Meeduse: A Tool to Build and Run Proved DSLs

Akram Idani(✉)

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

Abstract. Executable Domain-Specific Languages (DSLs) are a promising paradigm in software systems development because they are aiming at performing early analysis of a system's behavior. They can be simulated and debugged by existing Model-Driven Engineering (MDE) tools leading to a better understanding of the system before its implementation. However, as the quality of the resulting system is closely related to the quality of the DSL, there is a need to ensure the correctness of the DSL and apply execution engines with a high level of trust. To this aim we developed Meeduse, a tool in which the MDE paradigm is mixed with a formal method assisted by automated reasoning tools such as provers and model-checkers. Meeduse assists the formal definition of the DSL static semantics by translating its meta-model into an equivalent formal B specification. The dynamic semantics can be defined using proved B operations that guarantee the correctness of the DSL's behavior with respect to its safety invariant properties. Regarding execution, Meeduse applies the ProB animator in order to animate underlying domain-specific scenarios.

Keywords: B Method · Domain-specific languages · MDE

1 Introduction

Model Driven Engineering (MDE) tools allow a rapid prototyping of domain-specific Languages (DSLs) with automated editor generation, integrated type-checking and contextual constraints verification, etc. This technique is powerful and provides a framework to implement the dynamic semantics of the language or to build compilers that translate the input formalism into another one (*e.g.* bytecode, programming language or another DSL). However, the major drawback of this approach is that the underlying verification and validation activities are limited to testing, which makes difficult the development of bug-free language analysers and compilers. When these tools are used for safety-critical or high-assurance software, [20] 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*". Formal methods demonstrated their capability to guarantee the safety properties of languages

and associated tools [7]; nonetheless, there is a lack of available tools to bridge the gap between MDE and formal methods for the development of DSLs.

In our works we apply the B method [1] to formally define the semantics that make a DSL executable and hence guarantee the correctness of its behaviour. The challenge of executing a DSL is not new and was widely addressed in the literature at several abstraction levels with various languages [5,12,23,29]. An executable DSL would not only represent the expected system's structure but it must itself behave as the system should run. The description of this behaviour also applies a language with its own abstract syntax and semantic domain. Unfortunately, most of the well-known existing DSL development approaches apply languages and tools that are not currently assisted by formal proofs.

In [16] we showed that there is an equivalence between the static semantics of DSLs and several constructs of the B method. In this paper we present Meeduse, a MDE platform built on our previous works [13,16] and whose intention is to circumvent the aforementioned shortcomings of MDE tools. It allows to formally check the semantics of DSLs by applying tools of the B method: AtelierB [6] for theorem proving and ProB [21] for animation and model-checking. The Meeduse approach translates the meta-model of a given DSL, designed in the Eclipse Modeling Framework (EMF [27]), into an equivalent formal B specification and then injects a DSL instance into this specification. The strength of Meeduse is that it synchronises the resulting B specification with the DSL instance and hence the animation of the B specification automatically leads to a visual execution of the DSL. This approach was successfully applied on a realistic railway case study [14,15] and also to formalize and execute a real-life DSL transformation [17] which is that of transforming truth tables into binary decision diagrams. This paper shows how the B method can be integrated within MDE and presents by practice Meeduse.

Section 2 presents a simple textual DSL built in a MDE tool and discusses its underlying semantics. Section 3 gives an overall view about the Meeduse approach and architecture and shows how the B method is integrated within a model-driven architecture. Section 4 applies two approaches to define the DSL semantics: the meta-model based approach and the CP-net approach. Section 5 summarizes two realistic applications of Meeduse and discusses their results. Finally, Section 6 draws the conclusions and the perspectives of this work.

2   A Simple DSL

For illustration we apply a well-known DSL builder (Xtext [2]) that allows to define textual languages using LL(*) grammars and generate a full infrastructure, including an ANTLR parser API with a type-checker and auto-completion facilities. We define a simple DSL that represents configuration files edited by operating system administrators to configure GPU servers. These servers are packed with graphics cards, called Graphics Processing Units (GPUs) that are used for high performance computing. Roughly speaking, in a GPU architecture, significant jobs are broken down into smaller computations (called here processes) that can be executed in parallel by the different GPUs.

Figure 1 gives the Xtext grammar of our DSL; it defines three non-terminal rules: Server (the axiom), Gpu and Process. Every rule starts by the definition of a naming identifier (name=ID) representing object declaration. The declaration of a server object is followed by two declaration lists: that of GPUs (gpus+=Gpu)* and that of processes (processes+=Process)*. A GPU has a fixed number of slots: free slots are defined by an Integer value (size=INT) and the occupied slots directly refer to their occupying processes (usedBy+=[Process]*).

```
Server: name=ID (gpus+=Gpu)* (pocesses+=Process)*;
Gpu:    name=ID ':' size=INT '(slots)' '<-' (usedBy+=[Process])* ';';
Process: name=ID ;
```

Fig. 1. Example of an Xtext grammar.

Given a grammar, Xtext applies ANTLR to generate a Java API for a parser that can build and walk the abstract syntax tree (AST). One interesting feature of Xtext is that it defines the language AST by means of an EMF meta-model [27], which makes possible the integration of MDE tools that are built on top of EMF like OCL constraints checker, etc. Figure 2 provides the EMF meta-model of our DSL. It is composed of three classes, each of which is issued from a grammar rule. The grammar axiom is the root class of this meta-model and the associations represent the various object relationships.
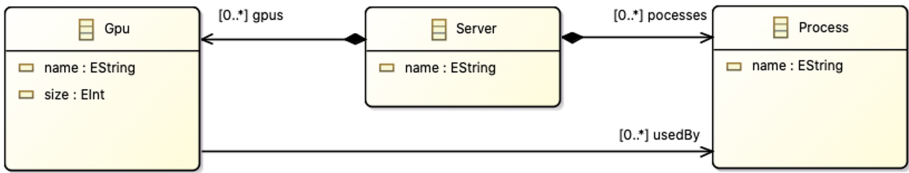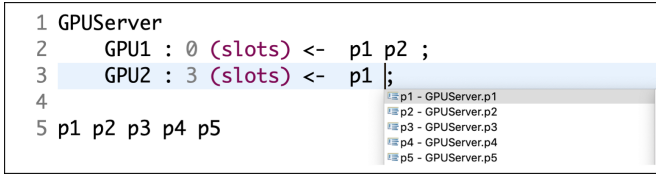

Fig. 2. The GPU server meta-model

Figure 3 presents the textual editor produced by Xtext for our DSL. In this file, the system administrator defined a server (GPUServer) with two GPUs (GPU1 and GPU2), and five processes (from p1 to p5). Process p2 is assigned to GPU1 and process p1 is assigned to both GPU1 and GPU2.

