Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems

Stefan Naujokat[1], Louis-Marie Traonouez[2], Malte Isberner[1], Bernhard Steffen[1], and Axel Legay[2]

[1] Technische Universität Dortmund, Chair for Programming Systems, Dortmund, D-44227, Germany
{stefan.naujokat,malte.isberner,steffen}@cs.tu-dortmund.de
[2] IRISA / INRIA, Rennes, F-35042, France
{louis-marie.traonouez,axel.legay}@inria.fr

Abstract. In this paper we discuss an elaborate case study utilizing the domain-specific development of code generators within the Cinco meta tooling suite. Cinco is a framework that allows for the automatic generation of a wide range of graphical modeling tools from an abstract high-level specification. The presented case study makes use of Cinco to rapidly construct custom graphical interfaces for multi-faceted, concurrent systems, comprising non-functional properties like time, probability, data, and costs. The point of this approach is to provide user communities and their favorite tools with graphical interfaces tailored to their specific needs. This will be illustrated by generating graphical interfaces for timed automata (TA), probabilistic timed automata (PTA), Markov decision processes (MDP) and simple labeled transition systems (LTS). The main contribution of the presented work, however, is the metamodel-based domain-specific construction of the corresponding code generators for the verification tools Uppaal, Spin, PLASMA-lab, and Prism.

1 Introduction

Code generators can be regarded as the enablers for model-driven software engineering (MDSE) [1], as they provide the means to bridging the final gap to the actual use of a system. Despite this importance the state of the art is still pretty disappointing: typically, models and code generators in MDSE environments are very generic and only generate partial code which needs to be manually completed. This does not only require a lot of expertise but it also leads to the typical problems of round-trip engineering whenever the systems evolve. Domain-specific tools have the potential to overcome this situation by providing full code generation for their naturally more restrictive contexts.

Metamodeling frameworks support the development of domain-specific modeling environments to great extent, but the development of code generators for a domain-specific language (DSL) defined in those frameworks is still a complicated task despite the existence of special *code generator DSLs*, such as Xtend [2]

T. Margaria and B. Steffen (Eds.): ISoLA 2014, Part I, LNCS 8802, pp. 481–498, 2014.

for the Eclipse modeling ecosystem [3], the MetaEdit+ Reporting Language (MERL) [4] in the context of Domain-Specific Modeling [5] with MetaEdit+ [6], or the IPTG language of Eli/DEViL [7,8]. These code generator DSLs provide means that extend the possibilities of manual programming with simple string concatenation or template frameworks, but they are difficult to learn and very generic: they are specific to the underlying metamodeling framework, but do not exploit the specifics of the considered problem domain.

The Cinco framework[1] [9] aims at aiding in the development of domain-specific modeling tools in a holistic fashion that in particular comprises code generation. While the domain's metamodel and the graphical editor can be fully generated from higher-level specifications, Cinco additionally provides modeling tool developers with a domain-specific code generator language specifically generated for *their* tool. In fact, this domain-specific code generation language can be automatically obtained from the same abstract specification as the GUI.

In this paper we discuss an elaborate case study utilizing this domain-specific development of code generators within the Cinco meta tooling suite: we show how to rapidly construct custom graphical interfaces for different kinds of concurrent systems, comprising non-functional properties like time, probability, data, and costs. As a result, the corresponding user communities and their favorite tools are provided with graphical interfaces tailored to their specific needs. This will be illustrated by generating graphical interfaces for timed automata (TA) [10], probabilistic timed automata (PTA) [11], Markov decision processes (MDP) [12] and simple labeled transition systems (LTS) [13]. The main contribution of the presented work, however, is the metamodel-based domain-specific construction of the corresponding code generators for the verification tools Uppaal [14], Spin [15], PLASMA-lab [16], and Prism [17].

We do not know of any other approach that provides the automatic generation of domain-specific code generator languages. As mentioned before, existing languages are commonly specific to the used modeling framework, but not specific to one's very own metamodel. Cinco's code generation concepts are based on preliminary work [18,19] for the creation of transformation and code generation modeling components for arbitrary Ecore [20] metamodels.

