Noname manuscript No.
(will be inserted by the editor)

Managing Design-Time Uncertainty

Michalis Famelis · Marsha Chechik

Received: date / Accepted: date

Abstract Managing design-time uncertainty, i.e., uncertainty that developers have about making design decisions, requires creation of "uncertainty aware" software engineering methodologies. In this paper, we propose a methodological approach for managing uncertainty using partial models. To this end, we identify the stages in the life-cycle of uncertainty-related design decisions, and characterize the tasks needed to manage it. We encode this information in the Design-Time Uncertainty Management (DeTUM) model. We then use the DeTUM model to create a coherent, tool-supported methodology centered around partial model management. We demonstrate the effectiveness and feasibility of our methodology through case studies.

1 Introduction

The development of any software system entails making decisions. However, developers often face uncertainty about making such decisions. This form of uncertainty, known as *design-time uncertainty* [?], concerns the *content* of a software system, and is different from uncertainty about the *environment* in which the system is meant to operate (known as *environmental uncertainty*). In other words, it is uncertainty that the developer has about what the system should be like, rather than about what conditions it may face during its operation.

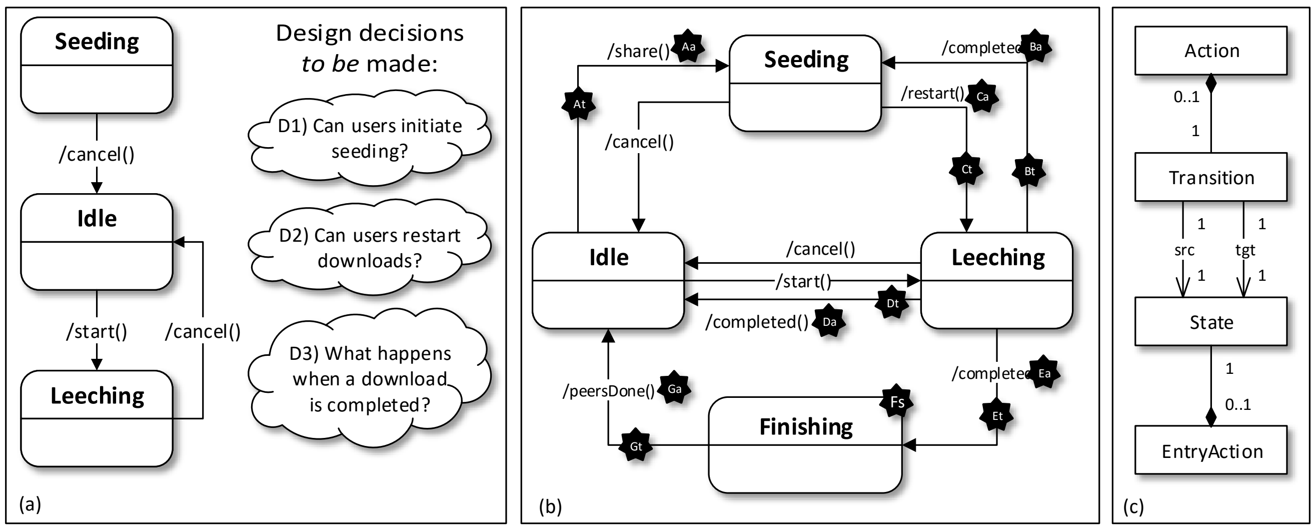
M. Famelis
Université de Montréal
E-mail: famelis@iro.umontreal.ca

M. Chechik
University of Toronto
E-mail: chechik@cs.toronto.edu

To tackle environmental uncertainty, developers use various strategies such as self-adaptation [?], probabilistic behaviour [?], identifying and explicating operational assumptions [?], etc. These mitigation strategies result in functional systems that can operate under uncertain conditions. Thus, mitigating environmental uncertainty entails creation of *uncertainty-aware software.* In contrast, design-time uncertainty cannot be "coded away" but must be taken into account in the process by which software is created. In other words, mitigating design-time uncertainty (henceforth, also simply "uncertainty") entails creation of *uncertainty-aware software development methodologies.* The reason for this is that existing software tools, languages and techniques assume that developers are able to make all relevant decisions, i.e., that their input does not contain any uncertainty. This renders design-time uncertainty an undesirable characteristic that needs to either be avoided or removed altogether before resuming development.

Thus, developers are forced to either refrain from using their tools until uncertainty is resolved, or to make provisional decisions and attempt to keep track of them in case they need to be undone. These options lead to either under-utilization of resources or potentially costly re-engineering attempts. In previous work, we have demonstrated that these are not the only viable strategies. Specifically, we have shown how different alternative design decisions can be encoded in a *partial model* [?] which can then be used to perform tasks such as reasoning [?], refinement [?] and transformation [?]. By implementing these techniques as *partial model operators* in Mu-Mmint, an interactive modelling tool, we have integrated them in a single model management IDE [?].

This points to an alternative tool-supported approach to managing uncertainty, centred around explicit uncer-



May formula for (b): $((\neg \mathtt{At} \wedge \mathtt{Bt} \wedge \neg \mathtt{Dt} \wedge \neg \mathtt{Et} \wedge \neg \mathtt{Fs}) \vee (\mathtt{At} \wedge \neg \mathtt{Bt} \wedge \mathtt{Dt} \wedge \neg \mathtt{Et} \wedge \neg \mathtt{Fs}) \vee (\mathtt{At} \wedge \neg \mathtt{Bt} \wedge \neg \mathtt{Dt} \wedge \mathtt{Et} \wedge \mathtt{Fs})) \wedge (\mathtt{At} \Leftrightarrow \mathtt{Aa}) \wedge (\mathtt{Bt} \Leftrightarrow \mathtt{Ba}) \wedge (\mathtt{Ct} \Leftrightarrow \mathtt{Ca}) \wedge (\mathtt{Dt} \Leftrightarrow \mathtt{Da}) \wedge (\mathtt{Fs} \Leftrightarrow \mathtt{Et} \Leftrightarrow \mathtt{EaGt} \Leftrightarrow \mathtt{Ga})$

Fig. 1 (a) Developing PtPP. Left: what is known, right: design decisions. (b) The M_{p2p} partial model. (c) Simplified state machine metamodel.

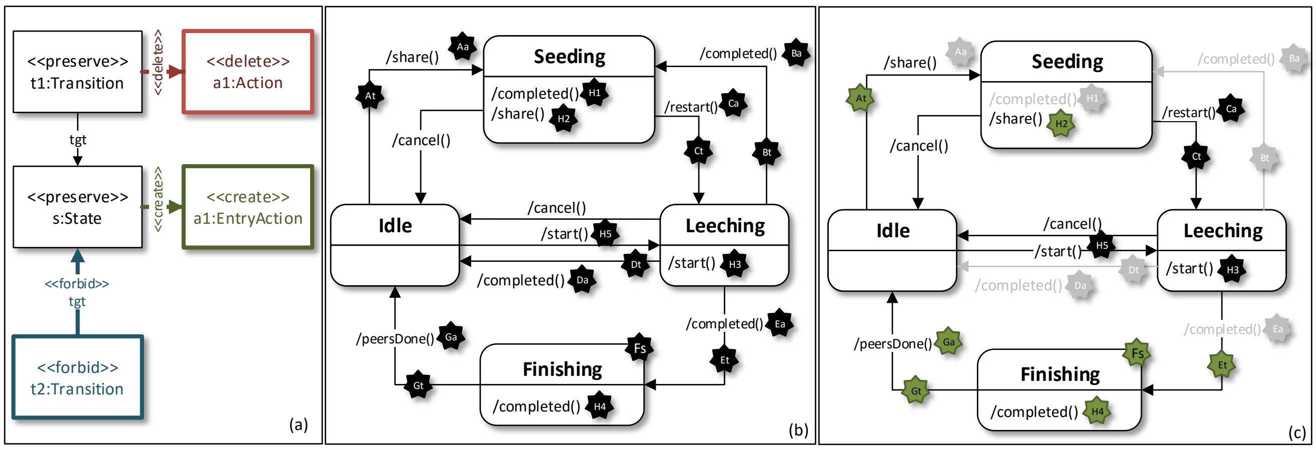
tainty modelling. Specifically, the combination of techniques for working with partial models allows deferring the resolution of uncertainty for as long as necessary. However, even though we have demonstrated using experimentation, benchmarking, and formal proofs that each partial model technique is individually feasible and practical, there is no guidance about how to use them in a concerted and coherent way. In other words, previously published partial model techniques are point solutions to narrowly defined problems and do not constitute a coherent methodology for tackling design-time uncertainty.

To create such a coherent methodology, we must answer some fundamental questions, such as: *What does it mean to "tackle" uncertainty? How does it get added to software artifacts? When does it get removed?* The first main contribution of this paper is thus understanding the lifecycle of design-time uncertainty, presented as the *"Design-Time Uncertainty Management"* (DeTUM) model. This allows a more detailed understanding of how uncertainty is manifested and manipulated at different stages, as well as the different inputs, outputs, pre- and post-conditions of partial model operators. Further, the DeTUM model allows us to to contextualize existing partial modelling operators and identify gaps through lessons learned from non-trivial scenarios. Thus, the second main contribution is a concerted, tool-supported uncertainty-aware software development methodology, centred on partial models. In all, this paper makes the following contributions: (1) the DeTUM model (2) the mapping of existing partial model operators to the DeTUM model, (3) based on the above, a methodology for managing design-time uncertainty, and (4) a validation of the usability and effectiveness of the methodology based on two non-trivial uncertainty management scenarios.

The rest of the paper is organized as follows: in Section 2 we introduce the PtPP example to help motivate and illustrate our approach. In Section 3, we describe the DeTUM model, which captures our proposed methodology for managing design-time uncertainty. In Section 4, we describe the various uncertainty management operators and their role in the DeTUM model. In Section 5, we introduce Mu-Mmint, an Eclipse-based IDE for managing uncertainty. In Section 6, we present two fully worked-out scenarios of uncertainty management and reflect on the lessons learned from the experience. We discuss related work in Section 7 and conclude the paper in Section 8.

2 Motivating Example

To illustrate and motivate our approach, we use a toy example developed by Famelis et al. [?]. The example uses state machines; however our approach applies to any modelling language. In the example, an engineering team developing a protocol for peer-to-peer downloads, called PtPP. The team has created an initial design of the protocol, shown in Figure 1(a), expressed in the simplified state machine metamodel in Figure 1(c). There is a state `Idle`, a state `Leeching` (sharing and downloading an incomplete file), and a state `Seeding` (sharing a complete file). When `Leeching` is initiated,



May formula for (b): ((¬At ∧ Bt ∧ ¬Ct ∧ ¬Dt ∧ ¬Fs)∨ (¬At ∧ Bt ∧ Ct ∧ ¬Dt ∧ ¬Fs)∨ (At ∧ ¬Bt ∧ ¬Ct ∧ Dt ∧ ¬Fs)∨ (At ∧ ¬Bt ∧ Ct ∧ Dt ∧ ¬Fs)∨ (At ∧ ¬Bt ∧ ¬Ct ∧ ¬Dt ∧ Fs)∨ (At ∧ ¬Bt ∧ Ct ∧ ¬Dt ∧ Fs))∧ (¬Aa ∧ ¬Ba ∧ ¬Ea)∧(Ct ⇔ Ca) ∧ (Dt ⇔ Da)∧(Fs ⇔ Et ⇔ Gt ⇔ Ga ⇔ H4)∧(H1 ⇔ Bt) ∧ (H2 ⇔ At) ∧ (H3 ⇔ ¬H5 ⇔ ¬Ct)
May formula for (c): (Ct ⇔ Ca) ∧ (H3 ⇔ ¬H5)

Fig. 2 (a) "Fold Single Entry Action" refactoring transformation. (b) The M_{p2p} model after refactoring. (c) Choosing the "compromising" candidate solution. The elements that reify the "compromise" candidate solution to D3 are set to True and highlighted in green. Elements reifying other candidate solutions to D3 are set to False and greyed out. Maybe elements that are not part of D3 are unaltered.

the action `start`() is executed. Both `Leeching` and `Seeding` can be cancelled to return to `Idle` by invoking the action `cancel`(). The PtPP developers are uncertain about three decisions, listed on the right of Figure 1(a): (D1) *Can users initiate seeding?* (D2) *Can downloads be restarted?* (D3) *what policy is followed when* `leeching` *ends?* The team further elicited three candidate solutions to (D3): (a) a "benevolent" option which automatically starts `Seeding`, (b) a "selfish" option described earlier, and (c) a "compromise" option where no new connections are accepted while waiting for existing peers to complete their copies.

The team uses a *partial model*, shown in Figure 1(b), to capture the space of candidate solutions to the three design decisions. In other words, M_{p2p} compactly encodes the set of models representing all possible ways to resolve uncertainty. Each such model is called a *concretization*. The full set of concretizations of M_{p2p} was developed by Famelis et al. [?].

The diagram of M_{p2p} consists of a model expressed in the same concrete syntax as the original PtPP model fragment, with the addition of annotations (represented as seven-pointed star icons) to some elements. These are called *Maybe elements* and are used to explicate points of uncertainty in the model. We use propositional variables as aliases for individual Maybe element decisions. E.g., the transition `share`() between `Idle` and `Seeding` has two seven-pointed star icons annotations, with the variables `At` and `Aa` for the transition and its action respectively. These indicate that the developer is unsure whether to include them in the model. E.g., setting `Aa` to True means that she decides to include the action `share`(), False to exclude it.

The partial model has an additional propositional *May formula*, shown below the diagram, which explicates dependencies between points of uncertainty. E.g., it specifies that each transition co-occurs with its corresponding action. Additionally, it specifies what are allowable configurations of Maybe elements if uncertainty is resolved. According to the solutions elicited by the team, the May formula allows three possible solutions to the policy about completed downloads: "benevolent", "selfish", and "compromise". As shown in Figure 1(a), a different point of uncertainty is whether users should be able to initiate seeding (i.e., having a transition `share` from `Idle`). The May formula in Figure 1, expresses the team's decision to correlate the two points of uncertainty, by allowing the ability to start seeding only for the "selfish" and the "compromise" behavioural scenarios.

Having expressed the space of candidate designs in a partial model, the team can now perform a variety of engineering tasks, *without* needing to resolve uncertainty. We outline some of them below, with references to previous work.

Transformation. The team may choose to perform refactoring, a common task done during development. Various kinds of refactoring can be done via a transformation [?], such as the rewriting rule "Fold Single Entry Action" (FSEA) shown in Figure 2(a). FSEA is expressed as a graph rewriting transformation rule [?] using the notation of the Henshin model transformation engine [?]. It tries to match a `State s` that has only one incoming `Transition t1`. The "negative application condition" (indicated by << `forbid` >>) stops the refactoring from being applied if there exists a second incoming `Transition t2`. Otherwise, it deletes `Action a1` of `t1` (indicated by << `delete` >>) and adds a new `EntryAction` with the same name (indicated by << `create` >>), associating it with `s`. Using transformation *lifting* [?], FSEA can be applied directly to M_{p2p}, resulting in the model shown in Figure 1(e). This new model is the same as if each concretization of M_{p2p} had been transformed separately and a new partial model was *constructed* [?] from the results.

Verification. The team may decide to perform some analysis on their model. E.g., they check a property P_1: *"no two transitions enabled in the same state can lead to the same target state"*. Such checks can be done using a specialized reasoning technique [?]. The result can be either True ("the property holds for all concretizations"), False ("it does not hold for any concretization"), or Maybe ("it only holds for some"). For the last two cases, it is desirable to generate a counterexample concretization for diagnosis. Checking P_1 for M_{p2p} results in Maybe, since the property only holds for some concretizations but not for others. For example, there exists a concretization with two outgoing transitions from `Leeching`, with actions `cancel`() and `completed`() respectively. There also exist concretizations that satisfy the property.

Refinement. Given the verification result, the team can ask for developer decide what to do next by generating appropriate feedback, such as a counter-example, or creating a partial model encoding all the counter-examples. This is a process of *removing* uncertainty (i.e., concretizations) from the partial model. It can be done declaratively, using "property-based refinement" [?], or operationally, by manually making decisions using a tool such as Mu-Mmint [?]. For example, the team may choose the "compromise" candidate solution for D3, resulting in the partial model shown in Figure 1(c).

While the PtPP team has such a variety of partial modelling techniques available to them, there is little guidance as to how to structure their different work tasks. How should the team derive M_{p2p} from the informal description in Figure 1(a)? What conditions should be in place to perform each task? For example, should should the team have refined if there had been a different verification result? In which order should each one be applied? For example, can the FSEA refactoring be done after refinement and vice versa? What are the results from doing them, i.e., what is the effect on the overall level of uncertainty? What should the team do if a new open design decision be encountered? These questions are not answered by each of the techniques individually. To answer them, we need to look at the bigger picture, to understand what tasks are appropriate at what points in the management of uncertainty.

3 Managing Uncertainty

The PtPP example already points us to a general outline of the "life" of a design decision about which a development team is uncertain. The space of candidate solutions is expanded and encoded in a partial model, which is then used as the primary artifact of development, until enough information is available to collapse the space of solutions. In other words, once uncertainty appears in the development process, a developer can attempt to capture it in a partial model. Using this, she can continue working while avoiding making decisions that, in the absence of enough information, would be premature. When sufficient information is available, she can systematically refine the partial model, such that the new information resolves the encoded uncertainty. We call these three basic stages Articulation, Deferral, and Resolution, respectively. Their succession follows a decreasing degree of uncertainty. Schematically, for a given design decision, uncertainty is introduced during the Articulation stage; it remains stable during the Deferral stage, and is reduced during the Resolution stage. We show this graphically in Figure 3, as a level of uncertainty present in the software models plotted over time.

In this paper, we assume that developers know what they are uncertain *about*. In other words, we do not examine the process of identifying which parts of a software system involve significant design decisions and what these are. E.g., the developers of PtPP know that they need to manage the decisions D1-D3. We also assume that the developers have some method of eliciting candidate solutions about each design decision. Therefore in the context of this paper, design-time uncertainty concerns a set of possible designs. In other words, we do not investigate the process by which developers elicit candidate solutions for a design decision posed as an open question. Instead, we assume that this process has already taken place and that for each design decision we are given a finite set of possible solutions. We also assume that the developers have concluded how to implement each candidate solution in their models. Partial models are then used to compactly and exactly encode the set of candidate solutions.