The DSL's grammar and the corresponding meta-model define the static semantics. Regarding the dynamic semantics, we informally define them with the following process scheduling actions:

– **enqueue/purge**: respectively assign and de-assign processes to a GPU server. Technically action **enqueue** declares a process in the DSL file, and action **purge** removes a process from this file. When assigned to the server the initial state of the process is Waiting.

```
1 GPUServer
2     GPU1 : 0 (slots) <-  p1 p2 ;
3     GPU2 : 3 (slots) <-  p1 |;
                            p1 - GPUServer.p1
4                           p2 - GPUServer.p2
                            p3 - GPUServer.p3
5 p1 p2 p3 p4 p5            p4 - GPUServer.p4
                            p5 - GPUServer.p5
```

Fig. 3. Generated Xtext textual editor

- **ready**: assigns a GPU with at least one free slot to a waiting process. The process becomes then Active. If all GPUs are busy, the process enters in an intermediary state called Ready.
- **swap**: releases a slot by deactivating the corresponding process (the process becomes Waiting) and allocates the freed GPU slot to some ready process (the latter becomes then Active).

These actions must guarantee the following safety properties:

- The number of free slots cannot be negative;
- A process cannot be running on more than one GPU;
- If there is a Ready process then all GPUs are busy;
- A process cannot be Active and Ready or Active and Waiting or Ready and Waiting at the same time;
- An Active process is assigned to a GPU;
- Waiting and Ready processes are not assigned to GPUs.

3   The Meeduse Approach

In MDE, the implementation of DSLs is derived from their meta-models and as the semantics of meta-models is standardized [24] (by the Object Management Group − OMG), the underlying DSL implementation and associated tool-set code generation follow well established rules. This makes the integration of MDE tools easy and transparent. In fact, there are numerous MDE tools with various purposes: model-to-model transformation, model-to-code generation, constraint-checkers, graphical concrete syntax representation, bi-directional DSL mappings, etc. All these tools have the ability to work together using shared DSLs, as far as the semantics of these DSLs are defined by means of meta-models. The overall principle of a model-driven architecture is that once a meta-model is instantiated, MDE tools can be synchronised using the resulting model resource.

3.1   Main Approach

Figure 4 shows how Meeduse integrates the formal B method within MDE tools in order to build proved DSLs and execute their dynamic semantics. The left hand side of the figure represents a model-driven architecture where a meta-model (*e.g.* Fig. 2) is extracted by Xtext from a DSL grammar (*e.g.* Fig. 1).

When an input textual file is parsed, a model resource is created as an instance
of the meta-model. Then every modification of the model resource implies a
modification of the textual file, and vice-versa. Thanks to the standard seman-
tics of meta-models, Meeduse can be synchronised with any model resource,
which opens a bridge between MDE tools (Xtext or others) and formal tools like
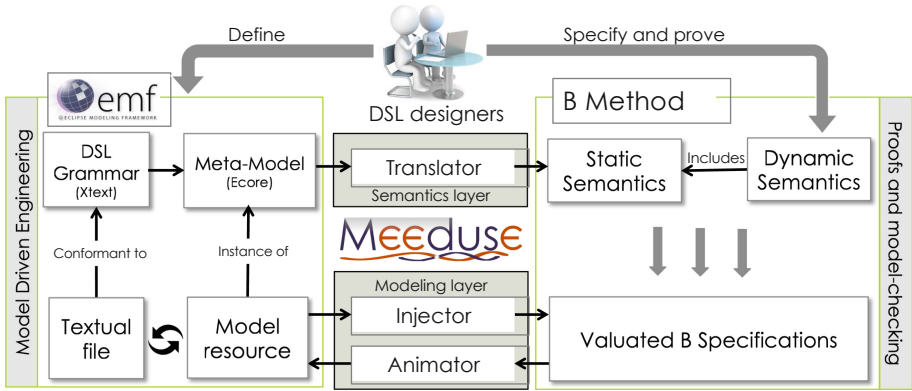animators and model-checkers.

Fig. 4. The Meeduse approach.

3.2   Semantics Layer

The Meeduse approach starts by translating the meta-model of a DSL into the
B language. This translation is done by component "Translator" of Fig. 4. The
resulting formal model represents the static semantics of the DSL. It defines the
structural features of the meta-model using B data structures: sets, variables
and typing invariants. This translation applies a classical UML-to-B transfor-
mation technique, because all constructs of a meta-model have an equivalent
in UML. For this purpose component "Translator" embeds B4MSecure [13], an
open-source MDE platform whose advantage, in comparison with other UML-
to-B tools, [8,26] is that it offers an extensibility facility allowing to easily add
new UML-to-B rules or to specialize existing rules depending on the application
context. In Meeduse the application context of UML-to-B rules is that of EMF
meta-models.

A meta-class Class is translated into an abstract set named CLASS repre-
senting possible instances and a variable named Class representing the set of
existing instances. Basic types (*e.g.* integer, boolean, etc.) become B types (Z,
Bool, etc.), and attributes and references lead to functional relations depending
on their multiplicities. Additional structural invariant properties can be written
in B based on the generated B data. Figure 5 gives clauses SETS, VARIABLES
and INVARIANT generated by Meeduse from the meta-model of Fig. 2.