The paper is structured as follows: in order to be able to explain the Cinco specification, transformation and code generation concepts using the "PSM" (Parallel Systems Modeling) case study as running example, the upcoming Section 2 motivates and explains it in detail. Section 3 then presents the basic concepts of Cinco and how the specification of the full graphical editor works. Section 4 details on the code generation concepts and how the individual code generators for our target tools are realized, before Section 5 elaborates on the model-to-model transformation of the various source model types into the PSM intermediate language. The paper concludes with a summary and plans for future work in Section 6.

[1] Cinco is developed open source under the Eclipse Public License 1.0. The framework as well as example projects are available at the Cinco website: `http://cinco.scce.info`

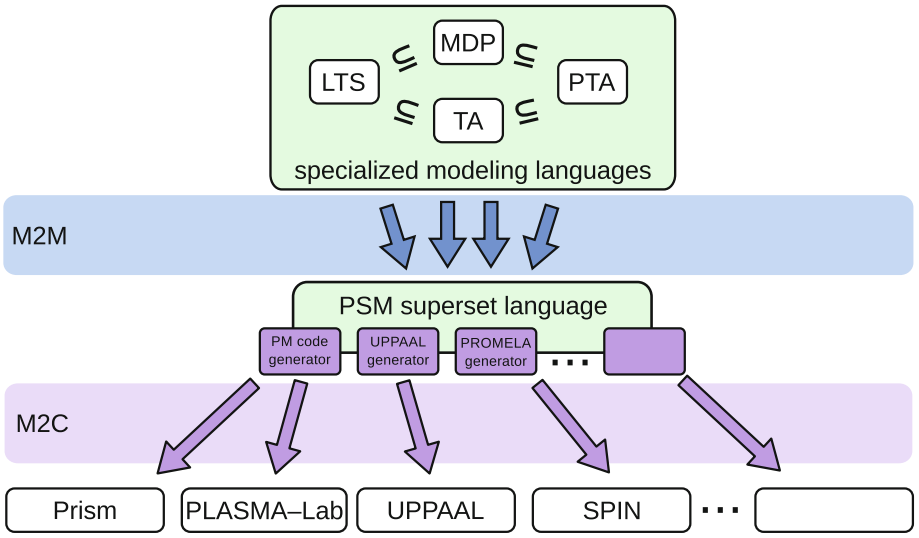


Fig. 1. Model-to-code (M2C) generation of multiple model checkers' input formats from one general model type after model-to-model (M2M) transformations on the specialized model types

2 The PSM Case Study

There is a wealth of model checking tools of concurrent systems, each with their own profile concerning the supported communication paradigm and features like time, probability, data, and costs. It is therefore not surprising that these tools come with dedicated input languages and formats, which can be graphical as in the case of Uppaal or textual, as in most other cases. This makes it difficult to work with (more than one of) these tools, in particular, as one needs to be aware of their syntactic peculiarities.

The "Parallel Systems Modeling" (PSM) case study therefore facilitates Cinco to rapidly construct custom graphical interfaces for the types of concurrent systems supported by those tools. The envisioned realization does not only allow one to customize the graphical interface, but also to generate tool-specific code which can directly be used as input in the considered tool landscape. In fact, it is possible to easily design one's own graphical language which can then be provided at low cost with code generators for the input formats of the considered tools.

2.1 Architecture

In our case study project several different model types (or languages) are involved, each of them represented by an own metamodel generated from a Cinco specification. Fig. 1 illustrates their overall architecture and interplay based on various model to model (M2M) and model to code (M2C) transformations.

Graphically designed model structures for LTSs, MDPs, TAs, and PTAs are transformed into the *PSM superset language* which faithfully captures all language paradigms provided by those 'source' languages. This richness implies some discipline in its use: by far not every syntactically correct PSM model makes sense. Of course, this does not pose any problems for the PSM models generated from the graphically designed 'source' language models, as they are consistent by construction. Moreover, constraints reminiscent of a type discipline can be used to address the consistency problem also directly at the PSM level.

