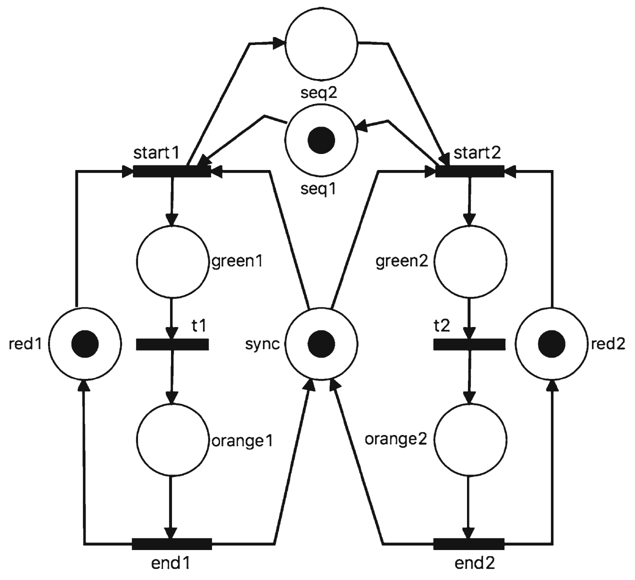


Fig. 22 Final traffic lights Petri-net (V3)

Note that by deleting place sync from Fig. 22, the resulting execution traces, with regard to the elements issued from the initial model, are included in the ones issued from Fig. 21. We can therefore assume that the fairness property is a refinement of the mutual exclusion property, and hence reduce the Petri-net model of Fig. 22 by deleting place sync.

8 Towards the realistic formal Petri-net DSL

The previous sections showed by practice how a formal model-driven executable Petri-net DSL can be developed, executed and debugged. The discussed examples, including those of our benchmark, are mainly illustrative and were used to point out a major limitation of MDE tools; the lack of support for verification. Our approach introduces within these tools formal reasoning, which seems highly required especially when the DSL tool, such as a Petri-net-based tool, has to be used for safety-critical systems. In the following, we present MeeNET[7] our formal Petri-net designer and animator, that is powered by Meeduse and built on top of PNML (Petri-Net Markup Language), the international standard ISO/IEC 15909 for Petri-nets.

8.1 PNML

The Petri-net community has worked several years ago on a standardised Petri-net DSL, called PNML (Petri Net Markup Language) [47,48]. The language provides an agreed-on interchange format that is compliant with a formal definition of Petri-nets. The standardisation process ISO/IEC 15909 was set up by the community in order to provide a commonly used specification for Petri-nets. The standard covers three classes of Petri-nets: High-Level Petri Nets (HLPN), Symmetric Nets and Place/Transition Nets (PT-Nets). In the previous sections, the simplified models belong to the third class (PT-nets).

We believe that for scalability it is important, for MDE tools and works that are dedicated to executable DSLs, to go beyond the illustrative case study and propose realistic applications based on PNML. Indeed, these tools have reached a good level of maturity and provide several powerful facilities such as omniscient debugging, trace optimisation, visualisation, etc. Furthermore, open-source implementations of PNML and the underlying meta-models are already integrated within EMF. There exist two major platforms: The ePNK [49] and PNML Framework [50]. In our work, as far as an ECore file of the meta-model is defined, Meeduse can be used, first to extract the formal static semantics from the file and prove its correctness, and then to execute input models if the formal B machine of the execution semantics is provided.

The aim of MeeNET is to provide a scalable executable formal specification of Petri-nets and experiment our approach on realistic input models. The underlying formal specification is currently experimented within The ePNK because this platform provides an extension point, so that new Petri-net types can be plugged in to the existing tool without touching the code of The ePNK. This mechanism is interesting from the Meeduse point of view because it is a kind of DSL refinement, which is a challenging topic for DSL execution and verification. As refinements are apart of the B method, then the tool opens interesting perspectives to this work. We partially address this perspective in the remainder.

8.2 The PNML Meta-model

PNML identifies concepts that are common to the three classes of Petri-nets in a so-called PNML Core model (Fig. 24). The common concepts are mainly places, transitions and arcs, and these objects can have some kind of label. The PNML core model provides means for splitting up larger Petri-nets into pages; connections between nodes on different pages can be established by reference places or reference transitions. PNML defines all kinds of graphical information that can be attached to the different elements, such as position, size, font-type and font-size. Our objective is to formalise the executable part of PNML, so not all the PNML concerns are taken into account. For this reason Meeduse provides an annotation mechanism allowing the DSL developer to select the concepts to be translated into B. This mechanism provides also a way to precise the namings and to strengthen the properties of the meta-model such as multiplicities when

[7] Demo videos can be found at: http://vasco.imag.fr/tools/meeduse/meenet/.

Fig. 23 Fairness checking with CSP

```
MAIN = RUN |[{step1, step2}]| FAIRNESS
RUN = getEnabled?trans → FIRE(trans) ; RUN
FIRE(trans) =
        getPlaces!trans?input?output → (
                CONSUME(input) ; PRODUCE(output)
        ) ;
        ( (trans = end1) : step1 → SKIP
          [] (trans = end2) : step2 → SKIP
          [] (trans ∉ {end1, end2}) : SKIP )
CONSUME(input) = |||[x∈input] removeToken!x → SKIP
PRODUCE(output) = |||[x∈output] addToken!x → SKIP
FAIRNESS = (step1 → SKIP ||| step2 →SKIP) ; goal → STOP
```