MACHINE *GPUModel*	

SETS
     SERVER;
     GPU;
     PROCESS
 VARIABLES
     Server,
     Gpu,
     Process,
     running,
     processes,
     gpus,
     Gpu_size
	INVARIANT
	     $Server \subseteq SERVER$
	     $\wedge\ Gpu \subseteq GPU$
	     $\wedge\ Process \subseteq PROCESS$
	     $\wedge\ running \in Process \nrightarrow Gpu$
	     $\wedge\ processes \in Process \nrightarrow Server$
	     $\wedge\ gpus \in Gpu \nrightarrow Server$
	     $\wedge\ Gpu\_size \in Gpu \rightarrow \mathbf{NAT}$

Fig. 5. Formal DSL static semantics.

During the extraction of the B specifications, the user can strengthen some properties of the meta-model. For example, attribute size is defined as an integer in the meta-model, but we translate its type into type NAT in order to limit its values to positive numbers as stated in the safety properties. The user can also complete the multiplicities over one-direction associations and apply to them specific names. For example, from our grammar Xtext generated a one-direction association from class Gpu to class Process with role usedBy. This means that the parser doesn't look at all to the opposite side of the association and hence it doesn't check the number of GPUs on which a process is running. During the translation of this association into B we assigned multiplicity 0..1 to its opposite side and we gave to it name *running*. This choice led to the partial relation named *running* from set Process to set Gpu.

Operations of this B specification, that may be generated automatically or even introduced manually, must preserve the structural invariant of Fig. 5. For this simple example, the generated invariant addresses four main static properties: (*i*) a process cannot be running on more than one GPU, (*ii*) the number of free slots is greater or equal to 0, (*iii*) processes are assigned to only one server at the same time, and (*iv*) GPUs cannot be shared by several servers. The proof of correctness guarantees that every provided operation never produces a wrong model − regarding this invariant − such as that of Fig. 3 where property (*iii*) is violated. Indeed, a proof-based formal approach is expected to provide error-free domain-specific operations.

3.3   Modeling Layer

The modeling layer is ensured by components "Injector" and "Animator". The "Injector" injects a model resource, issued from any EMF-based modeling tool (Xtext, Sirius, GMF, etc.) into the B specification produced from the meta-model. This component introduces enumerations into abstract data structures

like abstract sets, and produces valuations of the B machine variables. Figure 6 presents clauses SETS and INITIALISATION generated by component "Injector" from an input model resource where five processes are scheduled and such that `GPU1` is running `p2` and `GPU2` is running `p4` and `p5`. In this model, `GPU2` is busy and `GPU1` has one remaining free slot.

```
server1.gpu ⊠
  GPUServer
      GPU1 : 1 (slots) <-  p2 ;
      GPU2 : 0 (slots) <-  p4 p5 ;

  p1 p2 p3 p4 p5
```

↓ Extraction of B data values ↓

SETS
  $SERVER=\{GPUServer\}$;
  $GPU=\{GPU1, GPU2\}$;
  $PROCESS=\{p1, p2, p3, p4, p5\}$

INITIALISATION
  $Server := \{GPUServer\}$
  $\|\ Gpu := \{GPU1, GPU2\}$
  $\|\ Process := \{p1, p2, p3, p4, p5\}$
  $\|\ running := \{(p2 \mapsto GPU1),(p4 \mapsto GPU2), (p5 \mapsto GPU2)\}$
  $\|\ processes := \{(p1 \mapsto GPUServer), (p2 \mapsto GPUServer),$
  $\qquad\qquad\qquad (p3 \mapsto GPUServer), (p4 \mapsto GPUServer),$
  $\qquad\qquad\qquad (p5 \mapsto GPUServer)\}$
  $\|\ gpus := \{(GPU1 \mapsto GPUServer), (GPU2 \mapsto GPUServer)\}$
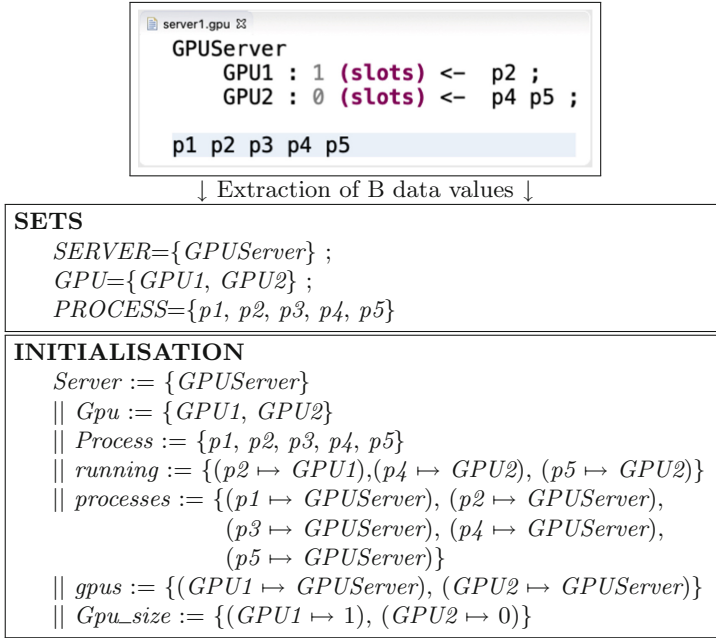 $\|\ Gpu\_size := \{(GPU1 \mapsto 1), (GPU2 \mapsto 0)\}$
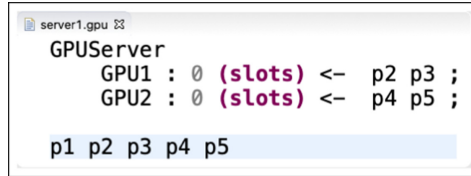
Fig. 6. Valuation of the B machine.

In our approach the execution environment of the DSL is composed of Meeduse coupled with ProB, and the domain-specific actions (*e.g.* `enqueue/purge`, `ready` and `swap`) are defined as B operations that can be animated by the domain expert. At the beginning of the animation, the injector produces a B specification whose initial state is equivalent to the input model resource. If the input model is wrong (such as that of Fig. 3) ProB would detect it and the animation is stopped. In fact, our objective is to safely execute the DSL. Given a correct input model, component "Animator" keeps the equivalence between the state of the B specifications and the input model resource all along the animation process. When a new state is reached, Meeduse translates it back to the model resource and all MDE tools synchronised with this resource are automatically updated.