The code generators for the considered target tools are then based on this PSM language to have one common technical input format that is available to all code generators. Of course, although the PSM models are consistent by construction, not each of them can be fed into all the code generators, as not all considered model types are supported by all target tools. Sometimes unsupported features can be emulated by others, e.g. modeling probabilistic decisions as nondeterministic choices, but in case this is not possible, the respective code generator will produce an error message.

2.2 Language Feature Selection

As a starting point, we have begun developing the PSM model type to contain the following language features of LTSs, MDPs, TAs, and PTAs (further features can be added at need):

Modules (or processes) define the concurrent components of the system. Each module is designed with a dedicated automaton.

States are the most fundamental modeling components. Their description typically comprises a name (i.e. a unique identifier), and additional information in terms of atomic propositions like being a start or an accepting state.

Data is stored in local variables (that belong to a module) or global variables. They can be updated with arbitrary expressions using assignments. We assume a C-like syntax for the expressions. Integer and Boolean variables are currently available, with the former one having a defined range (usually smaller than [0,MAXINT]).

Time is modeled using dedicated clock variables that all increase at the same pace. Clocks are always local and can only be reset (i.e. set to 0 using an assignment).

Guards and Invariants are expressions over all the variables that control the possible evolution of a module. Guards are assigned to edges and control which transitions are possible in a given a state. Invariants are assigned to a state and limit the evolution of clock variables in this state.

Probability is frequently used in models to represent uncertain behavior. It can be inserted using probabilistic decision nodes that split a transition into several outcomes according to probability weights. Additionally, a rate can be assigned to each state to model an exponential distribution for modeling the progress of time.

Nondeterminism is allowed whenever two transitions are enabled at the same time.

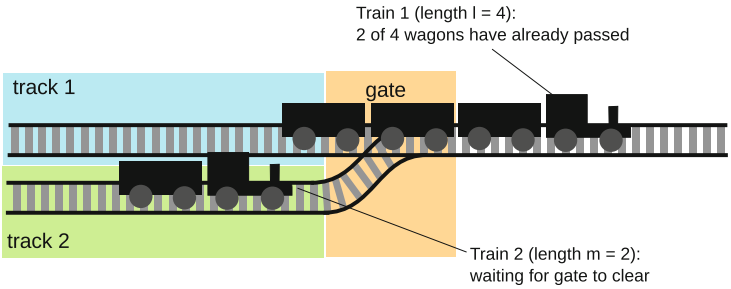


Fig. 2. Train 2 waiting for Train 1 to pass the gate

Arbitrary Code in form of C functions is realized in two ways. In the first case a dedicated node type contains a String attribute where the function body can be inserted. The second case makes use of Cinco's *prime reference* concept: Externally defined components can be placed in the model via drag&drop, creating a different kind of node that is automatically linked to that external element. This way arbitrary external libraries of modeling components can be included without the need to change the tool or metamodel. This closely resembles the concept of Service Independent Building Blocks (SIBs) from jABC [21].

Communication between processes is realized with channels on via transitions may be synchronized. They are declared globally and can be of three types:

- Pairwise (handshake) synchronization involves two transitions (possibly identified with input and output modalities added to the channel) chosen among the possible synchronizations.
- Global synchronization involves all the modules that may synchronize on the channel. Thus it may introduce deadlock in case one of the involved modules is not ready to participate in the corresponding communication.
- Broadcast synchronization involves a sender (identified with an output modality) that synchronizes with all the enabled transitions labeled by an input of the channel.

2.3 Example System

We consider a classical rail road example where two tracks are merged into one at a gate section and a controller of this system must ensure that trains arriving from either track will never collide by blocking one of the trains until the other one has passed (cf. Fig. 2). The time for passing the gate would in reality depend on several factors (e.g. length, speed, acceleration/deceleration etc.), but for our simplified example model we consider the amount of wagons the single factor that determines the passing time.