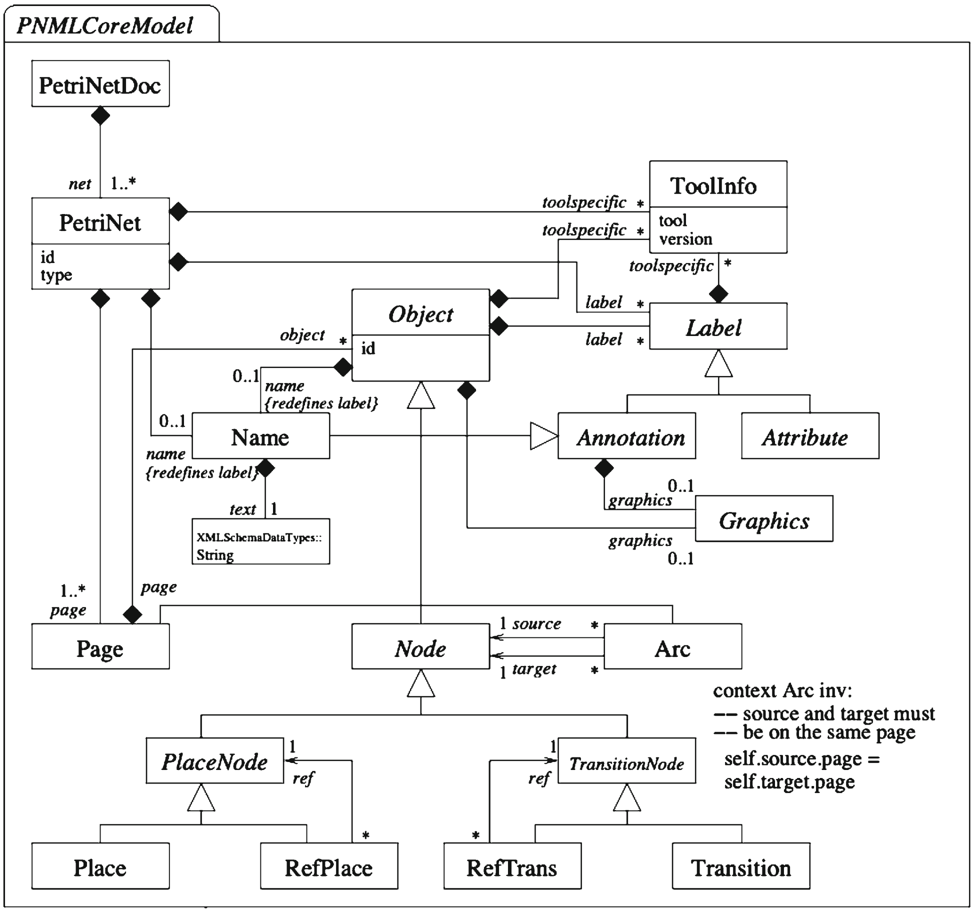


Fig. 24 The PNML core model of ISO/IEC 15909-2 (2011) [51]

required. Note that the current version of MeeNET does not yet cover the splitting of Petri-nets over pages, it is rather focused on the basic constructs. Classes RefPlace and RefTrans, for example, are not transformed into B.

In The ePNK, the PT-Nets class of Petri-nets is defined using an extension mechanism called Petri-net type definition (PNTD). Figure 25 shows the corresponding meta-model. It is the purpose of the PNTD to define the specialisations of meta-class Label that are possible in a specific class of Petri-nets, and also to define the additional restrictions on the legal connections. Figure 25 introduces two additional kinds of labels for Place/Transition systems: the initial marking for places, and the inscription for arcs. The initial marking can be any natural number (including 0) and the inscription for arcs can be any positive number. Technically, Figs. 24 and 25 are two distinct meta-models, but one is the extension of the other (PT-Nets is the extension of PNML Core). In our approach, we support this extension mechanism using B refinements.

8.3 Formal static semantics

In B, there are two kinds of refinements: behaviour refinement and data refinement.

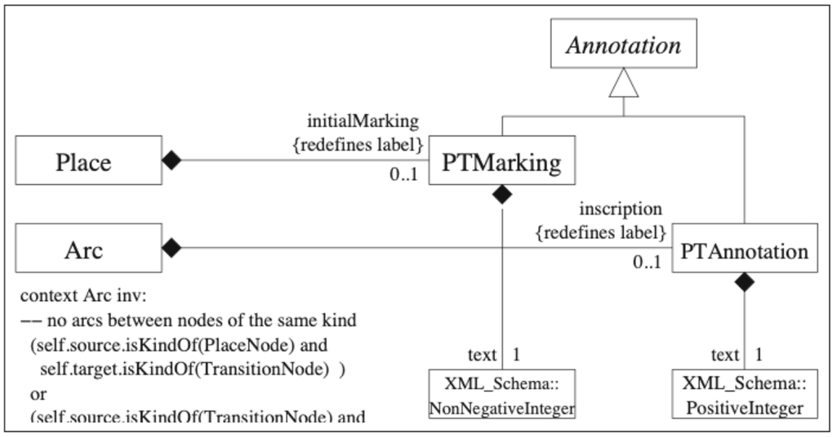


Fig. 25 The PNTD for PT-Nets [49]

– Behaviour refinement means that an algorithm is changed by another one but without violating neither the conditions under which the initial algorithm is defined nor its possible executions. This kind of refinement would be useful to introduce step by step execution semantics of a DSL. It is not illustrated in this work; we have defined the formal model of the PT-nets execution semantics once and for all.
– Data refinement applies a new set of data in the refined model, that is some data can be replaced and some data can be added. This kind of refinement would be useful to introduce incrementally the static semantics of a DSL. This is suitable for the PT-nets class because its data are defined by introducing some additional meta-classes and by specialising some meta-classes from PNML Core.