The "Animator" applies a constraint solving approach to compute for every variable the difference between its value before (v) and its value after (v') the animation of a B operation. Then, it applies the equivalent transformation to

the corresponding element in the model resource. Formula $v - v'$ computes the values that are removed by the operation, and $v' - v$ computes the added ones. Suppose, for example, that from the initial state of Fig. 6 the animation of a given operation produces the following computation results for variables *running* and *Gpu_size*:

1. $running - running' = \emptyset$
2. $running' - running = \{(p3 \mapsto GPU1)\}$
3. $Gpu\_size - Gpu\_size' = \{(GPU1 \mapsto 1)\}$
4. $Gpu\_size' - Gpu\_size = \{(GPU1 \mapsto 0)\}$

Having these results, Meeduse transforms the model resource as follows: (1) and (2) create a link `running` between objects `p3` and `GPU1`; (3) and (4) modify the value of attribute `size` of object `GPU1` from 1 to 0. Figure 7 shows how the input textual file is updated after these modifications: process `p3` is now running on `GPU1` and all GPUs became busy.

```
server1.gpu ⊠
  GPUServer
      GPU1 : 0 (slots) <-  p2 p3 ;
      GPU2 : 0 (slots) <-  p4 p5 ;

  p1 p2 p3 p4 p5
```

Fig. 7. Example of an output model.

The reverse translation from a given state of the B machine into the EMF model resource is limited by the constructs of meta-models that Meeduse is able to translate into B. If the user adds programmatically some concepts to the DSL implementation that are not introduced within the EMF meta-model, then these concepts are missed during the animation. This may happen, for example, when the DSL encompasses stateful computations that are hand written by the developer using the Java implementation generated by Xtext. Despite that Meeduse does not provide a checking facility to ensure that a given model can be animated, it guarantees that all concepts of the meta-model that are translated into B are covered during the animation.

3.4   Meeduse Contributions

Figure 8 is a screenshot of Meeduse where the left hand side presents the textual DSL editor, and the right hand side shows: (1) the list of B operations that can be enabled, and (2) the current B variable valuations. The B specification used in this illustration applies B operations that define the domain-specific actions based on the B data structure extracted from the meta-model. Operation `ready` can be applied to `p1` or `p3` because both are waiting. These processes can also

be purged using operation `purge`. Regarding the active processes (`p2`, `p4` and `p5`) they can be deactivated by operation `swap`. Playing with these operations in Meeduse automatically modifies the model resource and then the textual file is automatically updated. For example, Fig. 7 would result from the animation of `ready[p3]`. When ProB animates a B operation, Meeduse gets the new variable valuations and then it translates back these valuations to the model resource in order to keep the equivalence between these valuations and the model resource. The result is an automatic visual animation directly showed in the MDE tools that are synchronised with the model resource.
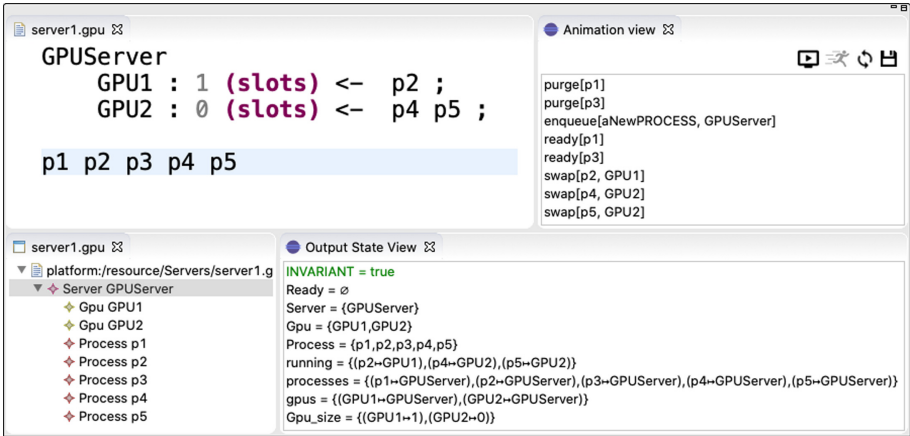

Fig. 8. DSL execution in Meeduse.

Several formal tools provide graphic animation and visualization techniques [11,19,22], which is intended to favour the communication between a formal methods engineer and the domain expert by using domain-specific visualizations. The contribution of Meeduse in comparison with these techniques is that the input model is provided by the domain expert using a dedicated language. Indeed, in tools like BMotion Studio [19], the domain-specific visualizations (textual or graphical) are created by the formal methods engineers who often would like to remedy the poor readability of their own specifications. We believe that visual animation may result in representations that lack of real-user perspective.

Furthermore, in visual animation tools, mapping a given graphical or textual representation to the formal specification is a rather time-consuming task (several days or several weeks as mentioned in [19]) and the creation of custom visualizations is often done when the formal model reaches an advanced stage during the modeling process. This may be counterproductive because the identification of misunderstandings often leads to enhancements of the formal specifications which in turn impacts the implementation of the visualization. In Meeduse, since the naming of the B data, generated from the DSL's static semantics, are not modified, the formal methods engineers do not need at all

to manage the visualizations by themselves. The Meeduse approach does not require any manual mapping between the domain-specific representations and the formal specification.

Meeduse presents also an advancement in comparison with existing approaches where DSLs are mixed with formal methods [4, 28]. In these works, once the formal model is defined (manually [4] or semi-automatically [28]), they don't offer any way to animate jointly the formal model and the domain model. These techniques start from a DSL definition, produce a formal specification and then they get lost in the formal process. In [28], the authors propose to use classical visual animation by applying BMotion Studio [19] to the formal specifications generated from the DSL. Our approach applies well-known MDE tools for DSL creation (EMF, Xtext, etc.) and automatically manages the equivalence between the formal model and the domain model.