3 High-Level Graphical Editor Specification

Technically, our solution is realized using the Cinco meta tooling suite [9], a framework based on various libraries from the Eclipse ecosystem [22,20,3]. Cinco is designed to ease the generation of tailored graphical modeling tools from specifications in terms of metamodels.

Metamodeling is the modern answer to the development of domain-specific tools. However, although popular metamodeling solutions – such as the Eclipse Modeling Framework (EMF) and its multitude of accompanying plug-ins – are quite rich in the provision of code generation and transformation features, it is still tedious to develop sophisticated graphical modeling tools on their basis. The goal of Cinco is to simplify this development by providing means to specify a tool's model structure as well as its graphical user interface (and partly also semantics) in an abstract fashion that suffices to automatically generate the whole corresponding modeling tool.

The key to obtaining this degree of automation is the restriction of EMF's generality to focus on graph-based models consisting of various types of nodes and edges. With this reduction, Cinco follows the "easy for the many, difficult for the few"-paradigm that dictates the bulk of problems to be solvable very easily [23]. In our experience, it is surprising how far this paradigm carries and how seldom we need to resort to the difficult for the few part.

At the core of each Cinco product[2] lies a file in the *Meta Graph Language* (MGL) format, which is in fact a domain-specific language for the specification of model types consisting of nodes and edges.[3] Listing 1.1 shows an excerpt from the `PSM.mgl`: the specified node type `State` as well as the edge type `GuardedTransition` have several declared attributes that allow to con-

[2] With the term Cinco product (CP) we denote a modeling tool that is developed using Cinco.

[3] This actually makes MGL a meta modeling language or, synonymously, a meta-metamodel for Cinco products.

```
1  @style(state, "${number}")
2  node State {
3    attr EInt as number
4    attr EBoolean as isStartState
5    attr EString as invariant
6    attr EString as exponentialRate
7  }
8
9  @style(guardedTransition, "${guard}", "${channel}")
10 edge GuardedTransition {
11   attr EString as guard
12   attr EString as channel
13   sourceNodes (State)
14   targetNodes (State, Assignment, ProbabilisticDecision)
15
16 }
```

Listing 1.1. Excerpt from the MGL file

figure their instances. Furthermore, the valid source and target node types are configured for the `GuardedTransition`. MGL allows arbitrary annotations to be added to the elements, which are interpreted by *meta plug-ins* during the Cinco tool generation process.

The annotation `@style` is used to refer to an element from the second core DSL of Cinco: the style definition file. Elaborating on the possibilities to describe the nodes' and edges' visual appearance in the generated editor is beyond the scope of this paper.[4] Just note that it is possible to combine different shapes, colors, line types etc. Beyond this static declaration it is also possible to add dynamic elements. For instance, the contents of attributes can be passed as a parameter to the style (as done with the parameters `guard`, `channel`, and `number`). Those parameters are formulated in the Java Expression Language (EL). They are evaluated at runtime and live updated whenever the attribute values change. Furthermore, it is possible to have arbitrary `AppearanceProviders` implemented in Java, which we, for instance, use to dynamically show a small arrow tip in the top left corner of a `State` node's visual representation in case `isStartState` is true.

Overall, the Cinco PSM tool specification that fully realizes the model and its editor only consists of 84 lines of MGL code, 155 lines of Style code, and 19 lines of Java code. In contrast, the generated Graphiti editor alone[5] already consists of over 7,000 lines of code. Of course, generated code tends to be a bit verbose, but it is fair to say that Cinco reduces the amount of code writing by an order of magnitude. Moreover, the required code is much simpler and better structured. In particular it does not require special knowledge about Eclipse and the Graphiti APIs.

Figure 3 shows a screenshot of the generated editor. It consists of some common Eclipse parts (called *views*), such as the Project Explorer, Miniature View, Console, and Properties. In the center is the main editor area showing the model for our train example. It consists of three `Module` containers, one for each train and one for the gate. The gray circles are `State` nodes, of which the one with the arrow tip marker depicts the initial state of the module. The small gray squares represent probabilistic decisions and the blue rectangles are variable assignments. Small circles with different background colors represent clocks, channels, and variables. Variables can either be placed within a module to become local variables, or outside, directly on the diagram canvas to become global variables. Channels are only allowed outside while clocks must be local.