MACHINE
$\quad PNMLCore$
SETS
$\quad ID\_AS;$
$\quad LABEL$
VARIABLES
$\quad ID,$
$\quad Label,$
$\quad Object,$
$\quad Node,$
$\quad Arc,$
$\quad PlaceNode,$
$\quad TransitionNode,$
$\quad Place,$
$\quad Transition,$
$\quad source,$
$\quad target$

INVARIANT
$\quad ID \in \mathcal{F}\ (ID\_AS) \wedge$
$\quad Label \in \mathcal{F}\ (LABEL) \wedge$
$\quad Object \subseteq ID \wedge$
$\quad Node \subseteq Object \wedge$
$\quad Arc \subseteq Object \wedge$
$\quad PlaceNode \subseteq Node \wedge$
$\quad TransitionNode \subseteq Node \wedge$
$\quad Place \subseteq PlaceNode \wedge$
$\quad Transition \subseteq TransitionNode \wedge$
$\quad source \in Arc \rightarrow Node \wedge$
$\quad target \in Arc \rightarrow Node \wedge$
$\quad Node \cap Arc = \emptyset \wedge$
$\quad PlaceNode \cap TransitionNode = \emptyset$

Fig. 26 Formal static semantics of PNML Core

Meeduse extracts from every meta-model a specific B machine defining its structural features. In this case study, we get two B machines and then we manually establish a refinement relation between them. Automatic DSL refinements is left to our future works, but we believe that it can be automated by exploiting the annotation mechanism of Meeduse. By this approach, it becomes possible to address the other Petri-net classes without any impact on the Core elements, because every Petri-net class would be defined as a specific refinement of PNML Core.

Figure 26 gives the variables and the invariants generated by Meeduse from PNML Core. In this machine, the inheritance is translated into set inclusion ($Place \subseteq PlaceNode \subseteq Node \subseteq Object \subseteq ID$). Variable *ID* refers to the objects identifiers. Regarding transitions and places, they are not directly related with input and output references such as in the simplified case study. In PNML Core, they are nodes, and their connections are defined by means of a class Arc.

The formal static semantics of the PT-nets meta-model is given in Figure 27. It is a data refinement of machine PNMLCore introducing several variables. In the meta-model, place markings and arc annotations are defined with the XML data-types NonNegativeInteger and PositiveInteger, as attributes of classes PTMarking and PTAnnotation. The translation into B applies types NAT (natural integers) and NAT_1 (strictly positive natural integers) to the corresponding variables *PTMarking_text* and *PTAnnotation_text*. The initial marking of places and the arc inscriptions are optional (multiplicity 0..1) and cannot be shared (because of the containment). They are then translated into partial injections.

The overall formal specification of the static semantics generated by Meeduse is about 457 lines of code. It provides 38 modelling operations from which the AtelierB prover produced 106 POs (95 were proved automatically and 11 using the interactive prover).

8.4 Execution semantics

In order to deal with the execution semantics of the PT-nets class of PNML, we introduce two transient (not serialised) attributes: attribute marking of class Place to represent the current number of tokens in a place and attribute value of class Arc to represent the number of tokens that are consumed

```
REFINEMENT
    $ptnets$
REFINES
    $PNMLCore$
VARIABLES
    $PTMarking,$
    $PTAnnotation,$
    $initialMarking,$
    $inscription,$
    $PTMarking\_text,$
    $PTAnnotation\_text$
INVARIANT
    $PTMarking \subseteq Label \wedge$
    $PTAnnotation \subseteq Label \wedge$
    $PTAnnotation \cap PTMarking = \emptyset \wedge$
    $initialMarking \in Place \rightarrowtail PTMarking \wedge$
    $inscription \in Arc \twoheadrightarrow PTAnnotation \wedge$
    $PTMarking\_text \in PTMarking \rightarrow \mathbf{NAT} \wedge$
    $PTAnnotation\_text \in PTAnnotation \rightarrow \mathbf{NAT}_1$
```

Fig. 27 Formal static semantics of PT-nets

```
MACHINE
    $semantics$
INCLUDES
    $ptnets$
VARIABLES
    $Place\_marking,\ Arc\_value$
INVARIANT
    $Place\_marking \in Place \rightarrow \mathbf{NAT} \wedge$
    $Arc\_value \in Arc \rightarrow \mathbf{NAT}_1$
INITIALISATION
$Place\_marking := (initialMarking\ ;\ PTMarking\_text)$
        $\cup\ (Place\ \text{-}\ \mathbf{dom}(initialMarking)) \times \{0\}$
$||\ Arc\_value := (inscription\ ;\ PTArcAnnotation\_text)$
        $\cup\ (Arc\ \text{-}\ \mathbf{dom}(inscription)) \times \{1\}$
```

Fig. 28 Structural part of the execution semantics

and/or produced when transitions are fired. These attributes are single-valued, mandatory and without an initial value. They are translated into total functions: *Place_marking* and *Arc_value*. Figure 28 shows the structural part of the execution semantics machine.