Specifying typing and semantics rules within Xtext in a formal style was investigated by the Xsemantics tool [3]. The tool aims at filling the gap between the theory and the implementation of static type systems and operational semantics for Xtext-based DSLs. However, it does not provide formal tools for proving the correctness of these semantics. The detection of errors is done after they happen while Meeduse keeps the input model in a safe state-space regarding its invariant properties. We believe that the alignment of Meeduse with Xsemantics is an intersting perspective because typing rules as defined by Xsemantics can be seen in our case as invariants that must not be violated.

4   Defining the Domain-Specific Actions

The dynamic semantics of the DSL can be defined as additional B specifications with specific invariants and operations that use the data structures issued from the static semantics. Meeduse offers two strategies to define these specifications: (1) the meta-model based approach that generates presetted utility operations from the meta-model, and (2) the CP-net approach in which the domain-specific actions are first defined using coloured Petri-nets and then translated into B.

4.1   The Meta-model Based Approach

The meta-model based approach generates a list of presetted utility operations: getters, setters, constructors and destructors. Figure 9 gives operations extracted for class Process in order to manage link running with class Gpu.

Operation Process_SetGpu creates a link between a GPU (parameter $aGpu$) with a given process (parameter $aProcess$) if the process is not already linked to the GPU ($\{(aProcess \mapsto aGpu)\} \not\subseteq running$). Process_UnsetGpu is the reverse operation; it removes the link if it already exits.

The utility operations are correct by construction with respect to the invariants produced automatically from the meta-model structure. Indeed, if the structural invariants are not manually modified, the AtelierB prover should be able to

$Process\_SetGpu(aProcess, aGpu) =$
  PRE
      $aProcess \in Process \land aGpu \in Gpu$
      $\land \{(aProcess \mapsto aGpu)\} \not\subseteq running$
  THEN
      $running := (\{aProcess\} \lhd running)$
                $\cup \{(aProcess \mapsto aGpu)\}$
  END;

$Process\_UnsetGpu(aProcess) =$
  PRE
      $aProcess \in Process$
      $\land aProcess \in \mathrm{dom}(running)$
  THEN
      $running := \{aProcess\} \lhd running$
  END;

Fig. 9. Example of basic setters.

prove the correctness of the utility operations regarding these invariants. Otherwise, the operations for which the proof fails must be updated also manually. For our example, the structural invariants were automatically generated and then the resulting utility operations didn't require any manual modification. Meeduse generated a B specification whose length is about 245 lines of code, with 27 utility operations from which the AtelierB prover generated 43 proof obligations and proved them automatically.

The advantage of these utility operations is that they guarantee the preservation of the static semantics. In the following, we will use the inclusion mechanism of the B method in order to apply them for the formal specification of the domain-specific actions (`enqueue`, `purge`, `ready` and `swap`). Figure 10 gives the header part and the invariant clause of the proposed specification.

MACHINE *DynamicSemantics*
INCLUDES *GPUModel*
VARIABLES
    Ready
INVARIANT
    $Ready \subseteq Process \land$
    $\mathrm{dom}(running) \cap Ready = \emptyset \land$
    $(\exists gpu . (gpu \in Gpu \land Gpu\_size(gpu) > 0) \Rightarrow Ready = \emptyset)$

Fig. 10. Machine DynamicSemantics.

As mentioned in the informal description of our simple DSL, there are active, ready and waiting processes. In this specification, states active and waiting are somehow implicit. The domain of relation *running* ($\mathrm{dom}(running)$) represents active processes. Thus, processes that are not active, are even ready, if they are member of set *Ready* or waiting, otherwise. For space reason, we give only the example of operation `ready` (Fig. 11). This operation selects a waiting process ($pp \in Process - (\mathrm{dom}(running) \cup Ready)$) and then it decides to activate it (if there exists a free slot) or to change its state to ready (if all GPUs are busy).

The activation of a process calls two utility operations from machine *GPUModel*: `Process_SetGPU` and `GPU_SetSize`.

The AtelierB prover generated from machine DynamicSemantics 39 proof obligations; 32 were proved automatically and 7 required the use of the interactive prover. These proofs attest that the domain specific actions, written in B, preserve the invariants of both dynamic and static semantics.

```
ready =
   ANY pp WHERE pp ∈ Process - (dom(running) ∪ Ready) THEN
      IF ∃ gpu . (gpu ∈ Gpu ∧ Gpu_size(gpu) > 0) THEN
         ANY gg WHERE gg ∈ Gpu ∧ Gpu_size(gg) > 0 THEN
            Process_SetGpu(pp,gg) ;
            Gpu_SetSize(gg, Gpu_size(gg) - 1)
         END
      ELSE
         Ready := Ready ∪ {pp}
      END
   END ;
```

Fig. 11. Operation `ready`.

4.2 The CP-net Approach

Meeduse offers a translation of coloured Petri-net models (CP-nets [18]) into B in order to help build the dynamic semantics using a readable graphical notation. CP-nets combine the strengths of classical Petri-nets (*i.e.* formal semantics) with the strengths of high-level visual languages (*i.e.* communication and readability) [10]. A CP-net model is an executable representation of a system consisting of the states of the system and the events or transitions that cause the system to change its state. Despite of a small basic vocabulary, CP-nets allow great flexibility in modeling a variety of application domains, including communication protocols, data networks, distributed algorithms, embedded systems, etc. All these domains apply their own DSLs and hence the CP-net approach of Meeduse coupled with DSLs, can have a wide range of applications.

The two main notions of CP-nets are: Places and Transitions. Places represent abstractions on data values (called tokens or colours in the CP-net vocabulary). A place is related to a data-type (called colour-set) that can be simple (*i.e.* Integer, Boolean, etc.) or complex (*i.e.* sequences, products, etc.). In Meeduse, colour-sets refer to the possible types provided by the B language. Regarding transitions, they are linked to input and output places. When fired, a transition consumes tokens from its input places and introduces tokens into its output places. In our approach, places represent B variables and transitions are B operations. We identify three kinds of places: existing, new and derived.

– Existing: refer to the B variables extracted from the meta-model (Fig. 5). These places must be assigned to the variables of the B machine.
– New: refer to additional variables that are useful to define the DSL's behaviour, such as the Ready state of processes. For these places the user must provide its type and initial value using the B language.
– Derived: refer to variables whose values are defined from other B data, such as the Active set of processes. Derivation rules are also written in B.