We introduce an abstract model for capturing the succession of stages of uncertainty management. This model is called *"Design-Time Uncertainty Management"* (DeTUM) model and presents an idealized timeline of

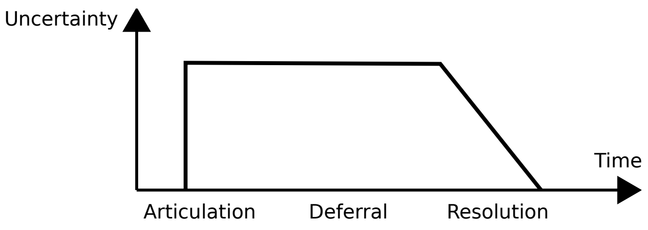


Fig. 3 Degree of uncertainty in different stages of the Design-Time Uncertainty Management (DeTUM) model.

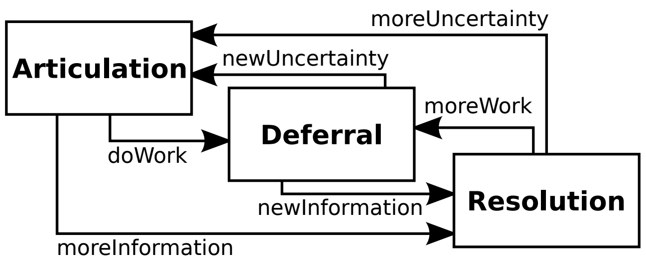


Fig. 4 The *Design-Time Uncertainty Management* (DeTUM) model: an idealized timeline of uncertainty management.

partial model use. It is shown in Figure 4 and it consists of the three stages: *Articulation* of uncertainty, *Deferral* of decisions, and *Resolution* of uncertainty. We discuss each one in more detail below.

Articulation stage: This stage is the entry point of uncertainty management: a developer is uncertain about some aspect of her model due to insufficient information. The main goal during this stage is therefore to change the model, so that it reflects the developer's uncertainty. In the PtPP scenario, this is stage when M_{p2p} is created, for each of the design decisions D1-D3. This involves eliciting a set of candidate solutions and producing the implementation for each. In this paper, we assume that these steps are handled by the developers or appropriate techniques such as Design Space Exploration [?,?]. During this stage, the degree of uncertainty in the software increases as the developer encodes in a partial model the different possible ways to resolve her uncertainty. These are encoded as concretizations of the partial model.

Deferral stage: In this stage, the main goal is to avoid premature decision making, while still being able to make use of software engineering techniques. To do that, the developer uses the partial model as the primary development artifact. The developer must therefore use versions of such techniques that have been appropriately *lifted* so that they are applicable to partial models. In PtPP, this is the stage where M_{p2p} is refactored or validated. During this stage *the degree of uncertainty in the model remains unchanged.* This is a fundamental property that lifted tasks must preserve.

Resolution stage: This stage begins when more information becomes available. The developer incorporates this new information into the partial model in a systematic way. E.g., the developers of PtPP acquire new information from the validation of the property P_1, which they use to refine M_{p2p}. In this paper we do not address issues stemming from randomness in the development context that impacts the underlying assumptions of a design decision and therefore the criteria for determining which solutions are acceptable candidates. Thus, during this stage, the degree of uncertainty in the partial model is reduced to reflect the newly acquired information. The ultimate result of this stage is a model without any partiality, i.e., a concretization of the original partial model.

The DeTUM model is an abstraction of an inherently messy process. It should not be misunderstood as a rigid prescription of a strict succession of stages, where an initial explication of uncertainty is necessarily followed by a series of partial models of monotonically decreasing uncertainty that culminates in a single concretization. Similarly, the plot in Figure 3 is also an abstraction. We offer the DeTUM model as an intuitive description of the stages of uncertainty management and the dependencies between them. In fact, the stages may overlap or be used in different orders. For example, the developers of PtPP might encounter only the decision D1, and encode its candidate solutions in a partial model M_1; apply FSEA to get M_2; encounter decisions D2 and D3 and encode their candidate solutions in M_2 to create M_3, etc.

The DeTUM model contains the following transitions between stages of uncertainty management:

doWork (Articulation→Deferral)
: Once the developer has completed expressing uncertainty in her software artifacts, she can use her partial models to perform software engineering tasks. This transition is triggered once there is no more uncertainty to express and the developer needs to continue working. E.g., once the PtPP team has created M_{p2p} they can proceed to do work such as refactoring.

newInformation (Deferral→Resolution)
: If new information becomes available, the developer can use it to resolve all or part of her uncertainty. This transition is triggered if the developer is in a position to resolve some of the uncertainty and wishes to do so. E.g., the result of checking the prop-

erty P_1 on M_{p2p} prompted the team to remove concretizations that violate it.

moreWork (Resolution→Deferral)

If uncertainty is only partially resolved, the developer can continue performing lifted engineering tasks while the remaining uncertainty persists. For example, the developer may perform a verification task, use the results to resolve some of the uncertainty, and then continue working with refined partial model. This transition is triggered once no more uncertainty can be resolved and the developer needs to continue working. E.g., after resolving the decision D3 by choosing the "compromise" candidate solution, the team can continue working with the partial model, such as doing additional verification.

newUncertainty (Deferral→Articulation)

In the course of her work with partial models, the developer might encounter additional points of uncertainty. For example, while already working with a partial model, she may become uncertain about some design decision for which there was previously no uncertainty. This transition is triggered when the developer faces the need to express further uncertainty in her software artifacts. E.g., the after refactoring M_{p2p}, the PtPP team might notice that they also need to address a new, open design decision D4 "what happens if the protocol is disconnected". Therefore, they would need to elicit candidate solutions for D4 and encode them in the partial model.

moreUncertainty (Resolution→Articulation)

The developer might need to articulate additional uncertainty immediately following a resolution. Similarly to the transition *newUncertainty*, this transition is triggered when the developer faces the need to express further uncertainty in her software artifacts. E.g., the team might discover D4 after resolving the decision D3, in which case they should skip the Deferral stage and work on encoding the candidate solutions of D4 in M_{p2p}.

moreInformation (Articulation→Resolution)

The developer might acquire information that allows her to resolve some uncertainty immediately following articulation. Similarly to the transition *newInformation*, this transition is triggered if the developer is in a position to resolve some of the uncertainty and wishes to do so. E.g., after creating M_{p2p}, the team might receive new requirements stating that users should indeed always be able to initiate seeding, thus resolving D1.

"Undo" transitions

In addition, we also allow *Undo* transitions between any two stages. For example, the transition:

Undo(Resolution→Articulation)

allows the developer undo some refinement in order to explore different alternatives, while the transition:

Undo(Articulation→Resolution)

allows the developer to revert to a previous a more concrete version of her artifacts if she decides there is no benefit to articulating a particular point of uncertainty. Undo transitions are triggered whenever the developer is not satisfied with the immediately previous transition she took. E.g., after resolving D3, the PtPP team might decide that the property P_1 is not that important and thus choose to undo the refinement.

The backward links to the Articulation stage can also be the result of eliciting possible solutions from a previously open design decision. E.g., faced with D3, the PtPP developers might initially be unsure about what are candidate solutions and thus leave it open ended. When they arrive to the three candidate solutions ("selfish", "benevolent", and "compromise"), the open-ended decision D3 is reduced to uncertainty about selecting one of them. The backward transitions to the Articulation stage, therefore allow this process of gradual elicitation of specific alternatives to an open-ended decision point.

The combination of the transitions *newInformation* and *moreWork* captures the fact that resolution of uncertainty is not always immediate (i.e., producing a concrete, non-partial model in one step) but rather that the Deferral and Resolution stages can be interleaved. This interleaving represents the gradual removal of uncertainty. For example, the developer might apply a transformation to her model, then resolve some of the uncertainty, check a property, apply a second transformation, and so on.

Finally, we note that the DeTUM model also illustrates the fundamentally *transient* nature of partial models: they are created when uncertainty is encountered, used as primary development artifacts in the presence of uncertainty and are ultimately discarded, collapsing to a single, concrete model.

4 Uncertainty Operators

In this paper, we propose an approach for managing uncertainty in software. Our approach for managing uncertainty in models entails doing *model management* [**?**] of models that contain uncertainty, i.e., *partial models*. The various partial model techniques are thus construed as model management *operators*.

The DeTUM model identifies three distinct usage contexts (Articulation, Deferral, Resolution) in which developers work with partial models. Below, we we describe the operators that pertain to each stage. The list is not complete, however it contextualizes existing work and identifies some obvious omissions. Each operator is described in a table that outlines:

(a) its name,
(b) a high level description of its functionality,
(c) its inputs and outputs,
(d) a description of the setting in which it is meaningful to invoke it,
(e) its preconditions and postconditions,
(f) its limitations in terms of effectiveness or usability, and
(g) the section in the paper where its implementation is described in detail.

4.1 Articulation stage

In this stage operators aim at explicating the developer's uncertainty about some design decision. Inputs to articulation operators are therefore inherently informal and subjective. We identify three such operators: *Construct*, *MakePartial*, and *Expand*.

The operator *Construct* creates a partial model from a set of concrete models, as described in Table 1.

The *Construct* operator assumes that the developer has full knowledge of the set of concretizations before invoking it. However, the articulation process can also be manual, requiring the intuition of the developer to appropriately capture the space of possibilities in the partial model. This is done using the operator *MakePartial*, described in the Appendix Table 3.

If uncertainty is encountered while the developer is already working with a partial model, the operator *Expand*, described in the Appendix Table 4, allows expanding the existing partial model with the new possibilities.

The common characteristic of the operators of the Articulation stage is that the size of the set of concretizations increases.

4.2 Deferral stage

The aim of operators in this stage is to facilitate decision deferral: if a developer can accomplish software engineering tasks using partial models, then there is no need to prematurely remove uncertainty. We therefore *lift* software engineering operators such they can be used with partial models, without affecting the degree of uncertainty. We identify two such operators: *Transform*, and *Verify*.

The operator *Transform*, described in the Appendix Table 5, allows developers to apply a model transformation to a partial model such that all it concretizations are correctly transformed, albeit without having to enumerate them.

The operator *Verify*, described in the Appendix Table 6, is used to determine whether a partial model satisfies a syntactic property.

Following the verification of a property, the developer may want to perform diagnostics, to determine the underlying reasons for the verification result. Since diagnostic operations appropriately remove uncertainty to illuminate parts of the set of the input partial model's concretizations, they are discussed in the next section.

During the deferral stage if some software engineering operation (in addition to the ones described above) is required, then it needs to be appropriately lifted. For arbitrary operations, correct lifting generally depends on the semantics of the base language and is thus outside the scope of this paper. It is however always possible (albeit expensive) to enumerate all concretizations, apply the non-lifted operation to each concretization separately, and then merge the results using *Construct*. We thus also introduce the operator *Deconstruct*, described in the Appendix Table 7, that, given a partial model, creates its set of concretizations. This can be accomplished by passing the partial model's May formula to an All-Solutions SAT solver [?]. All-Solutions SAT solvers are special purpose reasoning engines that specialize in efficiently computing all satisfying valuations of a boolean formula. Thus, the solver will enumerate all satisfying valuations of the May formula, which can then be translated into models [?].

Unlike operators in the Articulation and Resolution stages, that increase and reduce the degree of uncertainty in partial models respectively, operators in the Deferral stage do not affect the degree of uncertainty.

4.3 Resolution stage

Operators at this stage incorporate new information to a partial model, thus reducing its degree of uncertainty. All the operators can produce either a new partial model that refines the original, or a non-partial model, i.e., a concretization of the original. We describe two classes of operators for this stage. On the one hand, the *Decide* and *Constrain* operators incorporate new information obtained by the developer. Using them, the developer can resolve (partially or completely) the open questions that prompted the creation of partial models during the Articulation stage. On other hand, the diagnostic operators *GenerateCounterExample*, *Genera-*

Table 1 Operator *Construct*

Description	Create a partial model from a given set of concrete models that are alternative resolutions to uncertainty.
Inputs	A set of non-partial models.
Outputs	A partial model.
Usage context	The developer has at her disposal a known, fully enumerated set of alternative models, but has insufficient information about which of the models is best suited for her purpose.
Preconditions	No partial model exists. The set of models must be known and fully enumerated.
Postconditions	The resulting partial model is in Graphical Reduced Form (GRF) and its set of concretizations is exactly the set of input models.
Limitations	The developer must have the full knowledge of the input set.
Implementation	The operator is described in [?] as operator "OP1: Construction".

teExample, and *GenerateDiagnosticCore* produce feedback following an invocation of the *Verify* operator. They remove uncertainty from the input partial model in order to produce subsets of its concretizations to appropriately illuminate the results of verification.

The *Decide* operator, described in the Appendix Table 8, allows the developer to manually make decisions about which elements should and should not remain in the model. The *Constrain* operator, described in the Appendix Table 9, allows narrowing the set of concretizations of a partial model to a subset that satisfies some property of interest.

We define three diagnostic operators. The operator *GenerateCounterExample*, described in the Appendix Table 10 creates a concretization that functions as a witness as to why a partial model does not satisfy some property of interest. The operator *GenerateExample*, described in the Appendix Table 11, creates a concretization that functions as a witness as to why a partial model can satisfy some property of interest, depending on how uncertainty is resolved. The operator *GenerateDiagnosticCore*, described in the Appendix Table 12, creates a partial model that encodes the subset of concretizations of the input partial model that do not satisfy a property of interest. The common characteristic of the operators of the Resolution stage is that the size of the set of concretizations decreases.

We summarize the set of uncertainty management operators by overlaying them on the DeTUM model in Figure 5. Specifically, the operators *Construct*, *MakePartial*, and *Expand* are part of the Articulation stage, the operators *Transform*, *Verify*, and *Deconstruct* are part of the Deferral stage, and the operators *Decide*, *Constrain*, *GenerateCounterExample*, *GenerateExample*, and *GenerateDiagnosticCore* are part of the Resolution stage.

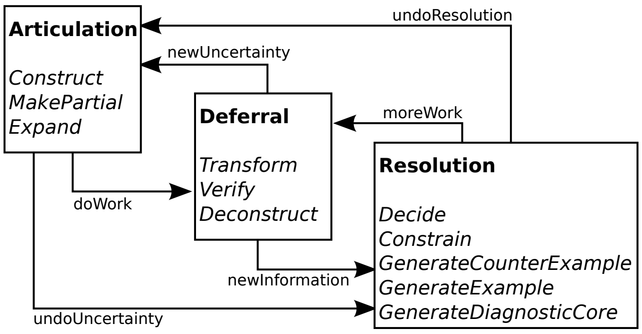


Fig. 5 Uncertainty management operators overlayed on the DeTUM model.

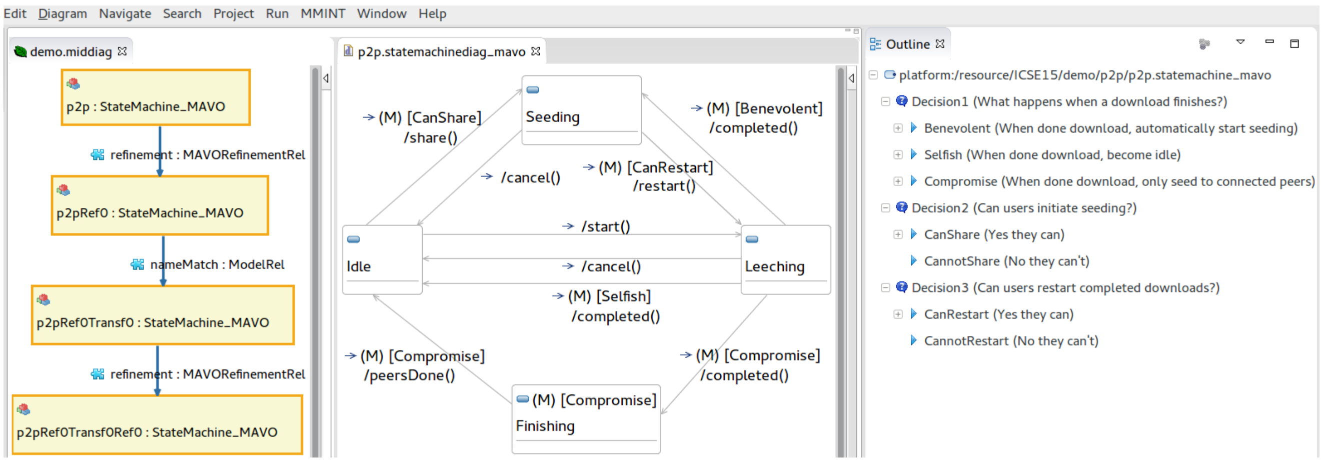
5 Tool Support

In this section, we present Mu-Mmint[1], a tool that implements management of models with uncertainty. Mu-Mmint is an Eclipse-based tool for model management [?] of partial models. It was created as an Integrated Development Environment (IDE) that bundles various uncertainty management operators in one coherent unit. We illustrate the main features of Mu-Mmint using the PtPP motivating example, introduced in Section 2.

The workspace of Mu-Mmint is an interactive megamodel [?], shown in the left panel of Figure 6, which allows modellers to create, manipulate and interact with partial models using the uncertainty management operations from the DeTUM model, shown in Figure 5.

Articulation stage. In Mu-Mmint, articulating uncertainty is done using the *MakePartial* and *Expand* uncertainty operators. We illustrate this in PtPP. Initially, the team's design is separated into known and unknown parts, as shown in Figure 1(a). In Mu-Mmint, developers can explicate this information in a single partial model M_{p2p}, shown in the middle and right panels of Figure 6.

[1] Available at: http://github.com/adisandro/mmint


Edit Diagram Navigate Search Project Run MMINT Window Help
demo.middiag
p2p : StateMachine_MAVO
refinement : MAVORefinementRel
p2pRef0 : StateMachine_MAVO
nameMatch : ModelRel
p2pRef0Transf0 : StateMachine_MAVO
refinement : MAVORefinementRel
p2pRef0Transf0Ref0 : StateMachine_MAVO
p2p.statemachinediag_mavo
(M) [CanShare] /share()
Seeding
(M) [Benevolent] /completed()
/cancel()
(M) [CanRestart] /restart()
/start()
Idle
/cancel()
Leeching
(M) [Selfish] /completed()
(M) [Compromise] /peersDone()
(M) [Compromise] /completed()
(M) [Compromise] Finishing
Outline

platform:/resource/ICSE15/demo/p2p/p2p.statemachine_mavo
- Decision1 (What happens when a download finishes?)
  - Benevolent (When done download, automatically start seeding)
  - Selfish (When done download, become idle)
  - Compromise (When done download, only seed to connected peers)
- Decision2 (Can users initiate seeding?)
  - CanShare (Yes they can)
  - CannotShare (No they can't)
- Decision3 (Can users restart completed downloads?)
  - CanRestart (Yes they can)
  - CannotRestart (No they can't)

Fig. 6 Screenshot of MU-MMINT. Left: interactive workspace showing different versions of PtPP. Middle: Graphical partial model M_{p2p} for PtPP. Model elements that reify solutions in the uncertainty tree are annotated with (M). Right: Decision tree for M_{p2p}.