If a place is linked to a PTMarking via relation initialMarking then the corresponding value (represented with variable *PTMarking_text*) is assigned to its current marking (*initialMarking* ; *PTMarking_text*) in the initialisation, otherwise the attribute is set by default at 0 (($Place - \mathbf{dom}(initialMarking)) \times \{0\}$). The same principle is applied to attribute value of class Arc, but it takes value 1 if the arc does not have an inscription. In fact, when an inscription is not specified on the input arc (respectively, the output arc) of a transition, then by default only one token is consumed from its source place (respectively, one token is produced into its target place) when the transition is fired. The proof of correctness of this initialisation guarantees that none of the place markings and arc values are missed and that they are conformant to their typing domains (NAT and NAT_1).

```
fire =
    ANY $tt$ WHERE
        $tt \in Transition \wedge$
         $\mathbf{card}(inputs(tt)) = \mathbf{card}(target^{-1}[\{tt\}])$
    THEN
        $Place\_marking := Place\_marking \mathbin{<\!\!+} consume(tt)$ ;
        $Place\_marking := Place\_marking \mathbin{<\!\!+} produce(tt)$
    END
```

Fig. 29 Operation fire used in MeeNET

```
$inputs(tt) == \{pp,\ aa \mid pp \in Place \wedge aa \in Arc$
                     $\wedge\ source(aa) = pp \wedge target(aa) = tt$
                     $\wedge\ Arc\_value(aa) \leq Place\_marking(pp)$
              $\}$

$outputs(tt) == \{pp,\ aa \mid pp \in Place \wedge aa \in Arc$
                     $\wedge\ target(aa) = pp \wedge source(aa) = tt$
              $\}$

$consume(tt) == \{pp,\ nn \mid pp \in \mathbf{dom}(inputs(tt))$
    $\wedge\ nn = Place\_marking(pp) - Arc\_value(inputs(tt)(pp))\}$

$produce(tt) == \{pp,\ nn \mid pp \in \mathbf{dom}(outputs(tt))$
   $\wedge\ nn = Place\_marking(pp) + Arc\_value(outputs(tt)(pp))\}$
```

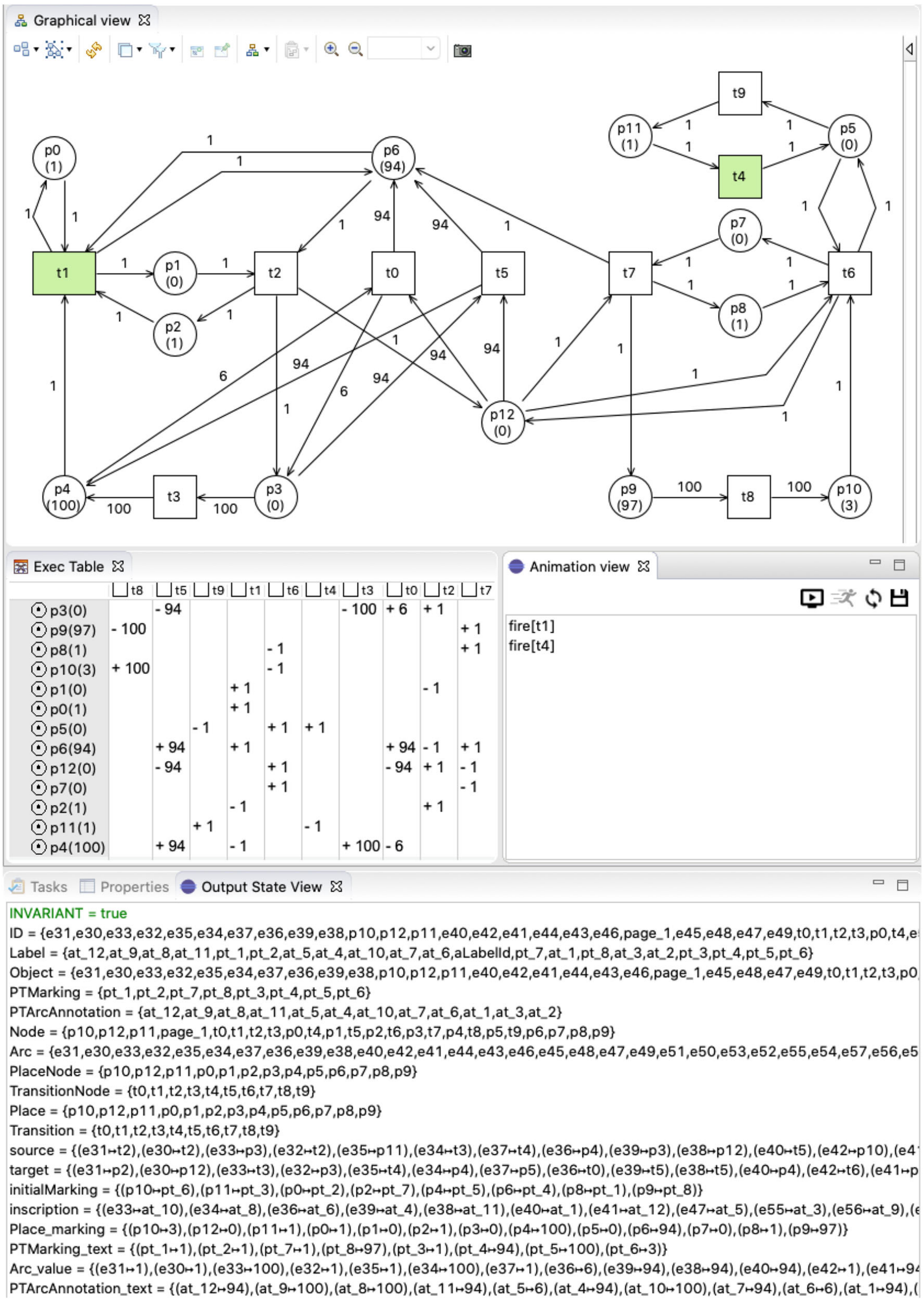
Fig. 30 Definitions

MeeNET integrates Meeduse and the formal static semantics of PT-nets within The ePNK, which allows one to experiment several formal specifications for the execution semantics together with realistic PNML models. For illustration, we propose a different technique than the application of CSP||B. The following formalisation (Fig. 29) is inspired by [52] were Event-B was used to provide a faithful formal semantics to basic and high-level Petri-nets. The notation $\mathbin{<\!\!+}$ denotes the overriding of a relation by another one. Operation fire selects non-deterministically a transition *tt* such that all its input places has a sufficient number of tokens; then it overrides relation *Place_marking* in order to update the markings of its input and output places. The various definitions used in this operation are given in Fig. 30.