The train model works as follows: in the transition from the initial state 0 to state 1 the train's approach is signaled via channel $appr1$, the local timer x is reset and the length of the train, here modeled as a random decision, is assigned to the variable l. If the train is signaled to stop within 10 time units, it will go to state 2 and wait for the resume signal on channel $go1$. Otherwise, it can pass

[4] Please refer to [24] for a detailed explanation on the *Meta Style Language* (MSL) and the other possibilities of the GUI generator.

[5] The Ecore metamodel is generated as well, together with the Java code that implements certain `EOperations`. However, much of the corresponding over 6,000 lines of Java code are generated by the EMF framework without Cinco support.

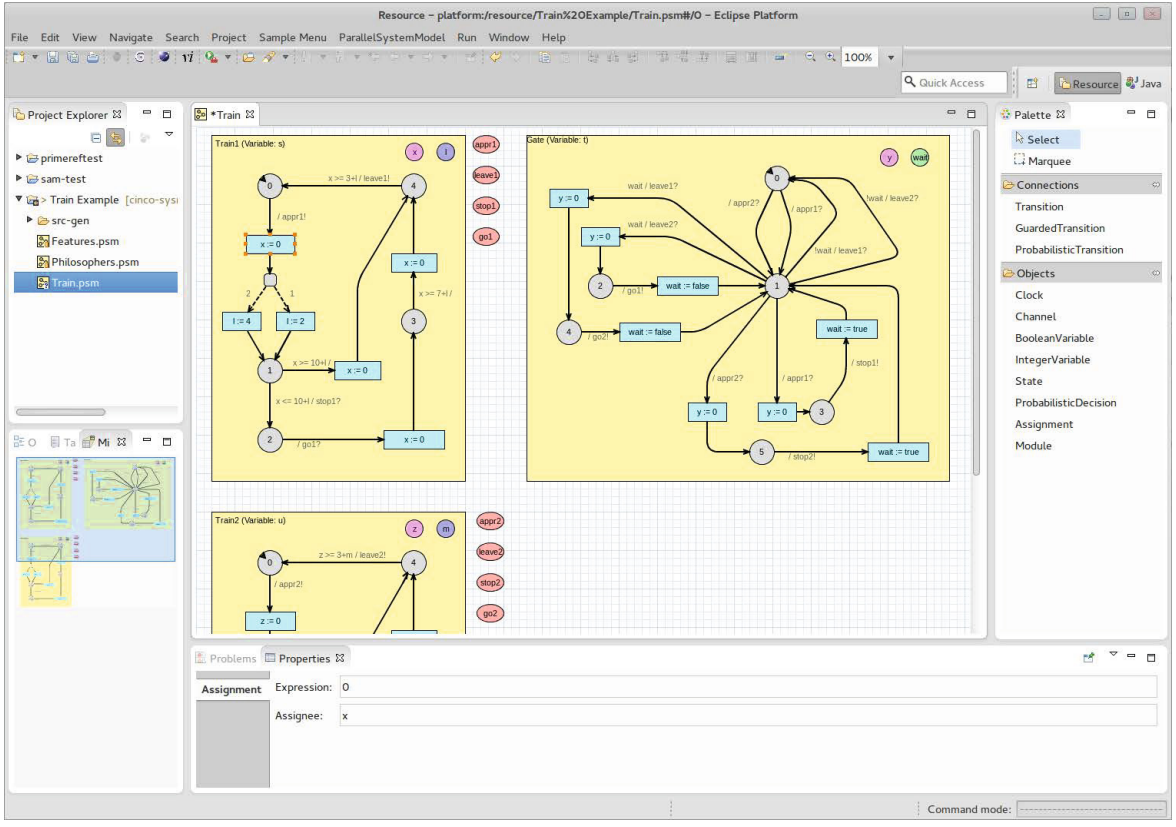


Fig. 3. Screenshot of the generated PSM tool showing the train crossing example

the gate. The second train is modeled analogously. The gate will transition from state 0 to 1 as soon as one of the two trains arrives. If the second train arrives before the first one has left, the local variable *wait* is set to *true*. In the waiting state 1, as soon as the first train has left, the go signal is given to the other one, either via state 2 or 4.