In MU-MMINT, the May formula is modelled graphically using the *uncertainty tree*, shown in the right panel of Figure 6. The uncertainty tree consists of a list of *decision* elements, each of which can have any number of children representing mutually exclusive *alternative solutions*. For M_{p2p}, the tree contains the three decisions listed in Figure 1(a). For each decision, the team explicates the possible solutions. For example, the decision about the policy when a download completes involves selecting among three alternative solutions, described in Section 2: "benevolent", "selfish", and "compromise".

Assuming that the uncertainty tree of a given partial model P has k decisions $\{D_1, ..., D_k\}$, that a given decision P_x has n alternative solutions $\{A_1^{D_x}, ..., A_n^{D_x}\}$ and that a given alternative solution $A_y^{D_x}$ has l model elements, the May formula ϕ_P of P is: $\phi_P = \bigwedge_{x=1}^{k} \phi_{D_x}$, where ϕ_{D_x} =Choose($\phi_{A_1^{D_x}}, \ldots, \phi_{A_n^{D_x}}$), where Choose is a boolean function that returns True if exactly one of its arguments is True. In turn $\phi_{A_y^{D_x}} = \bigwedge_{z=1}^{l} u_z$, where $u_z = e_z$ if $e_z \in A_y^{D_x}$ and $u_z = \neg e_z$ if $e_z \in A_w^{D_x} - A_y^{D_x}$ for $w \neq y$ and $e \in P$. An example of this construction is given in Appendix B.2.

The middle panel shows the graphical part of the M_{p2p} partial model. It consists of a diagram expressed in the language of partialized state machines, shown in Figure 1(c), that includes Maybe-annotated elements. These elements reify the various alternative solutions and are included in the final version of the model only if their respective solution is selected. For example, the state `Finishing` and its associated transitions are annotated with (M) and [`Compromise`] to indicate that they are part of that particular solution to the policy decision.

To further support the articulation process, MU-MMINT supports highlighting the elements reifying a

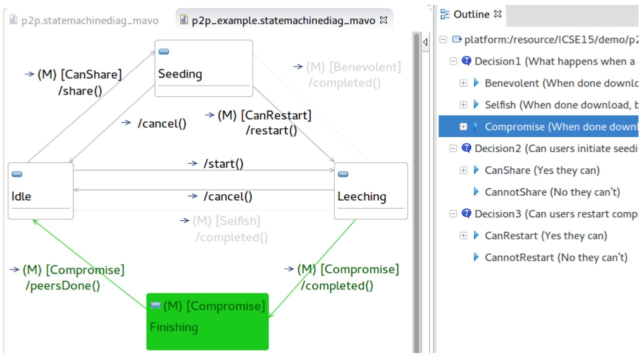

p2p.statemachinediag_mavo
p2p_example.statemachinediag_mavo
(M) [CanShare] /share()
Seeding
(M) [Benevolent] /completed()
/cancel()
(M) [CanRestart] /restart()
/start()
Idle
/cancel()
Leeching
(M) [Selfish] /completed()
(M) [Compromise] /peersDone()
(M) [Compromise] /completed()
(M) [Compromise] Finishing
Outline
platform:/resource/ICSE15/demo/p2
Decision1 (What happens when a
Benevolent (When done downlo
Selfish (When done download, b
Compromise (When done down
Decision2 (Can users initiate seedi
CanShare (Yes they can)
CannotShare (No they can't)
Decision3 (Can users restart comp
CanRestart (Yes they can)
CannotRestart (No they can't)

Fig. 7 Highlighting the elements that reify the "compromise" alternative solution in MU-MMINT.

particular solution, as shown in Figure 7. MU-MMINT can also highlight the alternative resolutions of a decision point, using different colours, as shown in Figure 8. These features allow developers to quickly examine the various possibilities in the partial model.

Deferral stage. MU-MMINT implements the *Verify* operator that allows users to check syntactic properties, such as the property P_2 ("no two transitions have the same source and target") in the PtPP example. If the result of property checking in Maybe or False, MU-MMINT allows users to invoke the *GenerateCounterExample* operator. For example, Figure 9 shows how MU-MMINT generates a concretization of M_{p2p} that is a counterexample for P_2. MU-MMINT contextualizes the counterexample with respect to the original partial model by greying out unused Maybe elements.

MU-MMINT also implements the *Transform* operator, using the Henshin graph transformation language and engine [?] to implement lifting as described by Famelis et al. [?]. However, support for the *Transform*

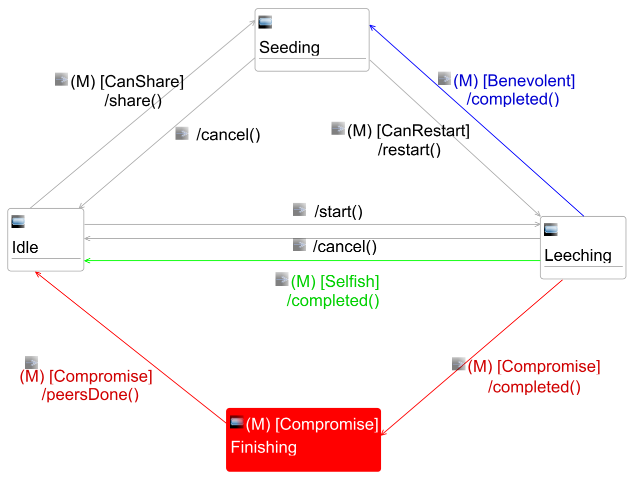


Fig. 8 Highlighting alternatives for the decision "What happens when a download is completed?" in Mu-Mmint. Maybe elements realizing the "benevolent" scenario are coloured in blue, those realizing the "selfish" scenario – in green, and those realizing the "compromise" scenario – in red.

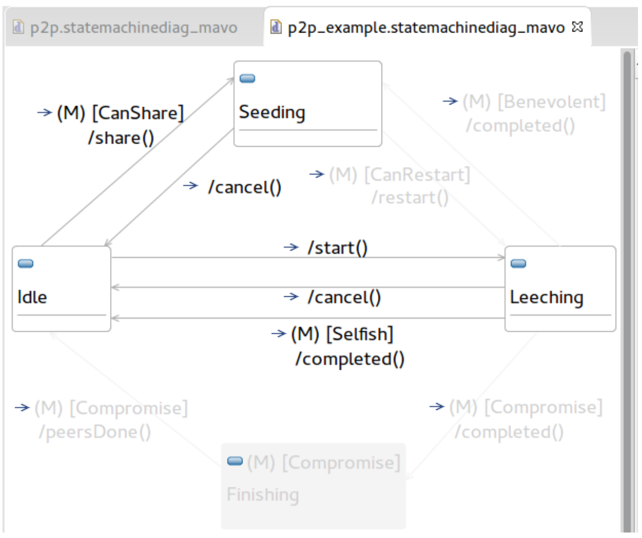


Fig. 9 Visualizing a counterexample to property P_2 in Mu-Mmint.

operator in Mu-Mmint is limited, due to the limited expressiveness of the uncertainty tree. Specifically, lifted transformations do not necessarily produce partial models whose May formulas are expressible as uncertainty trees. Therefore in Mu-Mmint, the May formula of partial models created as output by lifted transformations is not shown to users graphically. Instead, it is stored as a raw SMT-LIB [?] string in the workspace megamodel.

Resolution stage. In addition to the diagnostic operator *GenerateCounterExample* described in the previous paragraph, Mu-Mmint implements the *Decide* and *Constrain* operators.

Modellers can invoke the *Constrain* operator to restrict the possible concretizations of the partial model by enforcing a property of interest thus eliminating some design alternatives. Mu-Mmint then puts the partial model in Propositional Reduced Form (PRF), automatically recalculating the uncertainty tree and updating the diagrammatic part of the partial model.

Once the modeller has enough information to make a decision, she can invoke the *Decide* operator by selecting the desired alternative solution in the uncertainty tree. Specifically, Maybe elements that reify her chosen solution are kept, and turned into regular model elements, while Maybe elements reifying alternative solutions are removed from the model. Additionally, Mu-Mmint removes the resolved decision from the uncertainty tree.

Mu-Mmint also maintains full traceability between different versions of the partial model in its workspace, as shown in the left panel in Figure 6. This allows modellers to easily undo uncertainty resolutions in case they want to revisit certain design decisions (cf. the transition *undoResolution* in the DeTUM model).

Mu-Mmint was developed in Java by extending MMINT, an interactive environment for model management [?] developed at the University of Toronto [?]. MMINT consists of 140 KLOC, 80% of which is automatically generated. Mu-Mmint is implemented by an additional 10 KLOC, 60% of which is generated. MMINT uses the Eclipse Modelling Framework (EMF) [?] to express models and the Eclipse Graphical modelling Framework (GMF) [?] for creating graphical editors. Mu-Mmint extends MMINT's data model with EMF structures for uncertainty-related constructs. It also hooks specialized GMF diagram elements and views to represent partial models. Mu-Mmint uses the Z3 SMT solver [?] for performing reasoning tasks, and adapts parts of the Henshin [?] engine for lifted graph transformations.

The main limitation of Mu-Mmint is the expressiveness of the visual syntax used to represent uncertainty in partial models. Our original intent was to realize the MAV-Vis graphical syntax [?] which we created for design-time uncertainty using the theory of visual notations developed by D. Moody [?]. This was not possible because of our reliance on GMF. While GMF allows easy integration of existing model editors to Mu-Mmint, it only supports a limited visual vocabulary. To overcome this, we introduced the concept of the uncertainty tree, as shown in the right panel of Figure 6. This approach attempts to ameliorate the limitations of GMF by using a separate dialog, exclusively dedicated to modelling uncertainty at a higher level of abstraction. This idea followed from a preliminary empirical study with human participants [?] that pointed us to the need to elevate decisions and alternative solutions to first class concepts in partial modelling. The uncertainty tree approach is less expressive than the fully fledged propositional logic, which results in problems in supporting the *Transform* operator. However, since it has been shown that humans can graphically create and manipulate many useful formulas [?], this represents a trade-off between usability and expressive power. Resolving this trade-off is ultimately dependent on the usage-specific requirements of the context in which Mu-Mmint is deployed.

6 Evaluation

To evaluate and further illustrate our approach, we present two non-trivial worked examples. We recreate two realistic uncertainty management scenarios using non-trivial, publicly available modelling artifacts. Each worked example is elaborated using the Mu-Mmint implementation of the DeTUM model. At the end of each one, we discuss the main lessons learned, with pointers to future work.

6.1 UMLet Bug #10

On March 8th, 2011, a software developer under the alias "AFDiaX" submitted a bug report to the issue tracker of UMLet, an open source Java-based UML drawing tool [?]. The bug report, originally posted on Google Code and since migrated to GitHub as Bug #10[2], stated:

```
copied items should have a higher z-order priority

What steps will reproduce the problem?
1. Copy an item (per double-click)
2. Click on the area where the original
   and the copy are overlapping
3. Move the mouse

What is the expected output? What do you see instead?
Expected: The new copy should be moved
Instead: The original item is moved

Type-Enhancement Component-UI OpSys-All
Priority-Low Usability
```

That is, if the user copies and then pastes an item within the editor at a location where it overlaps with other existing items, the system does not recognize it as the topmost item, i.e., it does not give it "z-order priority".

In this section, we use this real world bug to illustrate explicit uncertainty management. Specifically, we create a fictional but realistic scenario in which the maintainer of UMLet attempts to create a fix for bug #10. In our scenario, the maintainer is a practitioner of model-driven software development that uses models as the main artifact for development, relying on code generation to derive the Java implementation.

In order to fix the bug, the maintainer modifies the UML Sequence Diagram modelling the behaviour of the system. However, soon after she realizes that her fix created additional problems because she modified the sequence diagram without properly synchronizing it with the structural aspects (e.g., classes) of the system. This causes her model to violate certain constraints required by the code generator. In order to resolve these constraint violations, she uses an automated technique that generates alternative model repairs [?].

In our scenario, uncertainty arises when the maintainer is unsure about which of these subsequent repairs to choose because their relative merits are unclear. She would thus like to reason with the set of alternative repairs to help her make the choice and possibly even defer the decision until more information is available. In the rest of this section, we show how uncertainty management can be deployed to help the maintainer, illustrating the use of partial modelling techniques throughout the different stages of the DeTUM model.

Description of bug #10 and the maintainer's bug fix. In the version of UMLet that was current at the time that the bug was reported[3], the paste functionality was implemented by instantiating the class `Paste` and invoking its `execute` operation. Figure 10 shows a fragment of the sequence diagram, generated from the code using the Borland TogetherJ tool[4]. The fragment shows `execute` with the circled portion representing the fictional bug fix that the maintainer creates.

Although UMLet has 214 classes in total, we restrict ourselves to a slice that is relevant to the `Paste` class consisting of 6 classes (in addition to Java library classes). These have a total of 44 operations, out of which 13 are invoked in `Paste`. The relevant slice of the UMLet model is captured by model K0, shown in the Appendix Figure 14. K0 consists of 63 EMF [?] model elements, out of which 43 are EMF references.

In Figure 10, the `for` loop statement block iterates through every item in the clipboard, indexed by variable `e`. First, each item's (`x`, `y`) coordinates in the editor window are identified (messages 1.36-1.38). The item is

[2] Bug #10: https://github.com/umlet/umlet/issues/10, URL accessed on 2015-10-22.

[3] Revision 59 on Google Code, since then migrated to GitHub and available at: https://github.com/umlet/umlet/commit/f708f57a1fbf98b3b083e583761e9887ea717ef3, URL accessed on 2015-10-22.

[4] http://www.borland.com/us/products/together/, URL accessed 2011-09-30

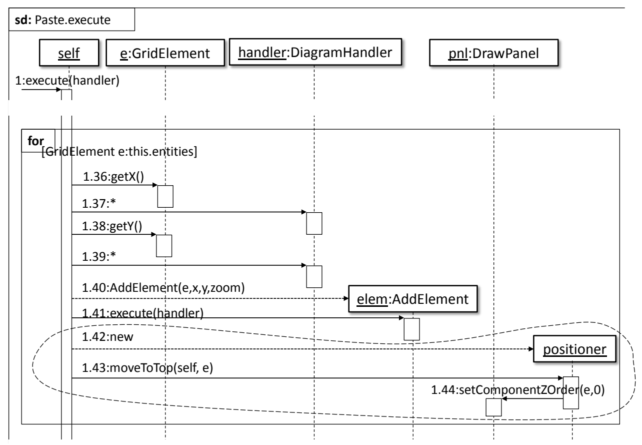


Fig. 10 Sequence diagram fragment of the UMLet paste function, depicting the `execute` operation. The maintainer's fix for bug #10 is encircled by a dashed line.

then added as an element to the editor window, represented by the object `pnl`, at the coordinates $(\texttt{x}, \texttt{y})$ of the drawing plane (messages 1.40-1.41).

The bug is caused because when an item is added to a `DrawPanel`, its order in the stack of other items at position $(\texttt{x}, \texttt{y})$, i.e., its "z-order", is not set to 0 by default. In our scenario, the maintainer fixes the bug by creating a transient object `positioner` (message 1.42). The `positioner` has a method `moveToTop(e)` that is invoked to place the item on top of others, using the library operation `setComponentZOrder` from the Swing graphical framework (messages 1.43-1.44). In the diagram, the bug fix is shown encircled by a dashed line.

6.1.1 Articulation stage

The fix to bug #10 created by the maintainer is conceptually correct but it violates two consistency rules, defined by Van Der Straeten et al. [?], that are required for code generation:

– **ClasslessInstance**: Every object must have a class.
– **DanglingOperation**: The operation used by a message in a sequence diagram must be an operation of the class of the receiving object.

In the maintainer's bug fix, shown in Figure 10, the `positioner` object violates **ClasslessInstance** because it is not associated with any class. Additionally, the message 1.43 in which the operation `moveToTop` is invoked violates **DanglingOperation** because it is not in `positioner`'s class (since `positioner` has no class).

In order to resolve these consistency violations, the maintainer uses an automated technique that generates alternative model repairs [?]. The technique proposes the following repair strategies for **ClasslessInstance**:

– *RC1:* Remove the object.
– *RC2(obj):* Replace the object with an existing object `obj` that has a class.
– *RC3(cls):* Assign the object to the existing class `cls`.
– *RC4:* Assign the object to a new class.

For **DanglingOperation**, it proposes the following repair strategies:

– *RD1:* Put the operation into the receiving object's class.
– *RD2(op):* Change the operation to the operation `op` that is already in the receiving object's class.
– *RD3:* Remove the message.

Since the strategy RC1 deletes the object it can only be combined with the strategy RD3, that also deletes the message.

Applying these repair strategies to UMLet results in a set of alternative repairs. Specifically:

– The object `positioner` can be removed (RC1), can be replaced by one of the 5 existing objects (RC2), can be designated as a separate instance of one of the existing 5 classes (RC3), or can be an instance of an altogether new class (RC4).
– The operation `moveToTop` can be removed (RD3), or if `positioner` is assigned a class, it can be either added it it (RD1) or it can be swapped for one of the existing 10 operations, depending on which class `positioner` was assigned to (RD2).

In total, there are 44 possible valid repair combinations, listed in the Appendix Table 13.

Using this list, the maintainer is able to express her uncertainty using the Articulation operator *MakePartial*. Specifically, she uses MU-MMINT to edit the UMLet model K0 and create a partial model K1. The diagram of K1 is shown in the Appendix Figure 15 and consists of 115 EMF model elements, 52 of which are annotated with Maybe. In K1 all the model elements of the maintainer's fix (shown encircled by a dashed line in Figure 10) become Maybe since they are present in some alternatives and absent in others. The May formula of K1 is modelled as an uncertainty tree two decision points, one for each consistency constraint, that have four and three alternative solutions respectively, as described earlier. As a result, K1 has 44 concretizations, each corresponding to a valid repair combination.

6.1.2 Deferral stage

Having constructed the partial model K1, the maintainer can defer the selection of one of the possible bug fixes until she has sufficient information about them.

In the meantime, she is able to perform other tasks, without having to artificially remove uncertainty. In our scenario, the maintainer wishes to reassure the users of UMLet that the changes introduced by her bug fix do not affect existing functionality. In particular, she focuses on the behaviour of the `Paste` command, wishing to show that the property U1 holds, regardless of the details about how bug #10 was fixed:

U1: *Whenever an item is pasted, a new item is created in the editor window.*

For her model to satisfy property U1, it must have: (a) the message `paste` from the `Paste` class to the `ClipBoard` class to obtain the pasted `GridElement e`, (b) the message `cloneFromMe` to `e` in order to create a new `GridElement` copy to be pasted, and finally (c) the instantiation of a new `AddElement` command object to add the item to the editor window. The maintainer uses the operator *Verify* to show that this is the case. To do this, she first encodes the property U1 in logic, creating the formula ϕ_{U1}. The grounded version of ϕ_{U1}, expressed in SMT-LIB [?] is shown in the Appendix Figure 18. She then invokes the *Verify* operator with K1 and ϕ_{U1} as inputs. In Mu-Mmint, the invocation of *Verify* follows a specialized decision procedure for the verification of properties of partial models [?], using the encoding shown in the Appendix Figure 17 and the Z3 SMT solver [?]. In our scenario, the *Verify* operator returns True, indicating that all concretizations of K1 satisfy the property. This is reasonable, since neither the bug fix, nor the automatically generated consistency repairs affected that part of the model.