Figure 12 is a screen-shot of the CP-net component of Meeduse. It shows the invariant properties, the definition of free slots and the derivation rules for derived places. The figure also gives the CP-net of operation `ready` together with the corresponding B specification. This operation is designed by means of two CP-net transitions with different guards $[freeSlots \neq \emptyset]$ and $[freeSlots = \emptyset]$. Places Gpu_size and running refer to existing variables and place Ready introduces a new one that is a subset of variable *Process*. This place is initialized to the empty set. Places Waiting and Active are derived with the following rules: Waiting = *Process* – (*Active* ∪ *Ready*) and *Active* = **dom**(*running*). When the


```
ready =
ANY pp WHERE pp ∈ Process ∧ pp ∈ Waiting THEN
    SELECT freeSlots ≠ ∅ THEN
        ANY gg, ss WHERE
            gg ∈ Gpu ∧ gg ∈ freeSlots
            ∧ ss ∈ NAT ∧ (gg ↦ ss) ∈ Gpu_size
        THEN
            running := running ∪ {(pp ↦ gg)} ||
            Active := Active ∪ {pp} ||
            Gpu_size := (Gpu_size - {(gg ↦ ss)}) ∪ {(gg ↦ ss - 1)}
        END
    WHEN freeSlots = ∅ THEN
        Ready := Ready ∪ {pp}
    END ||
    Waiting := Waiting - {pp}
END ;
```
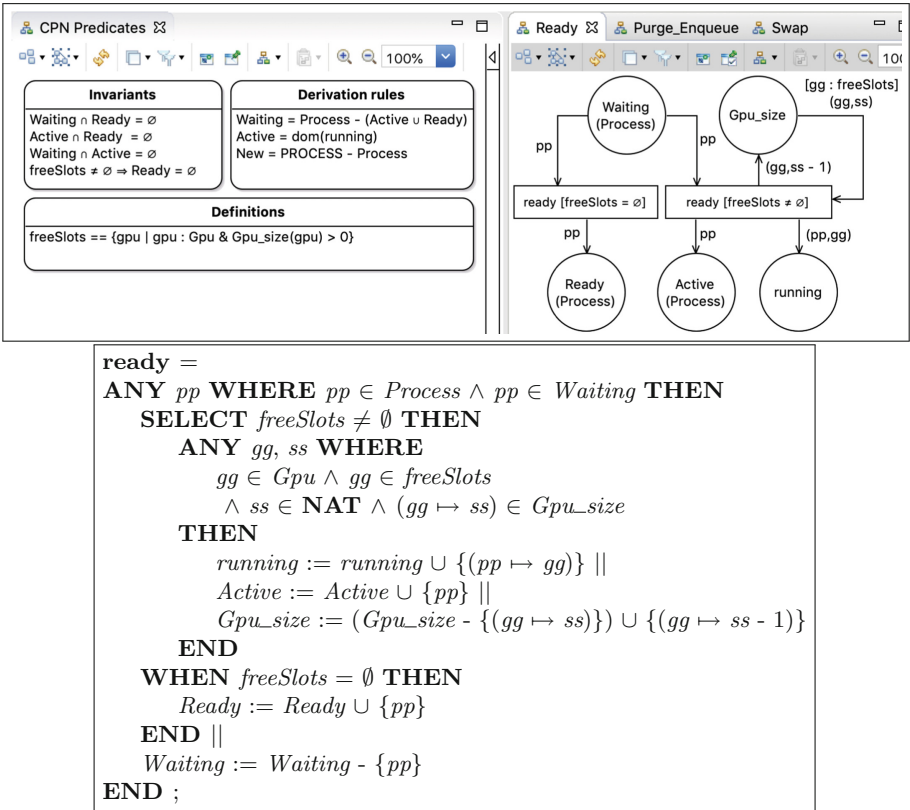
Fig. 12. The CP-net component of Meeduse.

GPU server is busy, transition `ready[freeSlots = ∅]` can be triggered, which consumes a token `pp` from place Waiting and introduces it into place Ready. If there exists a free slot, transition `ready[freeSlots ≠ ∅]` consumes token `pp` but introduces it in place Active. In this case, the transition also looks at a couple of tokens `(gg,ss)` from place Gpu_size such that `gg ∈ freeSlots`. The couple is consumed and replaced by couple `(gg,ss-1)`, and tokens `pp` and `gg` are introduced together in place running. In the B specification, a token t is selected from a place P, whose colour-set is C, using substitution: **ANY** t **WHERE** $t \in C \land t \in P \land condition$ **THEN**. Guards are translated into guards of the **SELECT/WHEN** substitutions. Regarding actions, they represent the consumption and production mechanism of CP-nets using set union and set subtraction.

In the CP-net approach the additional data structures, invariants, definitions and operations are injected in the B specification of the meta-model. The B specification produced by this technique is about 135 lines, for which the AtelierB prover generated 69 proof obligations: 53 were proved automatically and 16 interactively. Figure 13 gives the CP-nets of the other operations: purge, enqueue and swap.

We believe that the CP-net approach provides a good visualization of the dynamic semantics thanks to graphical views. The resulting models are much more accessible for stakeholders who are not trained in the B method than the meta-model based approach. However, this approach produces less concise B specifications (*e.g.* 11 lines for Fig. 11 against 16 lines for Fig. 12) and generates several additional variables due to the derived places that are often required. The number of proof obligations for the dynamic semantics is then greater than the meta-modeling approach (69 POs for the CP-net approach against 39 for the meta-modeling approach).
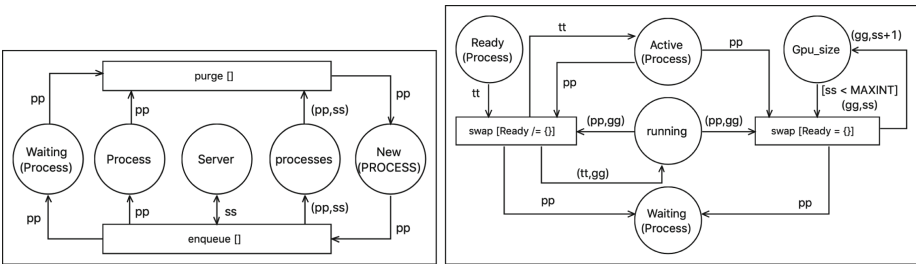

Fig. 13. Enqueue, purge and swap.