Sets *inputs*(*tt*) and *outputs*(*tt*) get, respectively, input and output nodes and arcs of a given transition *tt*. Note that set *inputs*(*tt*) is restricted to places from which the consumption is possible ($Arc\_value(aa) \leq Place\_marking(pp)$). Consumption and production of tokens are also specified by means of set definitions. Set *consume*(*tt*) (respectively, *produce*(*tt*)) computes the new marking of the input (respectively, output) nodes of a transition *tt*.

8.5 MeeNET

Figure 31 is a screenshot of MeeNET while executing a PNML file of a satellite memory system. This file is taken from the benchmark of PNML files provided by the 10th

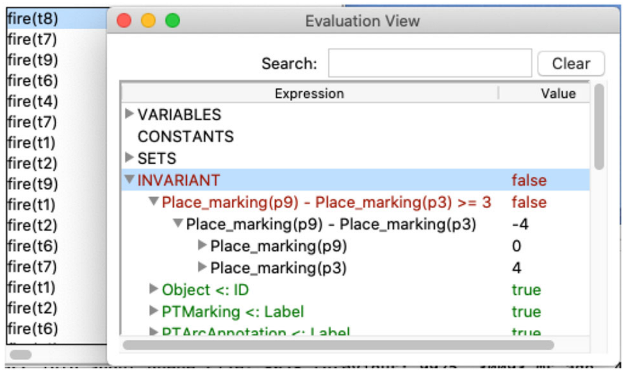


	t8	t5	t9	t1	t6	t4	t3	t0	t2	t7
p3(0)		- 94					- 100	+ 6	+ 1	
p9(97)	- 100									+ 1
p8(1)					- 1					+ 1
p10(3)	+ 100				- 1					
p1(0)				+ 1					- 1	
p0(1)				+ 1						
p5(0)			- 1		+ 1	+ 1				
p6(94)		+ 94		+ 1				+ 94	- 1	+ 1
p12(0)		- 94			+ 1			- 94	+ 1	- 1
p7(0)					+ 1					- 1
p2(1)				- 1					+ 1	
p11(1)			+ 1			- 1				
p4(100)		+ 94		- 1			+ 100	- 6		

```
fire[t1]
fire[t4]
```

```
INVARIANT = true
ID = {e31,e30,e33,e32,e35,e34,e37,e36,e39,e38,p10,p12,p11,e40,e42,e41,e44,e43,e46,page_1,e45,e48,e47,e49,t0,t1,t2,t3,p0,t4,e
Label = {at_12,at_9,at_8,at_11,pt_1,pt_2,at_5,at_4,at_10,at_7,at_6,aLabelId,pt_7,at_1,pt_8,at_3,at_2,pt_3,pt_4,pt_5,pt_6}
Object = {e31,e30,e33,e32,e35,e34,e37,e36,e39,e38,p10,p12,p11,e40,e42,e41,e44,e43,e46,page_1,e45,e48,e47,e49,t0,t1,t2,t3,p0
PTMarking = {pt_1,pt_2,pt_7,pt_8,pt_3,pt_4,pt_5,pt_6}
PTArcAnnotation = {at_12,at_9,at_8,at_11,at_5,at_4,at_10,at_7,at_6,at_1,at_3,at_2}
Node = {p10,p12,p11,page_1,t0,t1,t2,t3,p0,t4,p1,t5,p2,t6,p3,t7,p4,t8,p5,t9,p6,p7,p8,p9}
Arc = {e31,e30,e33,e32,e35,e34,e37,e36,e39,e38,e40,e42,e41,e44,e43,e46,e45,e48,e47,e49,e51,e50,e53,e52,e55,e54,e57,e56,e5
PlaceNode = {p10,p12,p11,p0,p1,p2,p3,p4,p5,p6,p7,p8,p9}
TransitionNode = {t0,t1,t2,t3,t4,t5,t6,t7,t8,t9}
Place = {p10,p12,p11,p0,p1,p2,p3,p4,p5,p6,p7,p8,p9}
Transition = {t0,t1,t2,t3,t4,t5,t6,t7,t8,t9}
source = {(e31↦t2),(e30↦t2),(e33↦p3),(e32↦t2),(e35↦p11),(e34↦t3),(e37↦t4),(e36↦p4),(e39↦p3),(e38↦p12),(e40↦t5),(e42↦p10),(e4
target = {(e31↦p2),(e30↦p12),(e33↦t3),(e32↦p3),(e35↦t4),(e34↦p4),(e37↦p5),(e36↦t0),(e39↦t5),(e38↦t5),(e40↦p4),(e42↦t6),(e41↦p
initialMarking = {(p10↦pt_6),(p11↦pt_3),(p0↦pt_2),(p2↦pt_7),(p4↦pt_5),(p6↦pt_4),(p8↦pt_1),(p9↦pt_8)}
inscription = {(e33↦at_10),(e34↦at_8),(e36↦at_6),(e39↦at_4),(e38↦at_11),(e40↦at_1),(e41↦at_12),(e47↦at_5),(e55↦at_3),(e56↦at_9),(
Place_marking = {(p10↦3),(p12↦0),(p11↦1),(p0↦1),(p1↦0),(p2↦1),(p3↦0),(p4↦100),(p5↦0),(p6↦94),(p7↦0),(p8↦1),(p9↦97)}
PTMarking_text = {(pt_1↦1),(pt_2↦1),(pt_7↦1),(pt_8↦97),(pt_3↦1),(pt_4↦94),(pt_5↦100),(pt_6↦3)}
Arc_value = {(e31↦1),(e30↦1),(e33↦100),(e32↦1),(e35↦1),(e34↦100),(e37↦1),(e36↦6),(e39↦94),(e38↦94),(e40↦94),(e42↦1),(e41↦94
PTArcAnnotation_text = {(at_12↦94),(at_9↦100),(at_8↦100),(at_11↦94),(at_5↦6),(at_4↦94),(at_10↦100),(at_7↦94),(at_6↦6),(at_1↦94),(
```