Since checking property U1 yielded True, the maintainer is able to reassure her users that the alternative bug fixes do not break the paste functionality. Having determined that U1 is not a factor in deciding how to resolve uncertainty, she now returns her attention to the property U2:

U2: *Each item that is pasted from the clipboard must have z-order=0.*

It was the violation of this property that originally caused bug #10.

In her model, ensuring that pasted elements are assigned the correct z-order is done by invoking the method `setComponentZOrder` of the class `DrawPanel` with the item `e` and the z-order 0 as parameters. Again, the maintainer invokes the operator *Verify* to check that this is the case. She encodes U2 in logic, creating the formula ϕ_{U2}, the grounded version of which is shown in the Appendix Figure 16. In this case the operator *Verify* yields the result Maybe, indicating that some but not all of the concretizations satisfy the property.

6.1.3 Resolution stage

The maintainer is alarmed by the verification result since it indicates that not all concretizations of K1 actually fix bug #10. To understand why that is the case, she uses the operator *GenerateCounterExample* with K1 and U2 as inputs. The operator finds the concretization K2, shown in the Appendix Figure 19, which is the result of applying the consistency repairs RC1 and RD3, i.e., deleting the model elements the maintainer added to K0 to fix the bug in the first place.

To ensure that bug #10 is fixed, the maintainer must refine her partial model, i.e., reduce its set of concretizations to the subset that satisfies U2. To accomplish this she invokes the operator *Constrain* with K1 and U2 as inputs. In the resulting partial model K3, the diagram of which is shown in the Appendix Figure 20, the `positioner`, as well as the messages `moveToTop`, `setComponentZOrder`, and `new` (which instantiates the `positioner`) are no longer annotated with Maybe. This means that they must exist in each concretization of K3. However, all the edges that have one of these elements as their source are annotated with Maybe. For example, the `positioner` object has 6 outgoing class reference edges, all of which are annotated with Maybe. This is because the decision about what class this object is an instance of has yet to be made. The allowable combinations of Maybe elements are captured in the May formula ϕ_{K3}. Specifically, ϕ_{K3} encodes the repair combinations 2 through 44 from the Appendix Table 13.

At this point, the maintainer has created a partial model that encodes a set of possible models, all of which are both valid fixes to bug #10 and consistent with the constraints imposed by the code generator. She can choose to again defer making a decision, thus entering a second Deferral stage, or to further resolve uncertainty if she feels she has adequate information to do so. In our scenario, she chooses the combination of repair options RC4, RD2, which assigns `positioner` to a new class `NewClass` that also has the method `moveToTop`. To accomplish this, she invokes the operator `Decide`, thus creating the concrete model K4, shown in the Appendix Figure 21.

6.1.4 Summary and lessons learned

Summary. We summarize the UMLet example in Figure 11, which superimposes the models and invocations of uncertainty management operators in the scenario over the DeTUM model.

At the start of the scenario, the maintainer of UMLet received a bug report saying that pasted elements in the UMLet editor do not have the correct z-order. To

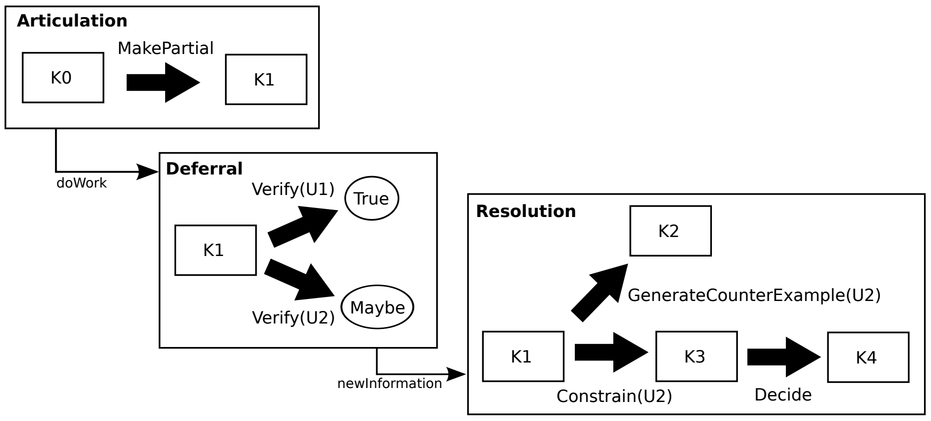


Fig. 11 Overview of the UMLet example, superimposed over the DeTUM model.

fix the bug, she identified a slice of UMLet that contained the bug, represented by the model K0. In fixing the bug, she realized that her changes caused two consistency constraints required by the code generator to be violated. She used an automated technique for generating repairs to these constraints, which resulted in a set of 44 alternative ways to fix the bug. Not having enough information to choose between them, she entered the uncertainty Articulation stage and created the partial model K1 using the operator *MakePartial* on K0. Having expressed her uncertainty, she entered the Deferral stage, where she used the *Verify* operator to check properties of her partial model.

In the process, she found out that K1 contained concretizations which did not really fix the original bug. She proceeded to remove them from her model, thus entering the Resolution stage. She used the *GenerateCounterExample* operator to diagnose why some of the concretizations were not valid fixes. The counterexample K2 showed her that a particular combination of strategies to repair the consistency violations effectively undid her original repair. To address this, she used the operator *Constrain* to remove the offending concretizations, thus creating the partial model K3 that only contained valid fixes. Finally, she chose a particular combination of consistency repairs and used the *Decide* operator to resolve all uncertainty in her models, creating the model K4.

Lessons learned. The UMLet example allowed us to work a realistic, non-trivial example through the stages of the DeTUM model. Two main lessons emerged:

1. Articulation of uncertainty requires additional automated support. During the Articulation stage, we had to express a large combination of possible repairs as a partial model. Even with the tooling support provided by Mu-Mmint, the manual construction of the partial model K1 using the *MakePartial* operator was tedious and error-prone. In order for the effort expended in constructing a partial model to be outweighed by the benefits, we need to create additional automated support. This could be achieved using techniques such as Design Space Exploration [?], by integrating partial modelling into sketching tools [?,?], or by mining the social context of development, such as online discussions between developers [?].
2. Rigid separation of verification from diagnostic operators is counter-intuitive. In the DeTUM model, the *Verify* operator is placed in the Deferral stage, whereas diagnostic operators, such as *GenerateCounterExample*, are placed in the Resolution stage. There are good reasons for this: during Resolution uncertainty is reduced, whereas during Deferral its level remains constant. However, since in practice verification and diagnosis are closely intertwined, it is important to emphasize that the transition between of stages is not as rigid, even when taking into account the backward transitions in Figure 4.

6.2 Petri Net Metamodel

In this section, we take the view of a toolsmith who is tasked with creating a fictional tool, called ConcMod, for modelling concurrent systems. ConcMod uses Petri nets (PTNs), a powerful formalism used widely in this domain, first introduced by Carl Petri in 1962 [?]. A succinct description of PTNs is given by Jensen and Kristensen [?]:

"A *Petri net* in its basic form is called a *place / transition net*, or PTN, and is a directed bipartite graph with nodes consisting of *places* (drawn as ellipses) and *transitions* (drawn as rectangles).

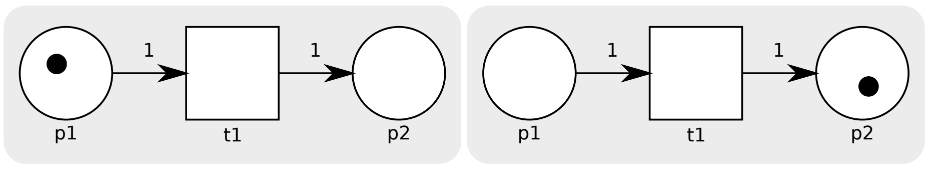


Fig. 12 Example Petri net token game. Left: `Transition` t1 is enabled. Right: the Petri net after firing t1.

The state of a Petri net is called a *marking* and consists of a distribution of *tokens* (drawn as black dots) positioned on the places. The execution of a Petri net (also called the "token game") consists of occurrences of *enabled* transitions removing tokens from input places and adding tokens to output places, as described by integer *arc weights*, thereby changing the current state (marking) of the Petri net. An abundance of structural analysis methods (such as invariants and net reductions), as well as dynamic analysis methods (such as state spaces and coverability graphs), exist for Petri nets."

An example PTN, consisting of two `Place` s and one `Transition` is shown in Figure 12. In the original marking, shown on the left, the `Place p1` contains one `Token`. The `Transition t1` has a single incoming arc, and since that arc has weight 1, and `p1` has one `Token`, `t1` is enabled. When `t1` fires, the Petri net gets the marking shown on the right. In this new state, the `Token` in `p1` has been consumed and because `t1` has a single outgoing arc with weight 1, a single token has been produced in `Place p2`. Since `p1` is empty, `t1` is no longer enabled.

In our scenario, we assume that during the initial design phase of the ConcMod project, the toolsmith wants to create a metamodel for representing PTNs such as the one shown in Figure 12. In this section, we use the development of this metemodel as an example for explicit uncertainty management.

6.2.1 Base PTN metamodel

Articulation stage. In order to avoid re-inventing a PTN metamodel from scratch, the toolsmith consults a public metamodel repository called Atlantic Metamodel Zoo (AMZ) [?]. In AMZ, she discovers eight different PTN metamodels.[5] While all metamodels have some basic PTN concepts in common, such as meta-classes for `Place` s and `Transition` s, they are different in a few significant ways. By inspecting the metamodels, the toolsmith identifies the following design decisions that cause them to diverge:

- **ArcClasses**: Arcs are represented using separate meta-classes.
- **WeighedArcs**: Arc meta-classes have attributes to represent arc weight.
- **Locations**: There is a way to store the location of graphical elements on the diagram.
- **TokenClass**: Tokens are represented using a separate meta-class.
- **Executions**: If a separate token meta-class exists, there is a mechanism for representing "token games".

Table 2 summarizes what decisions are implemented in each of the eight PTN metamodels in AMZ.[6] The case where a metamodel implements a decision is indicated by "✓" and the case where it does not — by "✗". Metamodels that implement the same decisions differ in minor ways. Specifically, `GWPNV1` differs from `GWPNV0` by requiring that a PTN must have at least one `Place` and `Transition`, `GWPNV3` from `GWPNV2` — by introducing a superclass for arcs, and `PetriNet` differs from `GWPNV4` by introducing a class `Element`. In our scenario, the toolsmith decides that these differences are not significant enough for her purposes, and thus chooses to disregard them.

Since each decision is binary and two decisions are dependent on others (*WeighedArcs* depends on *ArcClasses*, *Executions* depends on *TokenClass*), the toolsmith can create $(2 + 1) \times 2 \times (2 + 1) = 18$ different combinations. However, she does not have enough information to decide which combination is best for ConcMod. She thus uses the *Construct* operator to create the partial metamodel N0, the diagram of which is shown in the Appendix Figure 22. The partial model N0 has a total of 76 elements (52 node and 24 vertex elements), out of which 60 are annotated with Maybe. N0 has 18 concretizations, defined by its May formula, the construction of which is described in Appendix B.2.

Deferral Stage: Transformation. Having explicated her uncertainty in the partial model N0, the toolsmith can

[5] Atlantic Metamodel Zoo: http://www.emn.fr/z-info/atlanmod/index.php/ZooFederation, URL accessed 2015-11-04.

[6] The metamodels with the prefix `GWPN` were created by Guido Wachsmut and Kelly Garces. The metamodel `Extended` was created by Hugo Bruneliere and Pierrick Guyard. The metamodel `PetriNet` was created by David Touzet.

Table 2 Design decisions in Petri net metamodels from the Atlantic Metamodel Zoo. Each column represents one metamodel.

	`GWPNV0`	`GWPNV1`	`GWPNV2`	`GWPNV3`	`GWPNV4`	`GWPNV5`	`Extended`	`PetriNet`
ArcClasses	✗	✗	✓	✓	✓	✓	✓	✓
WeighedArcs	✗	✗	✗	✗	✓	✓	✓	✓
Locations	✗	✗	✗	✗	✗	✗	✓	✗
TokenClass	✗	✗	✗	✗	✗	✓	✓	✗
Executions	✗	✗	✗	✗	✗	✗	✓	✗

defer making a decision about how exactly to design the ConcMod metamodel. In the meantime, she can use N0 to proceed with other development tasks. In our scenario, the focus of the toolsmith is turned to the serialization of PTN models. In order to efficiently store PTN models, the toolsmith wants to use a relational database. She thus needs to create a database schema that allows her to store instances of PTN models.

Mapping the PTN metamodel to a relational database schema is an instance of the well-known "Object Relational Mapping" (ORM) problem. In model-driven software engineering, a typical solution to the ORM problem involves creating a model transformation that takes a class diagram as input and produces a relational schema model as output. In our scenario, the toolsmith uses an ORM transformation created by Varro et al. [?], and adapted by Famelis et al. [?], that consists of five layered rewrite rules that, given a class diagram, create a relational schema and traceability links between them. The rules and the input/output metamodels are shown in Appendix C.

The toolsmith invokes the uncertainty management operator *Transform* with N0 and the ORM transformation as inputs. The result is the partial relational schema model N1, consisting of 30 tables. Overall, N1 has 192 elements (tables, columns, and key references), out of which 184 are `Maybe`. The runtime of the transformation was 2.00 seconds and the generated SMT-LIB May formula has the size of 23.26 kB.

Deferral Stage: Verification. The toolsmith continues working in the presence of uncertainty. Having created a partial relational database schema, she decides to investigate whether her schema allows her to accurately store the graphical diagrams of PTN models. She formalizes this requirement as a property:

U3: *The database must allow storing the diagram coordinates of every graphical PTN construct.*

For her database schema to satisfy U3, for each table T_i in N1 that stores some graphical PTN construct (`Place`, `Transition`, `Token`, `PlaceToTransitionArc`, `TransitionToPlaceArc`) there must exist a table TL_i that maps tuples from the table T_i to tuples from the table `Location`. The toolsmith uses the operator *Verify* to check whether N1 satisfies U3. To do this, she first encodes the property U3 in logic, creating the formula ϕ_{U3}. The grounded version of ϕ_{U3}, expressed in SMT-LIB [?], is shown in the Appendix Figure 24. She then invokes the *Verify* operator with N1 and ϕ_{U3} as inputs. The result of the property check is `Maybe`, indicating that U3 may not be satisfied, depending on how uncertainty is resolved.

Resolution stage. In order to discover the reason why checking U3 on N1 results in `Maybe`, the toolsmith uses the diagnostic operators. Using the operator *GenerateCounterExample*, she discovers that since the `Location` table is annotated with `Maybe`, there exists at least one concretization in which it does not exist. Since her primary development artifact is the partial metamodel N0, she uses this new information to refine N0 using the operator *Decide* to ensure that the class `Location` is present in all concretizations. The result is the partial metamodel N2, shown in the Appendix Figure 25.

Additionally, by invoking the operator *GenerateDiagnosticCore* with N1 and U3 as inputs, she discovers that there are no tables mapping tuples of the tables `PlaceToTransitionArc`, `TransitionToPlaceArc`, and `Token` to the table `Location`. She decides that this is acceptable, since the location of these graphical elements can be derived from `Place` s and the combination of endpoint `Place` and `Transition`, respectively.

The toolsmith does not have any more information to fully resolve the uncertainty in N2. Additionally, she receives new requirements for ConcMod, which cause her more uncertainty, prompting her to enter a new uncertainty Articulation stage.

6.2.2 Extended PTN metamodel

Articulation stage. Due to its simplicity, the Petri net formalism lends itself to customization in order to capture domain-specific concerns. This has lead to the proliferation of Petri net extensions, which augment the base language with special-purpose constructs. Examples of such extensions include Coloured PTNs [?], Hierarchical PTNs [?], Prioritized PTNs [?], Timed PTNs [?], Stochastic PTNs [?], and others. In our scenario, the toolsmith learns that her employer is considering allowing such PTN models to be stored in ConcMod.

However, she does not have enough information about which domain-specific constructs should be included in the ConcMod metamodel.

Faced with uncertainty about which Petri net extensions to support, the toolsmith uses the operator *Expand* on the partial metamodel N2, to create the partial metamodel N3. The uncertainty tree of N3, shown in the Appendix Figure 27, contains several new (binary) design decision points:

- **ArcKinds**: There are different kinds of arcs, such as inhibitor and reading arcs [?].
- **Priority**: Transitions have priorities [?].
- **Timed**: Transitions are timed [?].
- **ColouredTokens**: Tokens have values ("colours") taken from "Colour Sets", i.e., types [?].
- **Stochastic**: Each transitions has a "firing rate" that indicates the probability that it will fire at every marking [?].
- **Guards**: Arcs have guard expressions [?].

The decisions *Guards*, *WeighedArcs*, and *ArcKinds* depend on the decision *ArcClasses*, while the decisions *Executions* and *ColouredTokens* depend on the decision *TokenClass*. Therefore, the uncertainty tree allows $(1+2\times2\times2)\times(1+2\times2)\times2\times2\times2 = 360$ concretizations. The model elements required to implement these design decisions bring the total elements of N3 to 117 elements, out of which 94 are Maybe. The diagram of N3 is shown in the Appendix Figure 26.

Deferral stage. Having explicated her uncertainty in the partial model N3, the toolsmith again generates a relational database schema using the ORM transformation. She invokes the uncertainty management operator *Transform* again, this time with N3 and the ORM transformation as inputs. The result is the partial relational schema model N4, which has 45 tables. In total, N4 has 293 elements (tables, columns, and key references), out of which 258 are Maybe. The total runtime of the transformation was 114.05 seconds and the generated SMT-LIB May formula is 33.78 kB long.

Resolution stage. Using lifted operations, the toolsmith can continue working in the presence of uncertainty for as long as necessary. In our scenario, we assume that at some point the toolsmith is able to resolve all of her uncertainty by making all of the design decisions. Specifically, she uses the *Decide* operator with the partial model N3 as input and makes the choice to not implement any of the domain-specific Petri net extensions, to allow executions to be stored, and to use separate arc and token meta-classes. These decisions result in the concrete metamodel N5, shown in the Appendix Figure 28.