5 Evaluation

Two realistic case studies were developed and showed the viability of the tool: (1) a railway DSL for which the CP-net approach was fully exploited [14,15], and (2) a model-to-model transformation that applies the meta-model based approach to transform a given DSL into another one [17].

5.1   A Formal Railway DSL

In contrast with the textual language developed in this paper, this application of Meeduse defines a graphical DSL that can be used by railway experts to design railroad topologies and simulate train behaviours (Fig. 14). This work starts from two main observations: first, in railway control and safety systems the use of formal methods is a strong requirement; and second, graphical representations of domain concepts are omnipresent in railway documents thanks to their ability to share standardized information with common knowledge about several mechanisms (*e.g.* track circuits, signalling rules, etc.). Meeduse showed its strength to mix both aspects in the same tool.
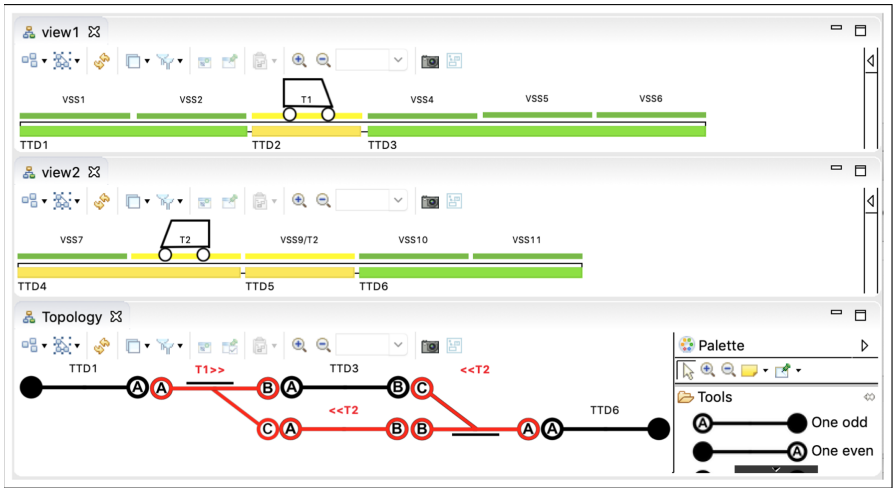

Fig. 14. Application of Meeduse to a railway case study.

We fully applied the CP-net approach to define the dynamic semantics of this DSL and represent train movements, assignment of routes to trains, modifications of switches positions, etc. This application deals with several safety-critical invariants for which theorem proving was applied in order to guarantee an accident-free behavior. The CP-net models were introduced incrementally using three proved refinement levels. The numbers of proofs generated from these refinements are presented in Table 1.

Table 1. Proof obligations generated from the railway DSL

	POs	Automatic	Manual
Level 1	17	11	6
Level 2	32	25	7
Level 3	62	41	21

5.2   A Formal DSL Transformation

We applied Meeduse to define a real-life DSL transformation [17]. Figure 15 shows the input and output models of the transformation: the input model is a truth table and the output model is a binary decision diagram (BDD). This application, carried out during the 12th edition of the transformation tool contest (TTC'19) won the award of best verification and the third audience award. The meta-model based approach was applied to take benefit of the utility operations and define B operations that consume truth table elements and progressively produce a binary decision diagram.
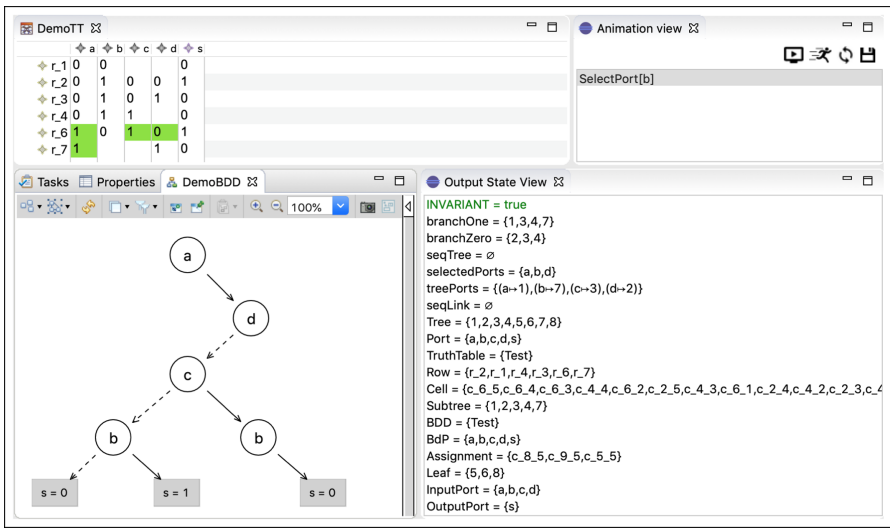

Fig. 15. Application of Meeduse to DSL transformation.

The B machine of the meta-model is about 1162 lines of code. 260 proof obligations were generated and automatically proved by the AtelierB prover, which guarantees that the static features of the output BDD are preserved during the DSL transformation. Regarding the dynamic semantics, they were specified by five B operations that are defined in an additional B machine whose length is about 150 lines of code. The correctness of the dynamic semantics was ensured by model-checking, rather than by theorem proving because on the one hand it is less time consuming, and on the other hand, it deals with bounded state spaces that can be exhaustively checked by the ProB [21] model-checker. The model-checking proof shows that both input and output models are equivalent.