Fig. 31 A satellite memory PNML model



```
fire(t8)
fire(t7)
fire(t9)
fire(t6)
fire(t4)
fire(t7)
fire(t1)
fire(t2)
fire(t9)
fire(t1)
fire(t2)
fire(t6)
fire(t7)
fire(t1)
fire(t2)
fire(t6)
```

```
Evaluation View
Search:                                   Clear
Expression                                              Value
▶ VARIABLES
  CONSTANTS
▶ SETS
▼ INVARIANT                                             false
    ▼ Place_marking(p9) - Place_marking(p3) >= 3        false
        ▼ Place_marking(p9) - Place_marking(p3)         -4
            ▶ Place_marking(p9)                         0
            ▶ Place_marking(p3)                         4
    ▶ Object <: ID                                      true
    ▶ PTMarking <: Label                                true
    ▶ PTArcAnnotation <: Label                          true
```

Fig. 32 Invariant violation detected by ProB

edition of the model-checking contest (MCC'2020)[8]. We designed the graphical and the tabular views in Sirius in order to visualise graphically the model. For example, the green transitions (t1 and t4) are transitions that can be fired in the current execution step. They satisfy the guard of operation fire. The animation view and the state view represent the output of ProB. The animation view is for interactive debugging, it allows the user to select which transition to fire; and State view gives the current valuations of the B variables of the static semantics machine.

The Petri-net of Figure 31 models the behaviour of the mass memory management system in a micro-satellite from the Myriade product line, designed by CNES, the French Space Agency. The corresponding controller must ensure that there is always a "security buffer" between the sectors pointed by places p9 and p3. The size of this buffer is a parameter Y (whose value in this model is equal to 3). The system should guarantee the following invariant:

$$p3 > p9 \Rightarrow p3 - p9 \geq Y$$

Based on the valued B specification produced by MeeNET, the ProB model-checker explored all the state-space and showed that this property is preserved all along the execution of the Petri-net model. We also checked other properties, such as for example to look at situations where $p9$ exceeds $p3$: $p9 - p3 \geq Y$. Figure 32 shows the corresponding counter-example and the invariant debugger of ProB. Other properties, such as LTL properties, are provided with the model and ProB was also able to check them.

We have tried several PNML files from the MCC'2020 in MeeNET (even the biggest ones) and it was able to inject their valuations within a valued B specification that can be further used in ProB for verification purposes. Nevertheless, the objective of this paper is not to evaluate the verification capabilities of ProB or its performance based on the resulting valued models. The contribution of our work is that it made possible the use of ProB in the Petri-nets field. Further works and experiments are required if one would like to apply ProB to compete with established Petri-net-based model-checkers.

9 Discussion and conclusion

Few proposals exist for bridging the gap between DSLs with their supporting tools and a proof based formal approach dedicated to execution semantics. The closest ones in comparison with the Meeduse approach in general are [53] and [54], where, respectively, MAUD and Abstract State Machines (ASM) are used. Unfortunately, these works do not address the joint execution of the formal model and the DSL, which is a major contribution of Meeduse. One interesting perspective is to integrate several target formalisms within Meeduse and take benefit of these works. This paper is a case study application rather than a research paper. It applies the strengths of Meeduse to the Petri-net field and proposes a formal support to PNML, the international standard ISO/IEC 15909 for Petri-nets. Before coming to this real-life application, we tried several simplified implementations of this DSL applying MDE tools for DSL execution. The study allowed us to illustrate via several obvious details, how useful is a formal approach especially when addressing a DSL that is dedicated to safety-critical systems.

The major observation is that during the verification activities several concerns must be dissociated: properties of the meta-model, those of the execution semantics and finally those of the coordination algorithm. We believe that existing MDE approaches for DSL development and execution such as those discussed in this paper suffer from the lack of formal reasoning. This is also confirmed by the studies of [6] and [5] showing that few tools provide support for verification. However, when an executable DSL is not carefully checked, it may lead to a succession of conceptual failures: failures of modelling operations (*e.g.* setTokens) may result in failures of execution operations (*e.g.* addToken), which in turn may result in failures of the coordination operations (*e.g.* fire/run). The major risk is that when debugging a model, the domain expert does not worry about the correctness of the DSL tool because it is basically the task of the MDE expert. In language programming the developer debugs his/her source code (in MDE this is seen as a model) using existing IDEs without paying attention to how the language semantics are encoded since in the language theory these semantics are formally defined and compilers are well-established techniques. Observations made by this work show that it is difficult to transpose debugging as used in language programming and compilers into executable DSLs especially for safety-critical systems.