6.2.3 Summary and lessons learned

Summary. We summarize the PTN metamodel example in Figure 13, which superimposes the models and invocations of uncertainty management operators in the scenario over the DeTUM model.

At the start of the scenario, the toolsmith in charge of creating ConcMod aimed to create a metamodel for representing Petri nets. She consulted an open repository of metamodels, where she located eight existing different Petri net metamodels. By analysing these metamodels, she identified five important design decisions that are the cause of the differences between them. Not having enough information to decide how to make these decisions, she used the operator *Construct* to create the partial metamodel N0.

Having articulated her uncertainty in N0, the toolsmith entered the Deferral stage. She applied the ORM transformation to her partial metamodel using the operation *Transform*, to create a derived partial relational schema model N1. She then applied the operator *Verify* to check whether N1 allows storing the diagrammatic locations of Petri net elements. The result of the check was Maybe, and prompted her to perform further diagnosis using the diagnostic operators. She discovered that (a) not all concretizations have a dedicated table for storing locations, and (b) the locations of some of the diagrammatic elements can be derived from others.

The first diagnostic insight led her to enter the Resolution stage, where she used the newly acquired information to resolve some of the uncertainty in N0, to ensure that the `Location` table is present in all concretizations. To do this, she applied the operator *Decide*, resulting in the partial metamodel N2. The second insight led her to re-examine the requirement to store the locations of all elements.

At this point in our scenario, the toolsmith had to consider the possibility of creating support in ConcMod for representing various Petri net flavours, that add domain-specific modelling constructs to the base Petri net language. She thus entered a second Articulation stage, where she applied to operator *Expand* to the partial metamodel N2, resulting in the partial metamodel N3.

The toolsmith entered a second Deferral stage, where she again invoked the operator *Transform* to apply the ORM transformation to the partial model N3, resulting in the new partial relational schema model N4. Subsequently, the toolsmith was able to remove all uncertainty from her models during a second Resolution stage, where she used the operator *Decide* to reduce N3 to the concrete Petri net metamodel N5.

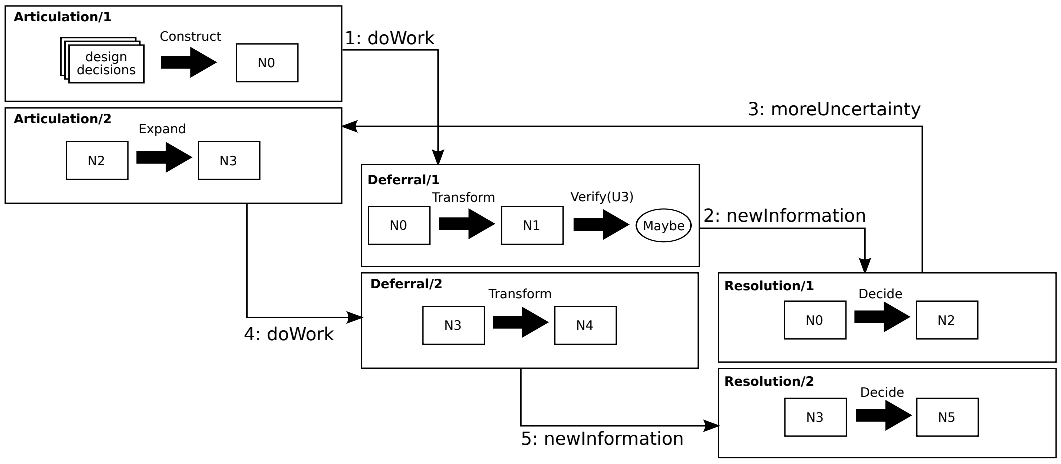


Fig. 13 Overview of the PTN metamodel example, superimposed over the DeTUM model.

Lessons learned. In the Petri net metamodel example, we worked with a medium-sized model with a large set of concretizations and a large percentage of Maybe elements through the different stages of uncertainty management. Three main lessons arose from this experience:

1. Even though we described the initial articulation of uncertainty in terms of invoking the operator *Construct* for the set of all possible metamodel designs, in practice, we had to adopt a slightly different approach. Specifically, we manually constructed the partial model N0, using the eight AMZ metamodels and the five design decisions as guidance. This was a different process than both *Construct* (which is fully automatic but depends on an enumerated set of alternatives) and *MakePartial* (which is manual but depends on a pre-existing concrete model). Rather, in practice, we ended up adopting elements of both, whereby we manually enumerated a set of possibilities, using the uncertainty tree as a guide.
2. The number of application sites and the number of Maybe elements in the partial models N0 and N3 was significantly larger than that of the models used for the ORM benchmark by Famelis et al. [?]. This made the original implementation of our lifted transformation engine in Mu-Mmint extremely inefficient. We were thus forced to revisit it and implement various optimizations that resulted in improvements in both runtime and the size of the generated May formula. These improvements were so dramatic as to render comparison with the benchmark results previously published [?] meaningless. However, during the implementation of the optimizations, we faced significant challenges in correctly engineering complex manipulations of large formulas while working with the application programming interface (API) of the Z3 solver [?]. This process was extremely error prone and hard to debug, thus exposing some of the practical difficulties of our lifting strategy, that focuses on manipulating a single large propositional expression. We contrast this with our experience implementing lifting for product lines, where individual elements are annotated with "presence conditions" [?,?]. This makes the implementation of lifting significantly easier, since instead of manipulating a single large formula, it uses smaller logical expressions that are localized to specific matching sites.
3. During the first Resolution stage, the toolsmith invoked the *Verify* operator, which returned Maybe. However, instead of using the result to constrain the partial model, we found that it was more appropriate for the toolsmith to change her expectations about the property. This points to a characterization of properties with modalities, such as "necessary" and "possible". Responses to property checks can therefore include either refining the underlying model or changing the modality of the property. In current work, we are investigating such property modalities and appropriate responses to the verification of such properties in the context of product lines with uncertainty [?].

7 Related Work

7.1 Mitigating Design-Time Uncertainty

As discussed in Section 1, mitigating design-time uncertainty entails creation of uncertainty-aware software *development methodologies*. Without a methodology like the one described by the DeTUM model, in the face of design-time uncertainty, developers are thus forced to either:

(a) avoid working on the uncertain parts of the system, delaying the decisions as long as necessary,
(b) entirely remove uncertainty by making educated guesses based on experience so that work can continue, or
(c) fork and maintain sets of alternative solutions.

We discuss these options below.

There is a considerable body of engineering research that focuses on avoiding design-time uncertainty by delaying making decisions until the most opportune moment. This is an established practice in many engineering disciplines, such as in industrial and mechanical engineering. One example is the Critical Chain Project Management approach [?] where tasks are scheduled as late as possible, while uncertainty about their completion is managed using purpose-specific “buffers” which allow new information to be systematically incorporated in the overall estimation of the project. Another well known example is the Toyota Production System (TPS) [?], which was developed in Japan during the mid-20th century with the aim to eliminate inefficiencies in the production of automotive vehicles. The TPS was a precursor to the practice of *lean manufacturing*, which in the software world inspired Lean Software Engineering [?,?], one of the tenets of the Agile methodology [?,?]. While Agile is becoming increasingly popular, it is not (to use the phrase coined by Fred Brooks [?]) a “silver bullet” that is appropriate in every organizational setting and project. This is evidenced by the proliferation of literature discussing success factors [?,?] and challenges to its adoption [?]. Moreover, while the stated goal of lean methods is to eliminate waste, i.e., under-utilization of resources, that is not necessarily the case in practice. For example, Ikonen et al. conducted a seven-week empirical study to identify sources of waste in lean development [?], focusing specifically on the Kanban method [?]. Among other sources of waste, they identified that there were delays due to some developers waiting, e.g., for the completion of tasks that were under-estimated, for clarification of requirements, or for customer validation. Thus, delaying decisions is not always the most effective way to handle design-time uncertainty.

In practice, developers often rely on experience and craftsmanship to make and keep track of *provisional* decisions that artificially remove uncertainty from their artifacts so that development can continue. This increases the risk of having to backtrack their work if new information shows that the provisional decisions were wrong. Even worse, it can mean committing too early to design decisions that cannot be reversed without significant costs, when it would be more desirable to keep many alternative options open for consideration. In fact, the skillful management of such provisionality is a defining characteristic of expert software designers [?]. Agile proposes to manage these risks by employing short iteration cycles and frequent customer feedback. As discussed earlier, Agile is not a “silver bullet”, so such practices are not always feasible. Additionally, there is the problem of keeping track of which decisions were made provisionally and which were not. This is aggravated by the preference in Agile for “working software over comprehensive documentation” [?]. Unless explicit traceability and provenance is maintained, the provisional character of a decision may be forgotten, thus implicitly turning the decision into a premature, undocumented commitment.

Given a design decision, engineers might choose to consider all alternative solutions, therefore forking the project into parallel streams. This allows them to keep their options open, as well as to potentially turn decisions into variability points, thus creating families of products that meet the needs of more than one customer [?,?]. Forking can be done in a vigorous and systematic way. For example, the “Programming by Optimization” (PbO) approach, developed by Holger Hoos [?], aims to help software developers avoid premature optimization commitments in settings where multiple algorithms can accomplish the same computational task, albeit in different ways. Design decisions and alternatives are explicitly tracked until the time when designers have enough hard evidence (obtained systematically using machine learning) regarding the optimality of each algorithm to make informed decisions. However, forking does not scale as a generic solution to mitigating design-time uncertainty due to the combinatoric explosion of possibilities in the case where multiple decision points must be managed. Additionally, since every fork must be maintained separately, every forking approach is limited by the size of the set of possible solutions to a decision point. Without any structure to describe their commonalities and differences, it is impossible to reuse results across forks. Thus, every engineering task such as verification, transformation, evolution, etc. has to be repeated for each fork, which is expensive and error-prone.

7.2 Managing the lifecycle of sets of artifacts

The management of uncertainty in the software lifecycle ultimately involves the systematic management of a representation of a set of possibilities. A similar need to manage the lifecycle of abstractions of sets of artifacts arises in related software engineering disciplines such as variability management and design space exploration.

Design space exploration (DSE) [?,?] is a technique for systematically traversing a space of design choices in order to identify functionally equivalent designs that best satisfy a given set of desired constraints. The DSE process follows a series of steps that is typically described as follows: (1) A reference model is created to represent the requirements and constraints of the desired design. (2) Using the reference model, the developer derives a set of criteria with which to evaluate designs. (3) A model of the design space is created. (4) The developer employs various algorithms to traverse the design space and generate candidate designs. Tools such as Alloy can be used to automate this step [?]. (5) Each candidate design is measured against the evaluation criteria. (6) Desirable candidate designs are chosen based on whether they optimally satisfy the evaluation criteria.

The DSE process has certain similarities with the DeTUM model: the first two steps resemble the mental process by which a developer identifies a source of uncertainty; the third step corresponds to the Articulation stage of uncertainty managament and the final step – to the Resolution stage. The intermediate stage between Articulation and Resolution differs between DSE and uncertainty management. In DSE, decisions are not deferred; rather they are algorithmically explored to produce candidate designs. However, in uncertainty management, the goal is to allow developers to *avoid* making such decisions. Thus, the focus during this stage is in creating support for lifted engineering tasks that allow developers to continue working with their artifacts without resolving uncertainty.

The goal of *variability management* [?] is to create and maintain families of products, called Software Product Lines (SPLs), that share a common set of features. In direct analogy to the DeTUM model, we can identify three stages of SPL lifecycle: Creation, Maintenance, and Configuration. During SPL creation, the aim is to develop a set of reusable assets that can be combined to produce individual products [?]. A common task during this stage is to reverse engineer a SPL from a set of existing products that share functionality, albeit in an ad-hoc way. To address this, various techniques have been developed such as feature location [?], clone detection [?] and others. During the Maintenance phase, variability-aware techniques are applied to SPLs in order to manipulate the entire set of products without having to enumerate it. Such techniques include model checking [?], type checking [?], testing [?], model transformations [?,?], and others [?,?]. During the Configuration stage, individual products are derived from the SPL to address individual customer needs. This is typically done by configuring a feature model that expresses the allowable combinations of features [?]. This can be done either at once or in stages [?].

The main methodological difference between variability management and uncertainty management is that SPLs represent a long term commitment to supporting and maintaining a product family. It is thus not meaningful to talk about a "Variability Management Model", akin to the DeTUM model: Configuration is not the ultimate endpoint of variablity management. Rather, it represents a set of tasks that developers expect to perform often during the lifetime of a SPL. In contrast, partial models are transient artifacts. The Resolution stage of the DeTUM model is an endpoint that represents the expectation that uncertainty is ultimately removed from software artifacts. Put simply, partial models are built to throw away, whereas SPLs are built to last.

8 Conclusion

We have investigated the management of design-time uncertainty from the perspective of software engineering methodology.

We have introduced the Design-Time Uncertainty Management (DeTUM) model as a high level abstraction of the lifecycle of design-time uncertainty in software artifacts. The DeTUM model consists of three stages: 1. during the Articulation stage, developers construct partial models (i.e., models that encode a set of possible designs) to express their uncertainty, 2. during the Deferral stage, the degree of uncertainty in software artifacts remains unchanged, while developers use lifted versions of existing operations to perform engineering tasks, 3. finally, during the Resolution stage, new information is incorporated to resolve uncertainty by refining partial models. Developers transition between these stages based on their engineering needs and the absence or presence of sufficient information to perform these needs.

We used the DeTUM model to contextualize various previously published techniques for manipulating partial models, such as creation, transformation, verification and refinement. We have summarized each technique as an "uncertainty management operator", and defined its place within the lifecycle of uncertainty in terms of its inputs, outputs, usage context, and conditions prior to and after their invocation. The operators are bundled in a coherent Integrated Development Environment in Mu-Mmint, an interactive Eclipse-based tool for partial model management.

In the future, we intend to further investigate the balancing of alternative approaches to tacking design-time uncertainty (deferral, forking, provisionality, and avoidance). To accomplish this, we intend to study the historical data of software systems in order to identify cases of catastrophic mismanagement of uncertainty. Subsequently, we will perform post-hoc analysis in order to determine the effectiveness of different uncertainty management strategies. In addition, we intend to study the integration of explicit uncertainty management into lean software development, such as the Scrumban methodology [?]. This will allow us to either develop mixed strategies, effectively weaving explicit uncertainty management with existing approaches, or to recognize niche contexts in which it is cost-effective.

Our approach makes two important assumptions: (a) that developers know what they are uncertain about, and (b) that all decision points are "known unknowns", i.e., closed questions where developers are uncertain about choosing one among a well understood finite set of acceptable solutions. Thus, the uncertainty management operators that we defined for the Articulation stage of the DeTUM model assume as input either a fully enumerated set of possibilities or an informal description of how uncertainty should be expressed in partial models. However, in order to use uncertainty management in the context of real software development, additional support must be provided for (a) identifying that development has reached a critical design decision such that explicit uncertainty management is cost-effective, (b) assessing the impact of uncertainty across multiple artifacts, (c) helping developers elicit a set of acceptable solutions to an open design decision (an "unknown unknown"), and (d) helping encode them in a partial model. To address the first two points, we intend to study the identification of design decisions and the assessment of their impact, by focusing on the socio-technical context in which such decisions are made. A first step in this direction is to develop a theory about when major design decisions occur in the lifecycle of software projects. A potential approach is to attempt to identify patterns among changes that had significant impact downstream, such as changes that resulted in a lot of bugs or changes that required a lot of development effort to be undone. A different approach is to study the social context in which design decisions are made. To address the latter points, we intend to investigate the use of domain space exploration techniques [?] for synthesizing sets of possibly acceptable solutions. This would entail mining the development context to identify a set of acceptability criteria and then generating designs that are equivalent with respect to these criteria. Additional research must investigate ways for presenting these recommendations to the developers in a user-friendly way.

Acknowledgements We grateful to Alessio Di Sandro, lead developer of the MMINT and Mu-Mmint tools. We also thank Rick Salay for developing the initial version of the UMLet worked example [?], on which Section 6.1 was based. Finally, we thank the anonymous reviewer #2 of the manuscript for pointing us to the work of E. Goldratt.

A Appendix: Operators

Here we give the detailed descriptions of the Uncertainty Operators from Section 4.

Table 3 Operator *MakePartial*

Description	Create a partial model from a given concrete model by introducing uncertainty to it.
Inputs	A concrete model and an informal description of uncertainty.
Outputs	A partial model.
Usage context	The developer uses the informal description of uncertainty and her intuition about how it should be expressed in the given model.
Preconditions	No partial model exists.
Postconditions	The input model is a concretization of the output partial model.
Limitations	Effectiveness depends on the intuition of the developer.
Implementation	The task is done manually using editor provided by the Mu-Mmint tool [?].

Table 4 Operator *Expand*

Description	Introduce additional uncertainty to a partial model.
Inputs	A partial model.
Outputs	A partial model.
Usage context	The developer encounters new uncertainty during the Deferral or Resolution stages of the DeTUM model.
Preconditions	Some uncertainty has already been explicated in the partial model.
Postconditions	The input partial model is a refinement of the output.
Limitations	Effectiveness depends on the intuition of the developer.
Implementation	The task is done manually using editor provided by the Mu-Mmint tool [?].

Table 5 Operator *Transform*

Description	Apply a transformation to a partial model ensuring that all concretizations are correctly transformed.
Inputs	A partial model
Outputs	A partial model
Usage context	The developer wishes each concretization to be transformed but does not wish to enumerate them all and do it for each one individually.
Preconditions	None.
Postconditions	The set of concretizations of the output partial model is exactly the same as if the input partial model had been broken down to its set of concretizations using the operator *Deconstruct*, then each concretization in that set had been transformed individually, and then a partial model had been constructed from that set using the operator *Construct*.
Limitations	The model transformation must be expressed as graph rewriting system (i.e., a set of graph rewriting transformation rules).
Implementation	The operator is realized by the lifting technique described in [?].

Table 6 Operator *Verify*

Description	Check whether a partial model satisfies a property.
Inputs	A partial model, a property.
Outputs	A value from the set {True,False,Maybe }.
Usage context	A developer is interested in determining whether a property is satisfied by some, all or none of the concretizations of the partial model.
Preconditions	None.
Postconditions	If the property is satisfied by all, none or some of the partial model's concretizations, then the output is True, False, and Maybe respectively.
Limitations	The property is syntactic, i.e., its verification does not require knowledge of the semantics of the partial model's base language.
Implementation	The operator is described in [?] as operator "OP2: Verification".

Table 7 Operator *Deconstruct*

Description	Produce the set of concretizations of a given partial model.
Inputs	A partial model
Outputs	A set of concrete models
Usage context	The developer needs to perform a task on each concretization of the input partial model and no lifted version of the task exists.
Preconditions	None
Postconditions	The output set contains exactly the set of concretizations of the input partial model.
Limitations	High cost.
Implementation	The operator can be implemented by passing the partial model's May formula to an All-Solutions SAT solver [?].