6   Conclusion

When an executable DSL is not formally checked, it may lead to a succession of failures: failures of modeling operations (*e.g.* define a negative value for free

slots) may result in failures of domain-specific operations (*e.g.* assign more processes to a GPU than its capacity), which in turn may result in failures of the coordination operations (*e.g.* wrong process scheduling algorithms). This paper gave an overview of Meeduse, a tool dedicated to build and run correct DSLs by formally defining their underlying static and dynamic semantics. It allows to develop DSL tools intended to be used by domain experts whose requirement is to apply domain-specific notations to model critical systems and to correctly simulate their behaviour. In addition to the benefits of the tool for DSL development, the proposed technique is a more pragmatic domain-centric animation than visual animation techniques provided by formal tools because the domain-centric representations are provided by the domain expert himself who has a greater knowledge of the application domain than the formal methods engineer.

Several state-based formal methods can get along with the tool as far as these methods are assisted by publicly available parsers and animators. Some research works have been devoted in the past to apply a formal method, such as MAUD [25] or ASM [9], for the verification of a DSL's semantics. Although these works are close to Meeduse, they don't cover the joint execution of the DSL and the formal model. The transformations they propose can be integrated within Meeduse in order to enhance them with our technique for DSL animation and be able to experiment several target formal languages in a single framework. The use of B is mainly motivated by our long experience with the UML and B mappings and the availability of B4MSecure [13].

Currently we are working on two main perspectives: (1) provide a palette of proved DSLs (such as the BPMN language, or a DSL for home-automation) that are powered by Meeduse in order to make the underlying formal semantics much more accessible to non-practitioners of formal methods, and (2) propose a technique for DSLs composition that favours the execution of several DSLs together and make them collaborate. This perspective would lead to the execution of several instances of ProB with the aim to animate jointly several heterogeneous models.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham (2016)
3. Bettini, L.: Implementing type systems for the IDE with Xsemantics. J. Log. Algebraic Meth. Program. **85**(5, Part 1), 655–680 (2016)
4. Bodeveix, J.-P., Filali, M., Lawall, J., Muller, G.: Formal methods meet domain specific languages. In: Romijn, J., Smith, G., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 187–206. Springer, Heidelberg (2005). https://doi.org/10.1007/11589976_12
5. Bousse, E., Leroy, D., Combemale, B., Wimmer, M., Baudry, B.: Omniscient debugging for executable DSLs. J. Syst. Softw. **137**, 261–288 (2018)
6. Clearsy: Atelier B. https://www.atelierb.eu/en/

7. Dave, M.A.: Compiler verification: a bibliography. SIGSOFT Softw. Eng. Notes **28**(6), 2 (2003)

8. Dghaym, D., Poppleton, M., Snook, C.: Diagram-led formal modelling using iUML-B for hybrid ERTMS level 3. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 338–352. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_23

9. Gargantini, A., Riccobene, E., Scandurra, P.: Combining formal methods and MDE techniques for model-driven system design and analysis. Adv. Softw. **3**(1&2) (2010)

10. Gehlot, V., Nigro, C.: An introduction to systems modeling and simulation with colored petri nets. In: Proceedings of the 2010 Winter Simulation Conference, WSC 2010, USA, 5–8 December 2010, pp. 104–118 (2010)

11. Hallerstede, S., Leuschel, M., Plagge, D.: Validation of formal models by refinement animation. Sci. Comput. Program. **78**(3), 272–292 (2013)

12. Hartmann, T., Sadilek, D.A.: Undoing operational steps of domain-specific modeling languages. In: Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling, DSM 2008, University of Alabama at Birmingham (2008)

13. Idani, A., Ledru, Y.: B for modeling secure information systems. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 312–318. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_20

14. Idani, A., Ledru, Y., Ait Wakrime, A., Ben Ayed, R., Bon, P.: Towards a tool-based domain specific approach for railway systems modeling and validation. In: Collart-Dutilleul, S., Lecomte, T., Romanovsky, A. (eds.) RSSRail 2019. LNCS, vol. 11495, pp. 23–40. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-18744-6_2

15. Idani, A., Ledru, Y., Ait Wakrime, A., Ben Ayed, R., Collart-Dutilleul, S.: Incremental development of a safety critical system combining formal methods and dsmls. In: Larsen, K.G., Willemse, T. (eds.) FMICS 2019. LNCS, vol. 11687, pp. 93–109. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-27008-7_6

16. Idani, A., Ledru, Y., Vega, G.: Alliance of model-driven engineering with a proof-based formal approach. Innov. Syst. Softw. Eng., 1–19 (2020). https://doi.org/10.1007/s11334-020-00366-3

17. Idani, A., Vega, G., Leuschel, M.: Applying formal reasoning to model transformation: the meeduse solution. In: Proceedings of the 12th Transformation Tool Contest, co-located with STAF 2019, Software Technologies: Applications and Foundations. CEUR Workshop Proceedings, vol. 2550, pp. 33–44 (2019)

18. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, vol. 1. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-662-03241-1

19. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising event-b models with b-motion studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 202–204. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04570-7_17

20. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**, 107–115 (2009)

21. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Softw. Tools Technol. Transf. (STTT) **10**(2), 185–203 (2008)

22. Li, M., Liu, S.: Integrating animation-based inspection into formal design specification construction for reliable software systems. IEEE Trans. Reliab. **65**, 1–19 (2015). https://doi.org/10.1109/TR.2015.2456853

23. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: executable DSMLs based on fUML. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 56–75. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_4

24. OMG: OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1 (June 2013). http://www.omg.org/spec/MOF/2.4.1

25. Rivera, J., Durán, F., Vallecillo, A.: Formal specification and analysis of domain specific models using Maude. Simulation **85**, 778–792 (2009)

26. Snook, C., Savicks, V., Butler, M.: Verification of UML models by translation to UML-B. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 251–266. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_13

27. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, vol. 2. Addison-Wesley, Boston (2008)

28. Tikhonova, U., Manders, M., van den Brand, M., Andova, S., Verhoeff, T.: Applying model transformation and event-b for specifying an industrial DSL. In: MoDeVVa@ MoDELS, pp. 41–50 (2013)

29. Wachsmuth, G.: Modelling the operational semantics of domain-specific modelling languages. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 506–520. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88643-3_16