4 Code Generation

So far we have explained how Cinco can be used to easily construct a metamodel and a graphical editor for the PSM superset language. In order to introduce semantics, a code generator needs to be realized, providing a translational semantics for PSM models. In fact, we will have three different code generators, one for each target language.

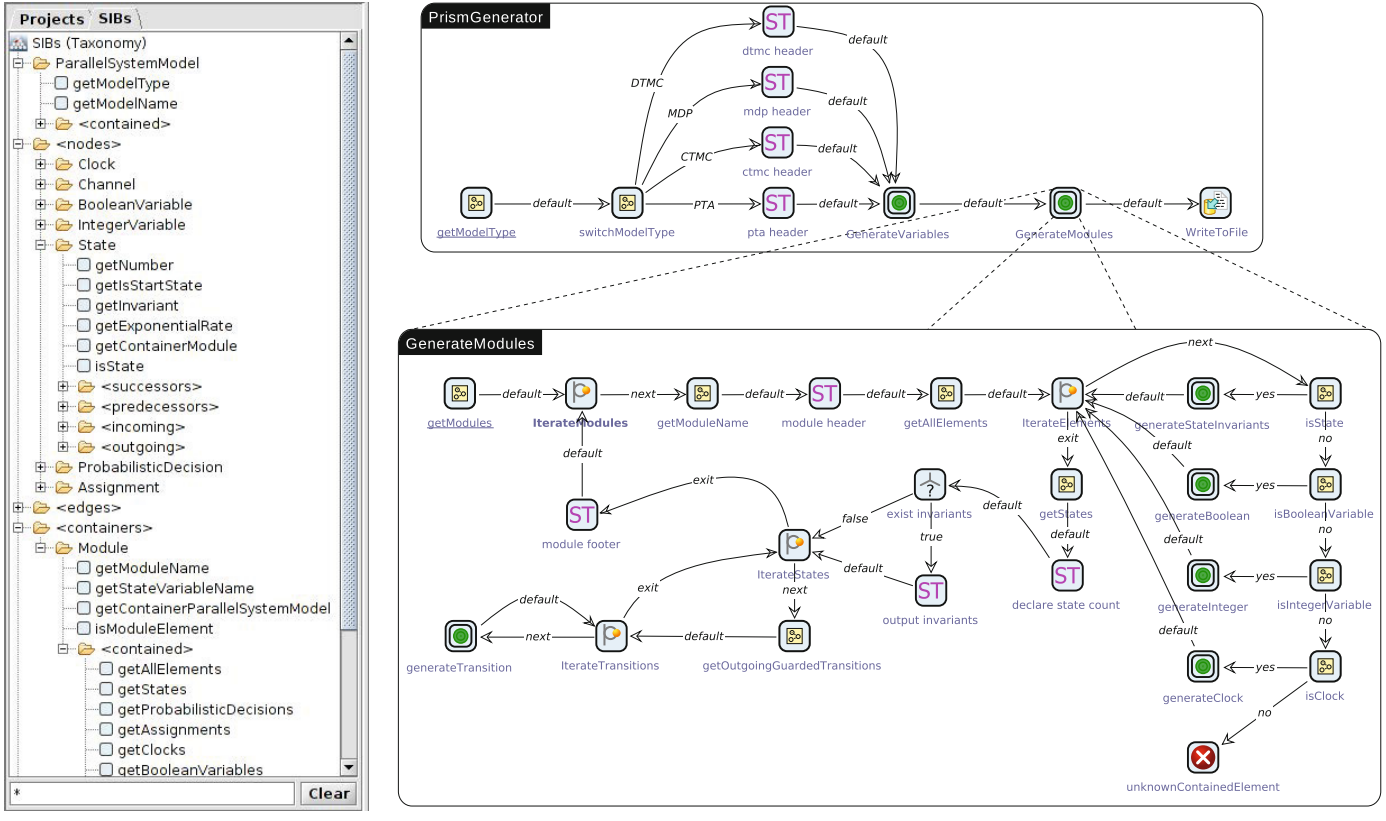
Cinco comes with a meta plug-in for code generation that interprets a `@generate` annotation in the MGL file. It generates the required Eclipse code, so that the programmer of the generator does not need to take care of any Eclipse APIs. A *Generate* button is added to the action bar of the Cinco product that triggers the generation of the currently edited model. The generator realization has to implement a certain interface and is then directly provided with the model, leaving all the Eclipse details transparent to the developer.