We recall that the four implementations of our benchmark tools are highly simplified and used for illustration

[8] The benchmark can be found at: https://mcc.lip6.fr/models.php.

only. The underlying approaches are accepted by the MDE community and have the ability to design and correctly instrument DSLs. From our experiments, $\text{PNet}_{\text{Java}}$ gave the better execution outputs; however, it suffers from its poor abstraction. 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. Other existing languages, such as MoCC within the Gemoc Studio tool for example, offer concurrency and communication models representing the control flow aspects, including the synchronisations and the causality relationships between the execution functions [13]. This is close to some CSP operators, but the missing piece is the use of a prover and a model-checker for automatic validation activities. Integration of MoCC [13] and CSP||B [46] is an interesting perspective of our works.

We think that safety-critical systems require the collaboration between a MDE expert and a formal method expert. The first actor defines the DSL with associated modelling tools and the second one defines the execution semantics with associated verification activities. An alliance of MDE and the B method is investigated in the current paper. The resulting tool and approach for Petri-nets contributes towards this field by:

– The possibility to benefit from the rich catalogue of MDE tools without losing sight of correctness and rigorous development. Indeed, several MDE tools exist: model-to-model transformation, model-to-code generation, constraint-checkers, graphical concrete syntax representation, bi-directional mappings, etc.
– The possibility to use ProB as a model-checker of Petri-net models. We presented an illustration in section 8 based on a satellite memory system. ProB has numerous capabilities: trace management, random animation, heuristic model checking, etc. Furthermore, the use of Petri-nets and B in the same development benefits from their complementarities: Petri-nets may be used as a graphical front-end of a B development project and the B framework complements with formal analysis of the system modelled using Petri nets.

Several perspectives arise from this work. The major one is to continue the development of MeeNET in order to cover the other Petri-nets classes of PNML. We showed that B refinements are a good solution to master the complexity of having several kinds of semantics that extend the core model. In the same direction, currently we used Sirius to provide graphical and tabular views of the DSL executions. We believe that more specific views are required to ease the debugging of a Petri-net model, especially ProB has several outputs (such as invariant evaluation) that can be used in the Sirius views for a better representation of the DSL characteristics while executing it.

References

1. Bandener N, Soltenborn C and Engels G (2011) Extending DMM Behavior Specifications for Visual Execution and Debugging. Software Language Engineering, volume 6563 of *LNCS*, pages 357–376. Springer
2. Engels G, Hausmann JH, Heckel R and Sauer S (2000) Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In Andy Evans, Stuart Kent, and Bran Selic, editors, UML 2000 — The Unified Modeling Language, volume 1939 of *LNCS*, pages 323–337. Springer
3. Tatibouet J, Cuccuru A, Gerard S, & Terrier F (2014) Formalizing execution semantics of uml profiles with fuml models. In *Model-Driven Engineering Languages and Systems (Models)*, volume 8767 of LNCS, pages 133–148. Springer
4. Gemoc. Gemoc. http://gemoc.org/
5. Kosar T, Bohra S, Mernik M (2016) Domain-specific languages: a systematic mapping study. Inf Softw Technol 71:77–91
6. Lung A, Carbonell J, Marchezan L, Rodrigues E, Bernardino M, Basso FP, Medeiros B (2020) Systematic mapping study on domain-specific language development tools. Empir Softw Eng 25(5):4205–4249
7. Meeduse http://vasco.imag.fr/tools/meeduse/. Accessed: 15-12-2020
8. Idani A (2020) Meeduse: A tool to build and run proved dsls. In Brijesh, D. and Elena, T., editors, *16th International Conference on Integrated Formal Methods (IFM)*, volume 12546 of LNCS, pages 349–367. Springer
9. Idani A, Ledru Y, Vega G (2020) Alliance of model driven engineering with a proof-based formal approach. Int J Innov Syst Softw Eng (ISSE) 16(3):289–307
10. Abrial JR (1996) The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA
11. Leuschel M, Butler M (2008) Prob: an automated analysis toolset for the b method. Int J Softw Tools Technol Transf 10(2):185–203
12. Idani A (2020) Dependability of model-driven executable dsls - critical review and solutions. In: Henry M, Paris A, Barbora B, Javier C, Mauro C, Mirco F, Anne K, Patrizia S, Catia T, Danny W, and Uwe Z (Eds.) 3rd International Workshop on Modeling, Verification and Testing of Dependable Critical Systems (DETECT), volume 1269 of CCIS, pages 358–373. Springer
13. Deantoni J (2016) Modeling the behavioral semantics of heterogeneous languages and their coordination. In 2016 Architecture-Centric Virtual Integration (ACVI), pages 12–18
14. Bousse E, Leroy D, Combemale B, Wimmer M, Baudry B (2018) Omniscient debugging for executable dsls. J Syst Softw 137:261–288
15. Langer P, Mayerhofer T and Kappel G (2014) Semantic model differencing utilizing behavioral semantics specifications. In 17th International Conference Model-Driven Engineering Languages and Systems - MODELS, volume 8767 of LNCS, pages 116–132. Springer
16. Mayerhofer T, Langer P, Wimmer M, & Kappel G (2013) Towards xmof: Executable dsmls based on fuml. In International Conference on Software Language Engineering - SLE, volume 8225 of LNCS, pages 56–75. Springer
17. Hoare CAR (1985) Communicating Sequential Processes. Prentice-Hall Inc, Upper Saddle River, NJ, USA
18. Petri CA, Reisig W (2008) Petri net. Scholarpedia 3(4):6477
19. Petri net ecore file. https://github.com/gemoc/petrinet/blob/master/petrinetv1/fr.inria.diverse.sample.petrinetv1.model/model/petrinetv1.ecore. Accessed: 15-12-2020
20. Wachsmuth G (2008) Modelling the operational semantics of domain-specific modelling languages. In: Lämmel R, Visser J, Saraiva J (eds) Generative and Transformational Techniques in