Table 8 Operator *Decide*

Description	Make decisions about whether to keep or discard individual Maybe elements from a partial model.
Inputs	A partial model and an informal description of information that resolves uncertainty.
Outputs	A (potentially singleton) partial model.
Usage context	The developer uses the informal description of newly acquired information, as well as her intuition about how it should be incorporated in the input partial model.
Preconditions	None.
Postconditions	The output partial model is a refinement of the input partial model.
Limitations	Using this operator to resolve design decisions requires explicit mapping of candidate solutions to Maybe elements.
Implementation	The editor provided by the Mu-Mmint tool allows making decisions and supports and explicit mapping of candidate solutions to Maybe elements [?].

Table 9 Operator *Constrain*

Description	Create a partial model with a subset of the concretizations of the input partial model such that all its concretizations satisfy a property of interest.
Inputs	A partial model, a property.
Outputs	A partial model.
Usage context	The developer has determined that concretizations of the input partial model that do not satisfy the input property are not valid ways to resolve uncertainty and should thus be excluded.
Preconditions	None.
Postconditions	The output partial model refines the input partial model.
Limitations	This operator is subject to the same limitations as *Verify*.
Implementation	The operator is described in [?] as operator "OP4: Property-Driven Refinement".

Table 10 Operator *GenerateCounterExample*

Description	Create a non-partial model that illustrates why a partial model does not satisfy a property.
Inputs	A partial model, a property.
Outputs	A concrete model.
Usage context	A developer wants to diagnose why a partial model does not satisfy a property, i.e., why the result of *Verify* is not True.
Preconditions	The result of the operator *Verify* using the input partial model and the input property is Maybe, or False.
Postconditions	The output model is a concretization of the input partial model and satisfies the input property.
Limitations	This operator is subject to the same limitations as *Verify*.
Implementation	The operator is described in [?] as operator "OP3a: Diagnosis - Return one counter-example".

Table 11 Operator *GenerateExample*

Description	Create a non-partial model that illustrates why a partial model may satisfy a property.
Inputs	A partial model, a property.
Outputs	A concrete model.
Usage context	A developer wants to diagnose why a partial model may satisfy a property, i.e., why the result of *Verify* is not False.
Preconditions	The result of the operator *Verify* using the input partial model and the input property is Maybe, or True.
Postconditions	The output model is a concretization of the input partial model and satisfies the input property.
Limitations	This operator is subject to the same limitations as *Verify*.
Implementation	The operator is described in [?] as operator "OP3b: Diagnosis - Return a concretization where the property does hold".

Table 12 Operator *GenerateDiagnosticCore*

Description	Create a partial model that illustrates why a partial model does not satisfy a property.
Inputs	A partial model, a property.
Outputs	A partial model.
Usage context	A developer wants to diagnose why a partial model does not satisfy a property, i.e., why the result of *Verify* is not True.
Preconditions	The result of the operator *Verify* using the input partial model and the input property is Maybe, or False.
Postconditions	The output partial model refines (or is equivalent to) the input partial model. The property is not satisfied by any of its concretizations.
Limitations	This operator is subject to the same limitations as *Verify*.
Implementation	The operator is described in [?] as operator "OP3b: Diagnosis - Return a partial model representing the set of all concretizations for which the property does not hold".

B Appendix: Models

In this appendix, we provide additional details about the worked examples in Sections 6.1 and 6.2.

B.1 UMLet Bug #10

Figure 14 shows the MU-MMINT model K0 that encodes the UMLet model shown in Figure 10. The encoded model contains the relevant slices of both the class diagram and the sequence diagram (bottom), as well as traceability links between them, linking messages in the sequence diagram to operations in the class diagram and objects to their classes. In the sequence diagram model, objects and lifelines are represented by the same model element. MU-MMINT uses yellow and pink stars to respectively indicate edges that represent the source and target lifelines of messages.

Table 13 shows the valid combinations of strategies for repairing the **ClasslessInstance** and **DanglingOperation** consistency violations. Each combination represent a repair of the model shown in 14, i.e., a concretization of the partial model K1, shown in Figure 15

Table 13 Valid combinations of repair strategies for ClasslessInstance and DanglingOperation.

#id	ClasslessInstance	DanglingOperation
1	RC1	RD3
2	RC2(self)	RD1
3		RD2(Paste)
4		RD3
5	RC2(e)	RD1
6		RD2(GridElement)
7		RD2(getX)
8		RD2(getY)
9		RD3
10	RC2(handler)	RD1
11		RD2(DiagramHandler)
12		RD2(mult)
13		RD3
14	RC2(pnl)	RD1
15		RD2(DrawPanel)
16		RD2(setComponentZOrder)
17		RD3
18	RC2(AddElement)	RD1
19		RD2(AddElement)
20		RD2(execute)
21		RD3
22	RC3(Paste)	RD1
23		RD2(Paste)
24		RD3
25	RC3(GridElement)	RD1
26		RD2(GridElement)
27		RD2(getX)
28		RD2(getY)
29		RD3
30	RC3(DiagramHandler)	RD1
31		RD2(DiagramHandler)
32		RD2(mult)
33		RD3
34	RC3(DrawPanel)	RD1
35		RD2(DrawPanel)
36		RD2(setComponentZOrder)
37		RD3
38	RC3(AddElement)	RD1
39		RD2(AddElement)
40		RD2(execute)
41		RD3
42	RC4	RD1
43		RD2(NewClass)
44		RD3

Figure 15 shows the diagram of the partial model K1 created by the maintainer to express her uncertainty about which of the 44 alternative repairs in Table 13 to select. To enhance diagram readability, we have hidden the labels of the arrow model elements. This has resulted in hiding Maybe annotations as well; however, it is easy to deduce that edges that are in K1 but not in K0 have Maybe annotations.

Figure 16 shows the ground propositional encoding ϕ_{U1} of the property U1, expressed in SMT-LIB [?]. The formula encodes the property as a conjunction of the variables (see [?]) of the messages that are required to perform pasting.

Figure 17 shows the ground propositional encoding in SMT-LIB of the sequence diagram well formedness constraints for the slice of K1 involved in checking U1. Specifically, we check that for each message in Figure 16, there is an operation reference that maps it to the appropriate method definition. The abbreviation "or" in the name of the variables means "operation reference".

Figure 18 shows the ground propositional encoding ϕ_{U2} of the property U2 for K1, in SMT-LIB. Specifically, we check whether the message `setComponentZOrder` exists in every concretization. We have combined the property with the sequence diagram well-formedness constraint additionally requiring the appropriate operation reference to the method definition.

Figure 19 shows the model K2, a concretization of K1. It was created by invoking the operator *GenerateCounterExample* with inputs K1 and U1. Mu-Mmint has greyed out Maybe elements of K1 that are not also part of K2.

Figure 20 shows the diagram of the partial model K3, resulting from invoking the operator *Constrain* with inputs K1 and U1. To enhance diagram readability, we have hidden the labels of edge elements. Every edge that is outgoing from the elements `new`, `moveToTop`, `setComponentZOrder`, and `positioner` is annotated with Maybe.

Figure 21 shows the (concrete) model K4, resulting from invoking the operator `Decide` with K3 as input while choosing the repairs RC4 and RD2.

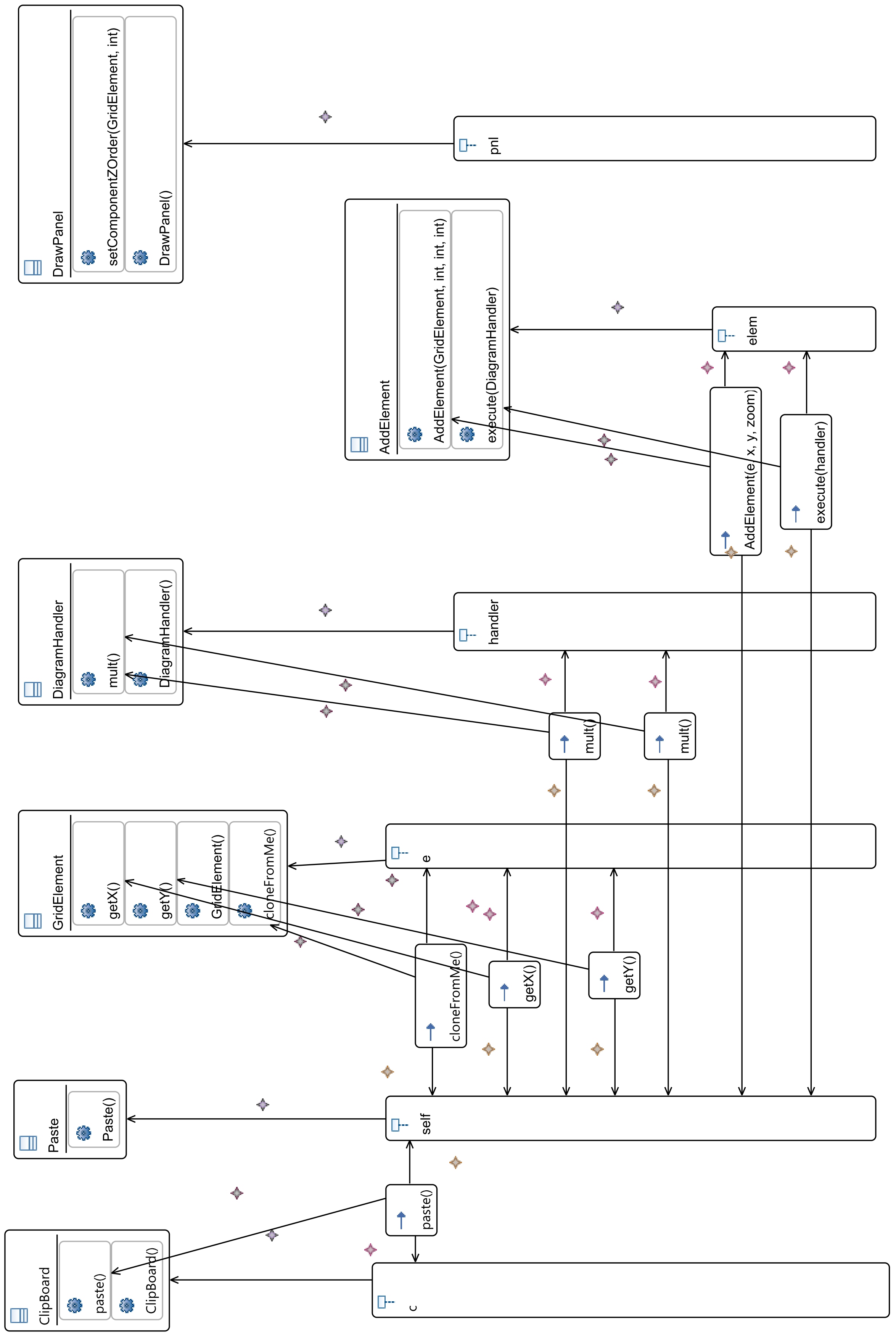


Fig. 14 Mu-Mmint model K0.

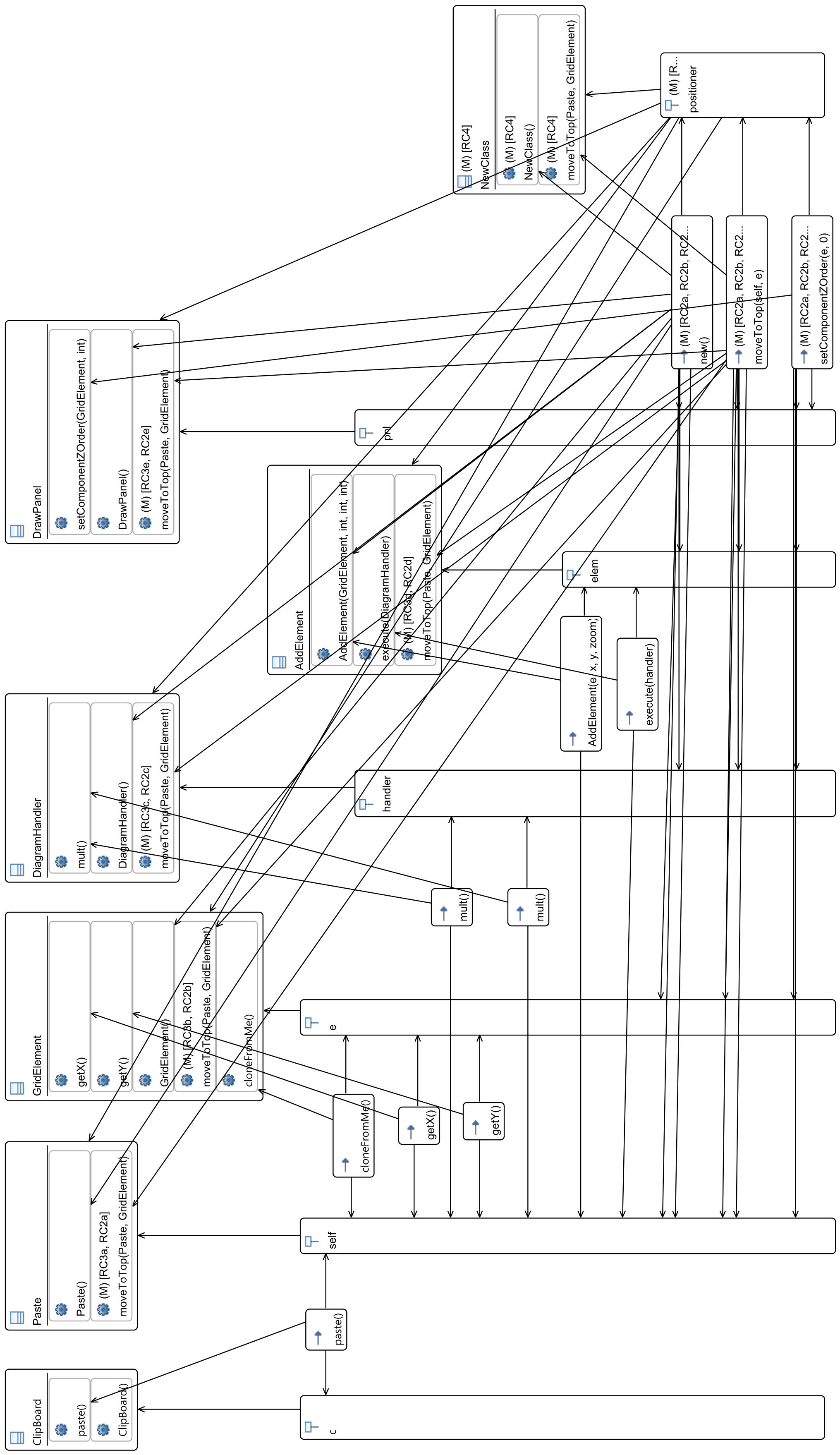


Fig. 15 Diagram of the Mu-Mmint partial model K1.

```
(and
    (node paste_message)
    (node cloneFromMe_message)
    (node AddElement_message)
)
```

Fig. 16 Ground propositional encoding of the property U1.

```
(and
    (edge or-paste-message2ClipBoard)
    (edge or-cloneFromMe_message2cloneFromMe)
    (edge or-AddElement_message2AddElement_constructor)
)
```

Fig. 17 Ground propositional encoding of sequence diagram well-formedness constraints involved in checking the property U1.

```
(and
    (node setComponentZOrder_message)
    (edge or-setComponentZOrder_message2setComponentZOrder)
)
```

Fig. 18 Ground propositional encoding of the property U2.

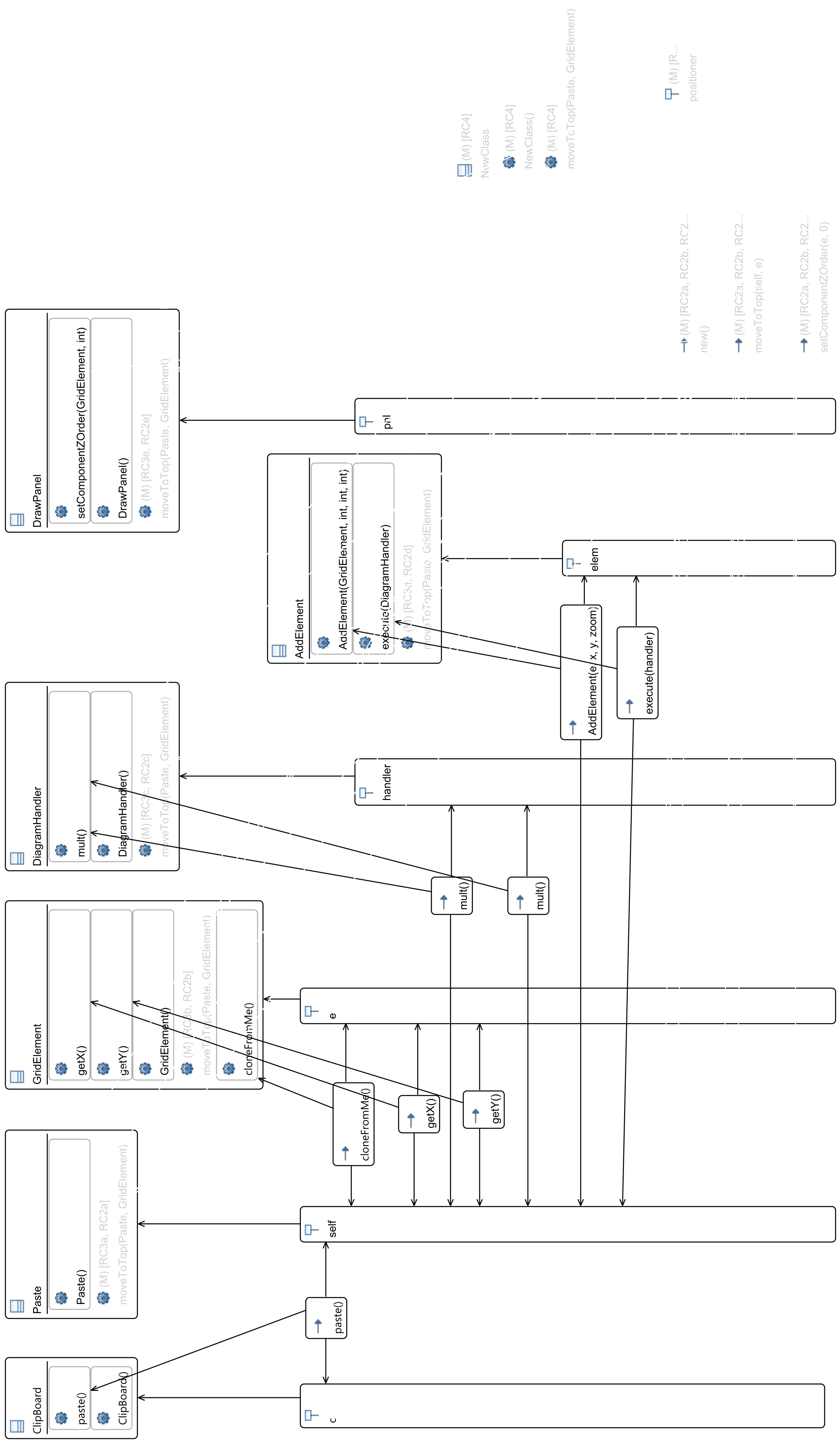


Fig. 19 Concretization K2 of the partial model K1, a counterexample demonstrating why checking U1 yields Maybe. MU-MMINT has greyed out elements of K1 that are not also part of K2.