4.1 Domain-Specific Code Generation

The (main) metamodel of a Cinco product is generated from the MGL specification. This means that we have more precise knowledge on the metamodel's structure than in the standard, purely Ecore-based EMF settings: models always have a graph structure with different node, container and edge types. Thus, the Cinco metamodel generator is directly enabled to generate a library of domain-specific functionality that allows for systematically traversing a given model instance. For example, the following operations are directly supported with a metamodel generated with Cinco:

- Retrieval of successors and predecessors of a given node element. Those specifically generated getters can even be parameterized with a node type to only retrieve successors and predecessors of a given type.
- Access to the source and target nodes with correctly typed getters for every edge type.
- Type sensitive access to all the inner modeling elements of a container (i.e. nodes or other containers).

Please note that these are only some examples and that we constantly enrich the Cinco generator with new domain-specific operations. Of course, similar functionality can also be directly implemented for Ecore metamodels that are not generated with Cinco. However, this would require a lot of tedious code comprising of "instanceof" checks and type casting. In contrast, the Cinco approach allows for the fully automatic generation of this domain-specific functionality.



(a) PSM components (SIBs)

(b) Top-level generator model and expansion of the submodel “GenerateModules”

Fig. 4. Excerpts from the modeled PRISM code generator with domain-specific component library

Within the CINCO approach we provide domain-specific functionality for the development of code generators in a twofold fashion: for Java programmers on the one hand a dedicated API is generated. Technically, this is for the most part realized by generating special `EOperation`s and their according implementations into the metamodel. On the other hand, for domain experts who are not necessarily programmers, we generate the same domain-specific functionalities as modeling components for the jABC process modeling framework [21] (cf. Fig. 4(a)). The resulting *Service-Independent Building Blocks* (SIBs) can then easily be combined with SIBs for output generation (e.g. for StringTemplate [25] or Velocity [26]; see also [18]) into a modeled code generator. Figure 4(b) shows an excerpt from the modeled PRISM code generator (cf. upcoming Sec. 4.2) using the generated PSM components as well as some generic components for text generation with StringTemplate (`ST` on the icon), file I/O, and common tasks such as iterating over elements and error processing. The first model shows the top level of process hierarchy (i.e. the generator root model), while the second one exemplary shows one expanded submodel. While the gain for non-programmers using the jABC is obvious, we think that also people who know programming (which probably can be assumed for developers of modeling tools) strongly benefit from our domain-specific API, as it hides the internal Eclipse structures and is thus also service-oriented in spirit.

Each of the following subsections details on one code generator. As PRISM and PLASMA-LAB use the same input format, they are treated in one section, followed by the UPPAAL generator and the PROMELA generator. The code generators each assume that certain restrictions apply to the model, as the PSM language allows us to build models none of the code generators can handle anymore. Of course, if a new target platform supports more features included in PSM, they would be easy to capture by a corresponding code generator.