Software Engineering II (GTTSE). Springer, Berlin Heidelberg, pp 506–520
21. Hartmann T and Sadilek DA (2008) Undoing operational steps of domain-specific modeling languages. In Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling (DSM'08) - University of Alabama at Birmingham
22. EProvide. http://eprovide.sourceforge.net. Accessed: 15-12-2020
23. Jezequel JM, Combemale B, Barais O, Monperrus M, Fouquet F (2013) Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. Softw Syst Model 14(2):905–920
24. Object Management Group. Meta Object Facility (MOF) 2.5.1 Core Specification. https://www.omg.org/spec/MOF/2.5.1/, 2015
25. XTend example of Petri-nets. https://github.com/gemoc/petrinet/blob/master/petrinetv1/. Accessed: 15-12-2020
26. Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (fUML)*. https://www.omg.org/spec/FUML/, 2011
27. fUML source code. http://www.modelexecution.org/moliz/xmof/. Accessed: 15-12-2020
28. Thong WJ and Ameedeen MA (2015) A survey of petri net tools. In Advanced Computer and Communication Engineering Technology, pages 537–551, Cham. Springer
29. Lienhard A, Girba T, & Nierstrasz O (2008) Practical object-oriented back-in-time debugging. In Jan Vitek, editor, ECOOP 2008 – Object-Oriented Programming, pages 592–615. Springer
30. Baar T (2005) Non-deterministic constructs in OCL - what does any() mean. In Model Driven - 12th International SDL Forum, volume 3530 of LNCS, pages 32–46. Springer
31. Vallecillo A, & Gogolla M (2017) Adding random operations to OCL. In Proceedings of MODELS 2017 Satellite Event, CEUR Workshop Proceedings, pages 324–328. CEUR-WS.org
32. Andova S, van den Brand MG, Engelen LJ and Verhoeff T (2012) MDE basics with a DSL focus. In Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, volume 7320 of LNCS, pages 21–57. Springer
33. Leroy X (2009) Formal verification of a realistic compiler. Commun ACM 7:107–115
34. Leonid A (1994) Levin. Birkhauser Verlag, Randomness and non-determinism, In International Congress of Mathematicians
35. Object Management Group (2014) Object Constraint Language (OCL) 2.4 Core Specification. https://www.omg.org/spec/OCL/
36. Woodcock J, Larsen PG, Bicarregui J, Fitzgerald J (2009) Formal methods: Practice and experience. ACM Comput Surveys (CSUR) 41(4):1–36
37. Hoare T (2007) The ideal of program correctness. Comput J 50(3):254–260
38. Object Management Group (2017) Unified Modeling Language (UML) 2.5.1 Core Specification. https://www.omg.org/spec/UML/
39. Gosling J, Joy B, Steele G, Bracha G, Buckley A and Smith D (2018) The Java Language Specification, Java SE 10 Edition. https://docs.oracle.com/javase/specs/
40. Behm P, Benoit P, Faivre A and Meynadier JM (1999) Météor: A successful application of b in a large project. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems*, FM '99, pages 369–387, London, UK, UK. Springer-Verlag
41. Bobbio A (1990) System modelling with petri nets. In: Colombo AG, de Bustamante AS (eds) Systems Reliability Assessment. Springer, Netherlands, Dordrecht, pp 103–143
42. Cortadella J and Reisig W (2004) editors. Applications and Theory of Petri Nets 2004, 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings, volume 3099 of *LNCS*. Springer
43. Lann Le G (1996) The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. Research Report RR-3079, INRIA. Projet REFLECS
44. Atelier B (2020) http://www.atelierb.eu/en/. Accessed: 15-12
45. Oxford. *The Oxford Dictionary*. Oxford University Press
46. Butler M and Leuschel M (2005) Combining CSP and B for specification and property verification. In International Symposium of Formal Methods - FM 2005, volume 3582 of Lecture Notes in Computer Science, pages 221–236. Springer
47. Hillah LM, Kindler E, Kordon F, Petrucci L, Treves N (2009) A primer on the Petri Net Markup Language and ISO/IEC 15909–2. Petri Net Newslett 76:9–28
48. PNML Homepage (2020) http://www.pnml.org. Accessed: 15-12
49. The ePNK Homepage (2020) http://www2.compute.dtu.dk/~ekki/projects/ePNK/index.shtml. Accessed: 15-12
50. PNML Framweork Homepage. https://pnml.lip6.fr. Accessed: 15-12-2020
51. ISO/IEC Systems and software engineering "High-level Petri nets" Part 2: Transfer format, International Standard ISO/IEC 15909-2
52. Attiogbe C (2009) Semantic Embedding of Petri Nets into Event-B. In *Integration of Model-based Formal Methods Tools (IM_FMT @ IFM'2009)*, Dusseldorf, Germany, March . http://www.lina.sciences.univ-nantes.fr/apcb/IM_FMT2009/index.html
53. Rivera JE, Duran F, Vallecillo A (2009) Formal specification and analysis of domain specific models using maude. Simulation 85(778–792):10
54. Gargantini A, Riccobene E, Scandurra P (2010) Combining formal methods and mde techniques for model-driven system design and analysis. Advances in Software 3(1 & 2)