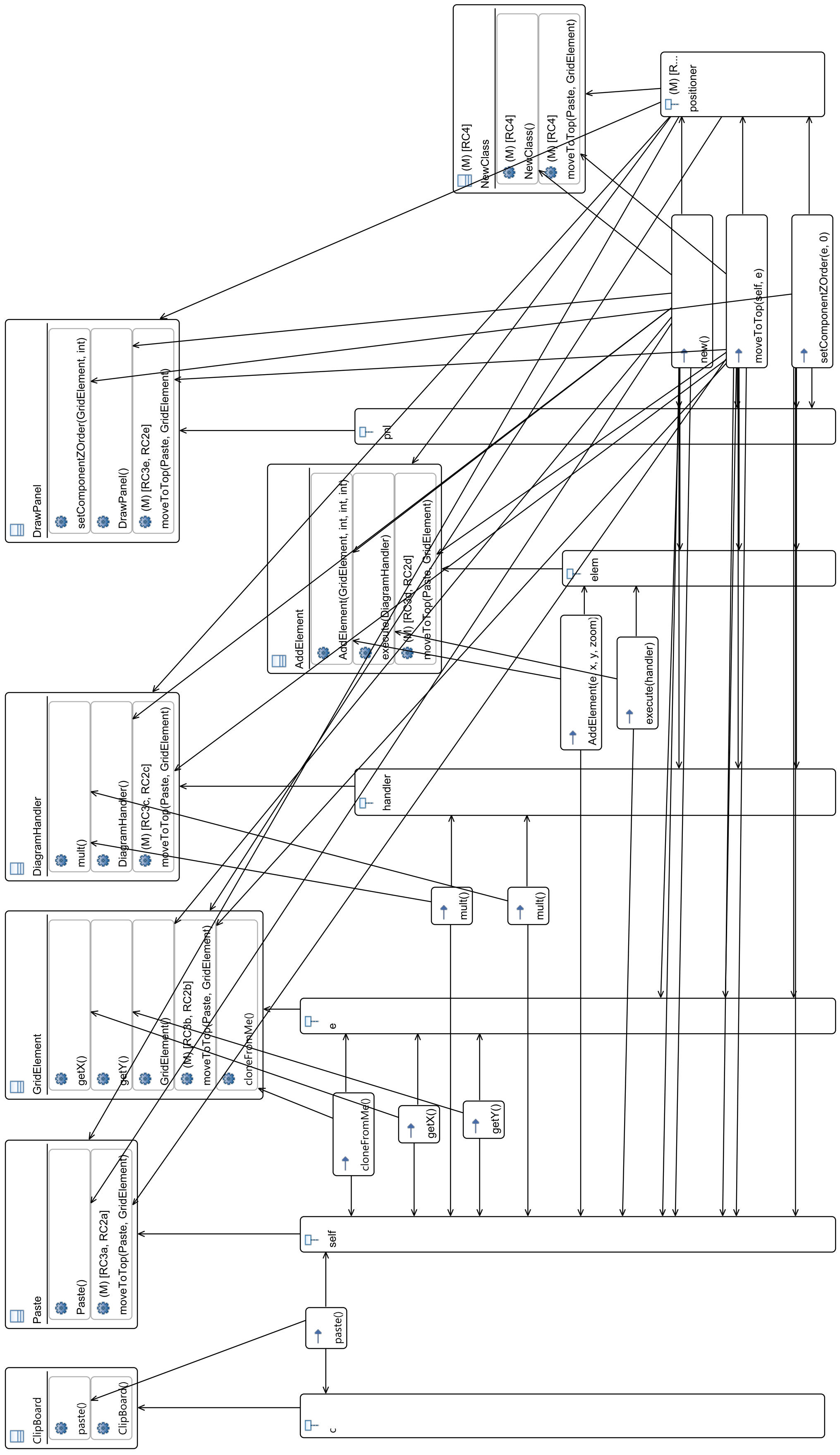


Fig. 20 Diagram of the partial model K3.

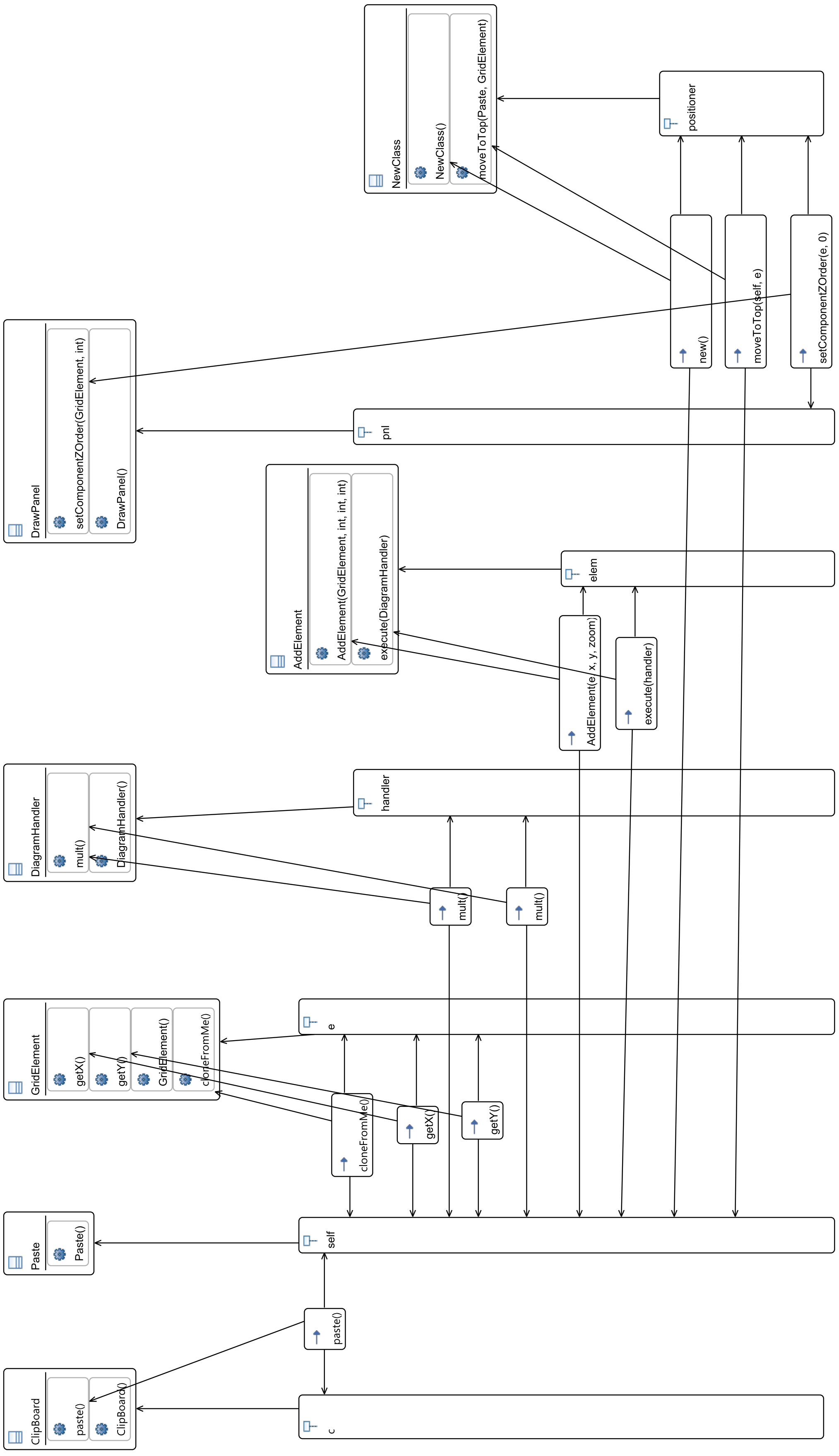


Fig. 21 Diagram of the final model K4, implementing the repairs RC4 and RD2.

B.2 Petri net metamodel

Figure 22 shows the Mu-Mmint diagram of the partial metamodel N0. Meta-associations are decorated with the icon "/", whereas containment references — with the icon "✓". Maybe elements are annotated with "[M]" and one or more alternatives from the uncertainty tree of N0 in square brackets. The uncertainty tree of N0 is shown in Figure 23.

The May formula of N0 is constructed from the uncertainty tree using the technique described in Section 5. Specifically:

– The May formula is a conjunction of the decision variables:
  `d1_ArcClasses` $\wedge$`d4_Locations` $\wedge$
  `d5_TokenClass` $\wedge$`d7_Executions`
– Each decision variable is equivalent to an exclusive disjunction of the alternative variables. For example:
  `d1_ArcClasses` $\Leftrightarrow$((`d1ynw` $\bigoplus$`d1yw`) $\bigoplus$`d1n`).
– Each alternative variable is equivalent to the conjunction of the Maybe elements that are annotated with the alternative and the negations of the Maybe elements that are annotated with other alternatives of the same decision. For example:
  `d1n` $\Leftrightarrow$ `src_placeToTransition_Association` $\wedge$
  `src_transitionToPlace_Association` $\wedge$
  `dst_placeToTransition_Association` $\wedge$
  `dst_transitionToPlace_Association` $\wedge$
  $\neg$ `PlaceToTransition_Class` $\wedge\neg$
  `TransitionToPlace_Class` $\wedge \ldots \wedge$
  $\neg$ `weight_P2T_Attribute` $\wedge$
  $\neg$ `getWeight_P2T_Operation` $\wedge$
  $\neg$`setWeight_P2T_Operation` $\wedge \ldots$

With the exception of the class `Location`, the various meta-attributes, and the getter and setter operators, the diagram of the metamodel N0 was created by merging slices of the AMZ metamodels listed in Table 2. We sliced the AMZ metamodels in order to get a model that can be used with the ORM transformation described in Appendix C, which requires the input class diagram to have a flat class inheritance hierarchy.

Figure 24 shows the ground propositional encoding ϕ_{U3} of the property U3, expressed in SMT-LIB [?]. The formula encodes the property as a conjunction of the variables (cf. *atomToProposition*) of the tables that are needed to map tuples from tables representing graphical PTN elements to tuples of the table `Location`.

Figure 25 shows the partial PTN metamodel N2 that results from invoking the operator *Decide* on the partial model N0 to select the alternative `d4y` of the decision `d4_Locations` in the uncertainty tree in Figure 23.

Figure 26 shows the diagram of the partial PTN metamodel N3 that results from invoking the operator *Expand* on the partial model N2 to include uncertainty about which domain-specific PTN constructs should be included in the ConcMod tool.

Figure 28 shows the concrete PTN metamodel N5, resulting from invoking the operator `Decide` with N3 as input and making the decisions `d1yw`, `d2n`, `d3n`, `d5y`, `d6n`, `d7y`, `d8n`, `d9n`, `d10n` from the uncertainty tree in Figure 27.

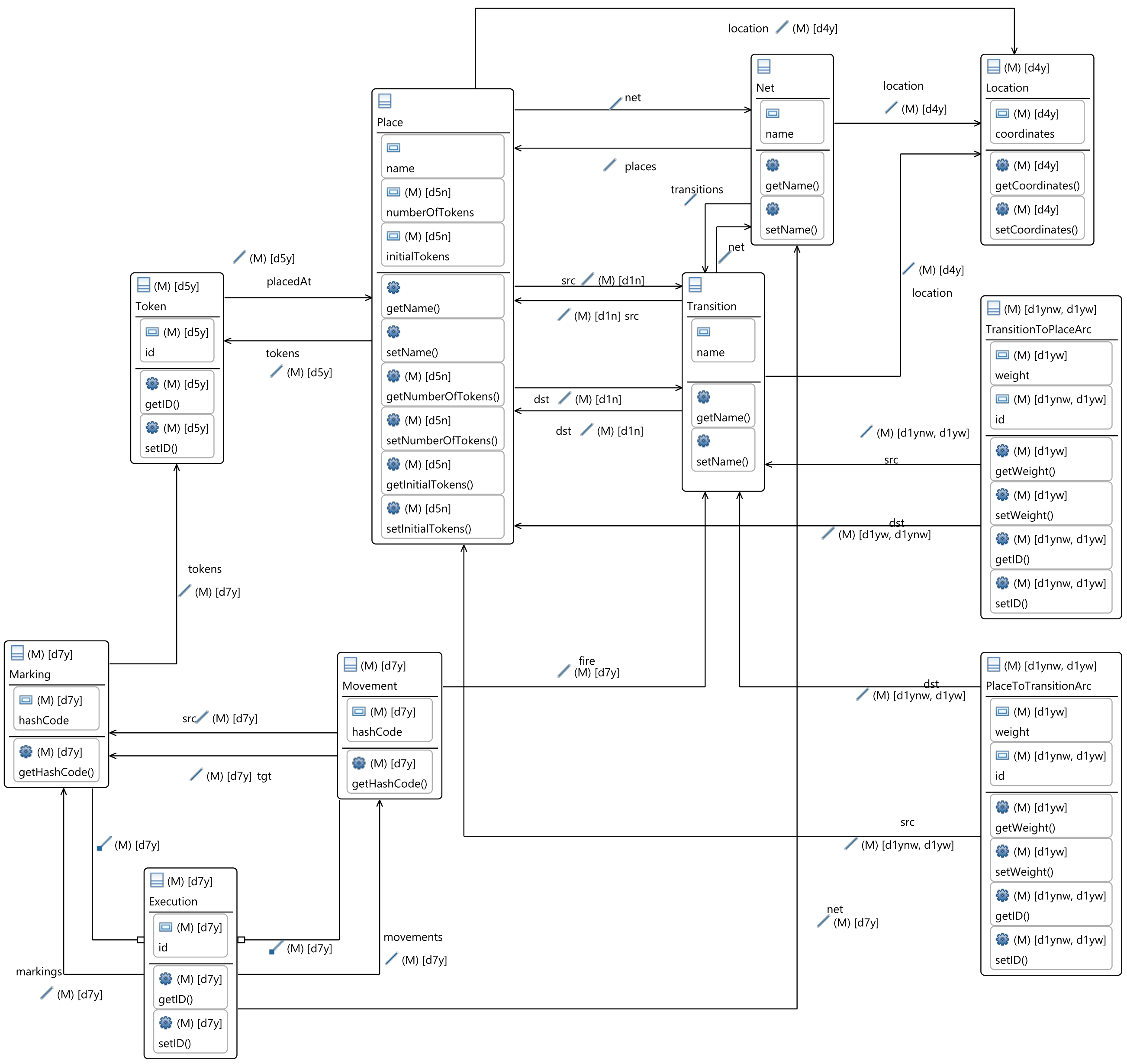


Fig. 22 Diagram of the partial metamodel N0.

d1_ArcClasses Are arcs represented as separate meta-classes?
- [d1ynw] Yes. Arcs have weights.
- [d1yw] Yes. Arcs do not have weights.
- [d1n] No.

d4_Locations Is the location of elements in the diagram stored?
- [d4y] Yes.
- [d4n] No.

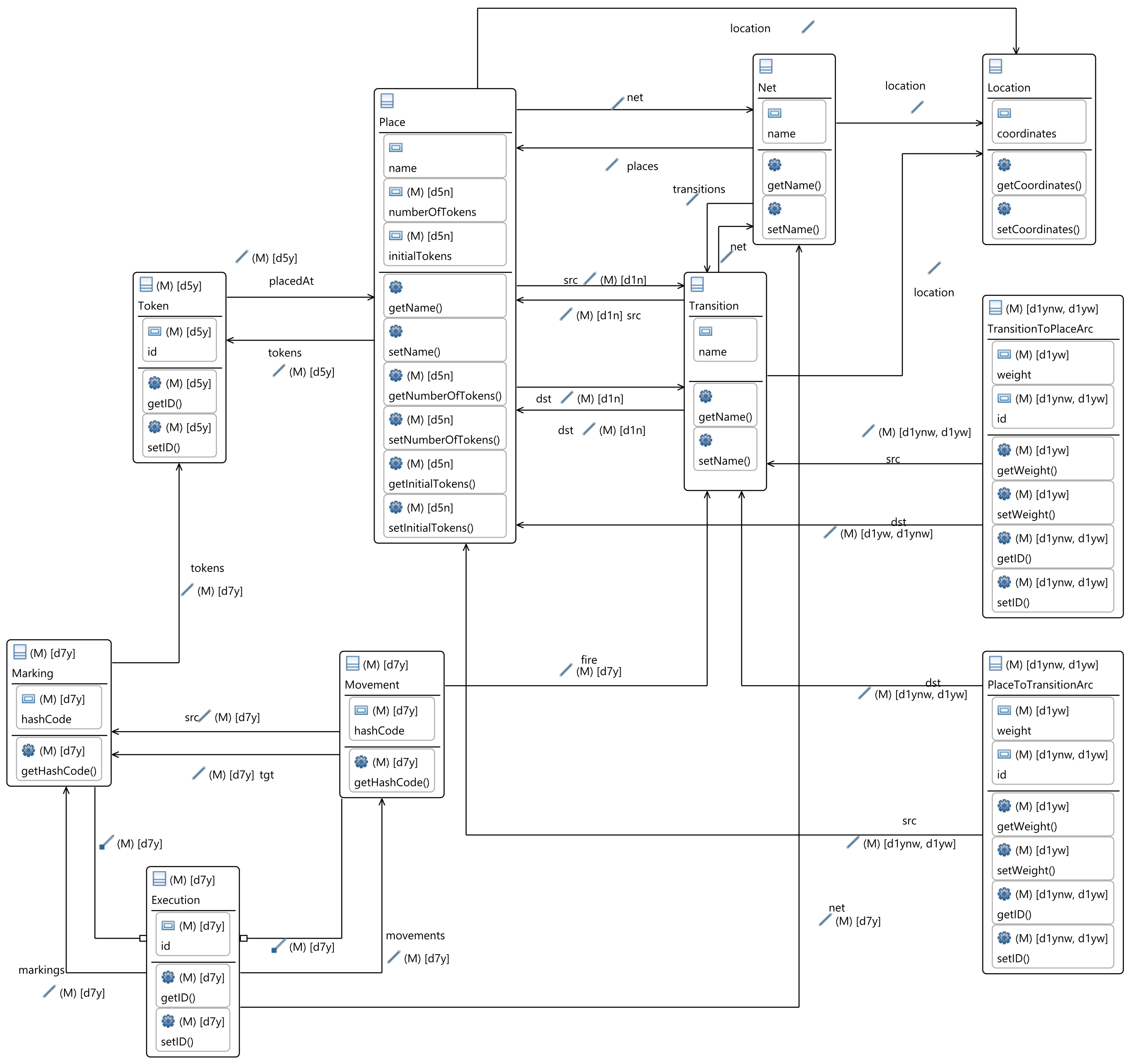
d5_TokenClass Is there a separate meta-class for tokens?
- [d5y] Yes.
- [d5n] No.

d7_Executions Is there a mechanism fore representing executions?
- [d7y] Yes.
- [d7n] No.

Fig. 23 Uncertainty tree of the partial metamodel N0.

```
(and
    (node placeToLocation)
    (node transitionToLocation)
    (node tokenToLocation)
    (node placeToTransitionArcToLocation)
    (node TransitionToPlaceArcToLocation)
)
```

Fig. 24 Ground propositional encoding of the property U3.



d1_ArcClasses Are arcs represented as separate meta-classes?

[d1ynw] Yes. Arcs have weights.
[d1yw] Yes. Arcs do not have weights.
[d1n] No.

d5_TokenClass Is there a separate meta-class for tokens?

[d5y] Yes.
[d5n] No.

d7_Executions Is there a mechanism fore representing executions?

[d7y] Yes.
[d7n] No.

Fig. 25 Partial PTN metamodel N2.

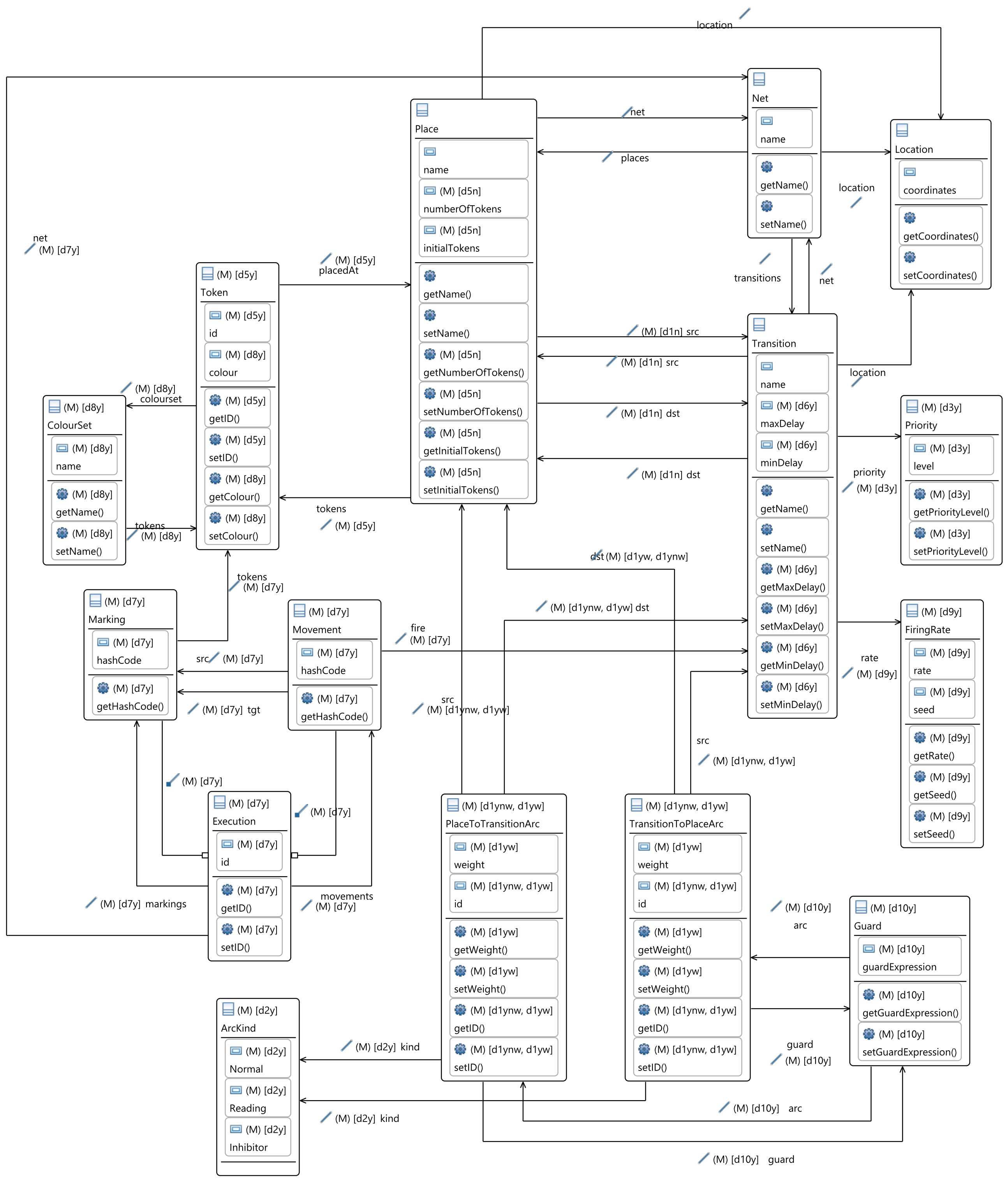


Fig. 26 Partial PTN metamodel N3. The uncertainty tree is shown in Figure 27.

d1_ArcClasses Are arcs represented as separate meta-classes?
- [d1ynw] Yes. Arcs have weights.
- [d1yw] Yes. Arcs do not have weights.
- [d1n] No.

d2_ArcKinds Are there different kinds of arcs?
- [d2y] Yes.
- [d2n] No.

d3_Priority Do transitions have priority?
- [d3y] Yes.
- [d3n] No.

d5_TokenClass Is there a separate meta-class for tokens?
- [d5y] Yes.
- [d5n] No.

d6_Timed Are transitions timed?
- [d6y] Yes.
- [d6n] No.

d7_Executions Is there a mechanism fore representing executions?
- [d7y] Yes.
- [d7n] No.

d8_ColouredTokens Are tokens coloured?
- [d8y] Yes.
- [d8n] No.

d9_Stochastic Are transitions stochastic?
- [d9y] Yes.
- [d9n] No.

d10_Guards Are arcs guarded?
- [d10y] Yes.
- [d10n] No.

Fig. 27 Uncertainty tree of the partial metamodel N3.

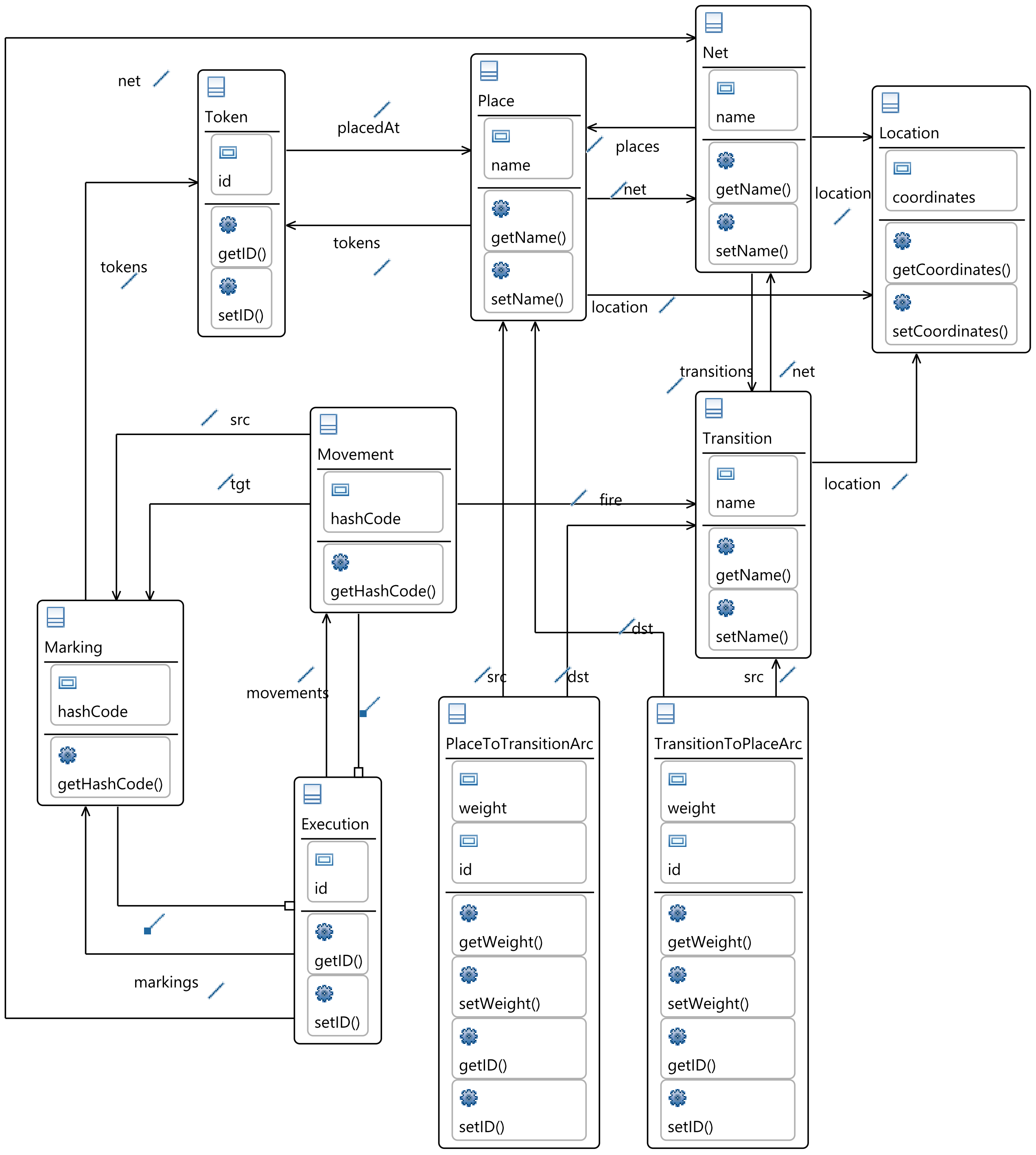


Fig. 28 Diagram of the final PTN metamodel N5.

C Appendix: Object-Relational Mapping

Figure 29 shows the first three Henshin rules `classToTable`, `associationToTable`, and `attributeToColumn` used to perfom the Object-Relational Mapping (ORM) transformation.

Figure 30 shows the last two Henshin rules `attributeToForeighKey` and `associationToForeignKey` used to perfom the ORM transformation.

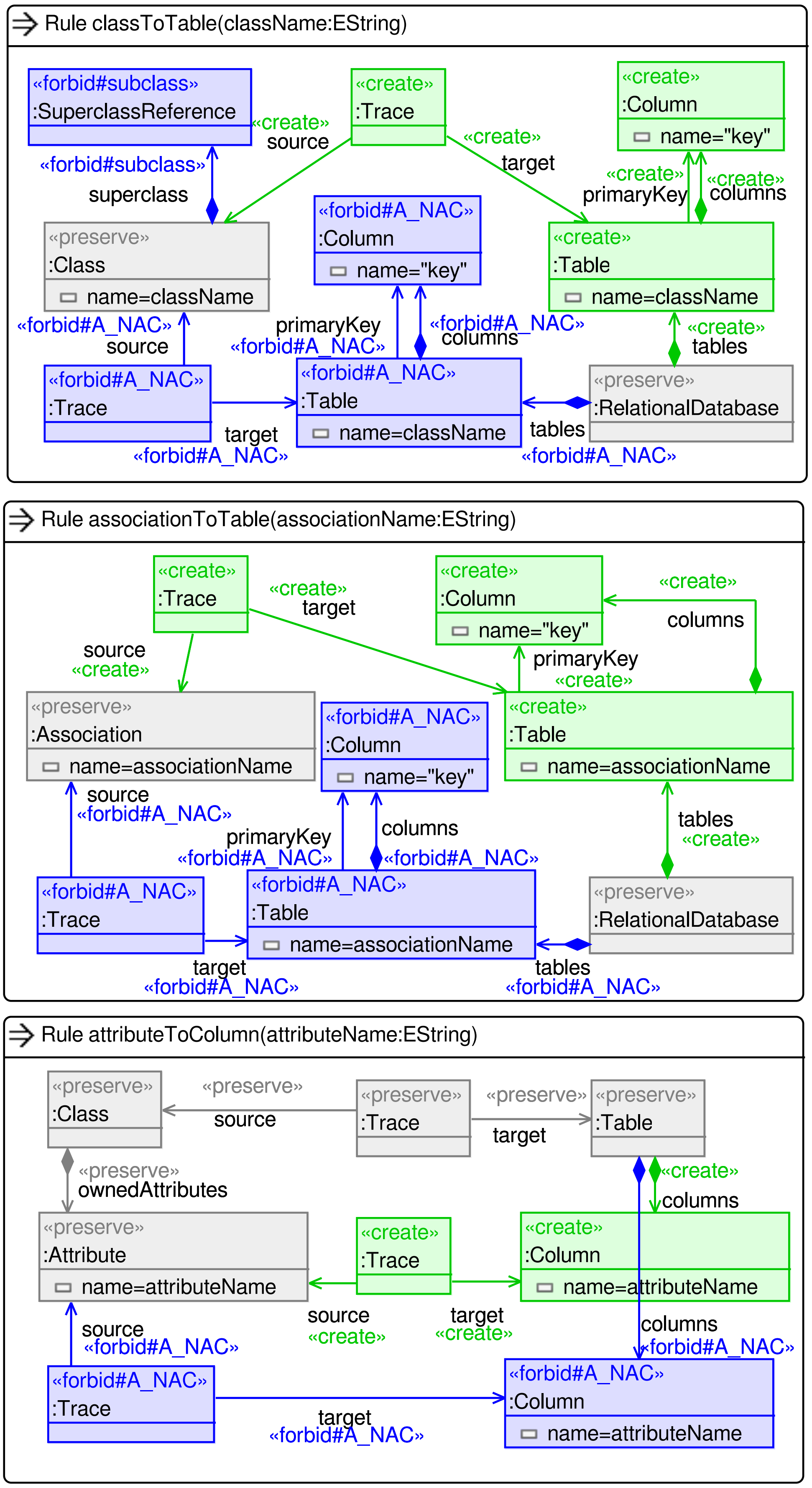


Fig. 29 Rules `classToTable`, `associationToTable`, and `attributeToColumn` used to perfom the ORM transformation.

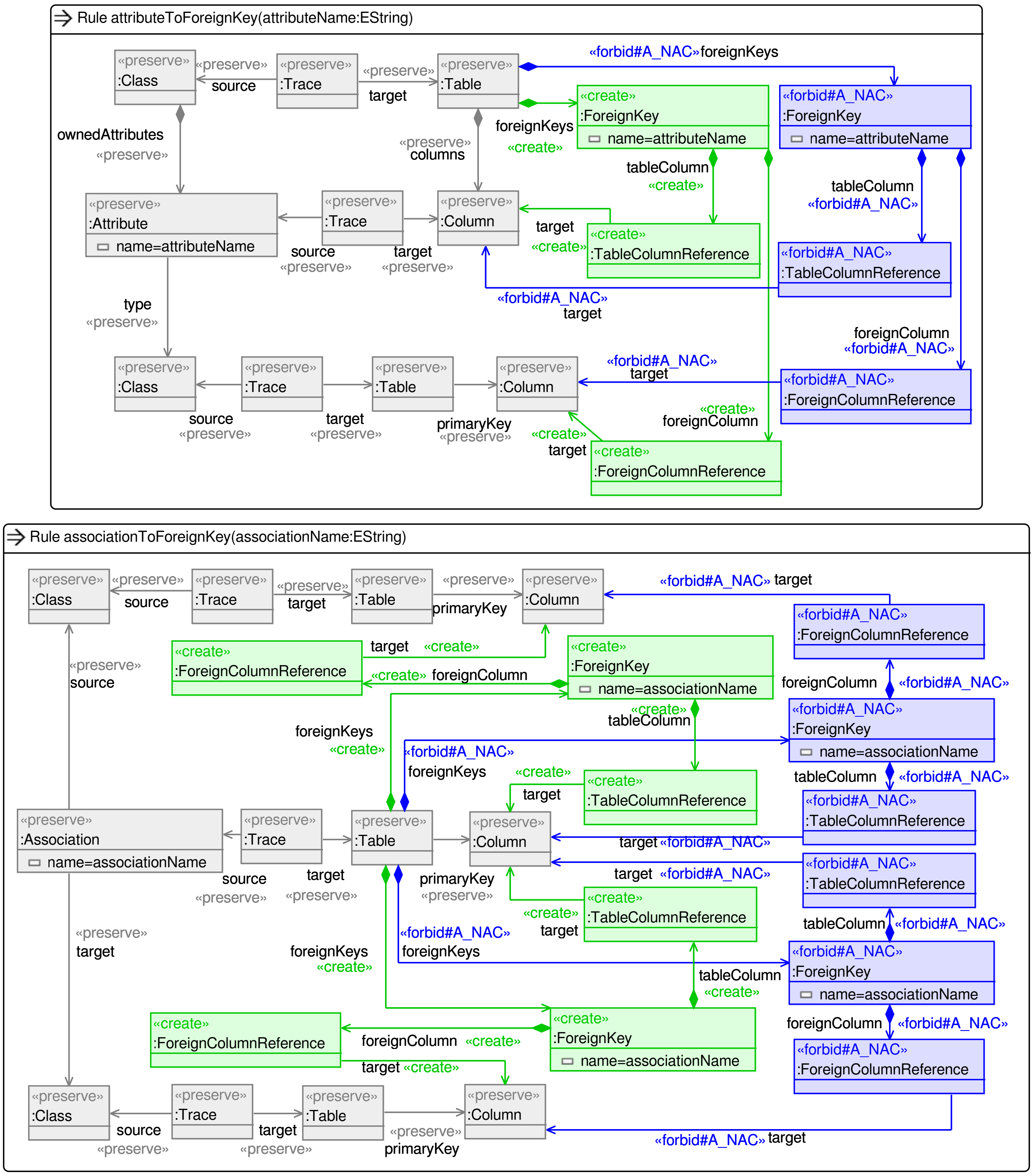


Fig. 30 Rules `attributeToForeighKey` and `associationToForeignKey` used to perfom the ORM transformation.