4.2 Prism/PLASMA-Lab

The Reactive Module Language (RML) is a textual language based on the Reactive Modules of Alur and Henzinger [27]. The language has been introduced in the model-checker PRISM [17] and is also used by the statistical model-checker PLASMA-LAB [16]. It describes a set of concurrent components with four different semantics:

1. Discrete time Markov chains (DTMCs).
2. Continuous time Markov chains (CTMCs).
3. Markov decision processes (MDPs).
4. Probabilistic timed automata (PTAs).

Each semantics imposes some restrictions on the syntax of the language, although the main elements described below are similar. We present briefly the syntax of the language[6] and then discuss the generation of RML models from our framework.

[6] See `http://www.prismmodelchecker.org/` for a more complete description.

In RML each component is modeled as a *module* that consists of a set of local declarations of integer and Boolean variables, and a set of *commands*. A command is enabled by a guard and then performs a probability choice among a set of *updates*. An update consists of a set of assignments that update the values of the variables of the model. Commands can be a assigned to a channel, in which case a global synchronization must be performed between all the modules that communicate with the channel. In CTMCs probability choices are governed by rates according to a *race semantics*, whereas in DTMCs, MDPs and PTAs only probability values may be used. The sum of the probabilities of a command must then be equal to 1. RML also allows to model PTAs by introducing real-time clocks as a new type of variables and an invariant expression to each module.

To generate a RML model from our framework, each state is assigned to a value of the state variable of the module. Then each meta-transition from a state to a set of states, passing trough assignments and probability node, is translated in a command. In this process assignments that happen before a probability node are copied in each update. This operation is only safe if these assignments prevent any side effects in the value of the probabilities. Finally, if the model is not a CTMC, all the probabilities of leaving a probabilistic decision node are normalized such that in the generated code the sum of the probabilities is always equal to 1. The resulting generator is able to produce valid RML models from our meta-model under the following restrictions:

- Synchronizations are only global.
- Clocks are only used in the PTA model.
- Synchronized transitions only update local variables.

Example 1. From the model presented in Fig. 3 we generate the following RML code:

```
module Train1
  x : clock;
  l : [0..4] init 0;
  s : [0..4] init 0;

  invariant
  (s=1 => x<=20+l)&(s=3 => x<=15+l)&(s=4 => x<=5+l)
  endinvariant

  [appr1] s=0  -> (2)/((2)+(1)):(s'=1)&(x'=0)&(l'=4) +
                  (1)/((2)+(1)):(s'=1)&(x'=0)&(l'=2);
  [stop1] s=1 & x<=10+l  -> (s'=2);
  [] s=1 & x>=10+l  -> (s'=4)&(x'=0);
  [go1] s=2  -> (s'=3)&(x'=0);
  [] s=3 & x>=7+l  -> (s'=4)&(x'=0);
  [leave1] s=4 & x>=3+l  -> (s'=0);
endmodule
```

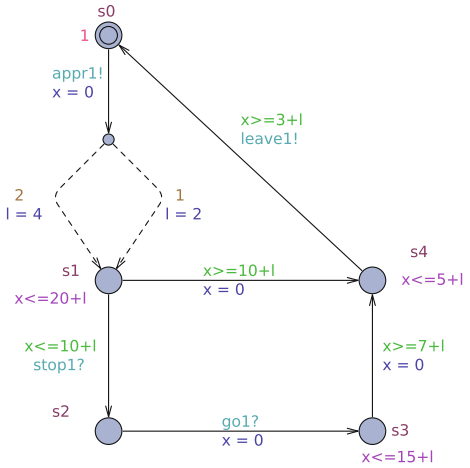


Fig. 5. UPPAAL model of the train

4.3 UPPAAL

UPPAAL [28] allows us to design timed automata models, possibly extended with variables and stochastic features, and load them from XML files. A model in UPPAAL consists of a set of *templates*, each modeled by a timed automaton. Local and global variables can be integers, Booleans, and clocks. A timed automaton consists of a set of control states, called *locations*, to which may be assigned an invariant expression and an exponential rate. UPPAAL also allows us to use probabilistic decision nodes that are used by the statistical model-checker. Then two types of transitions are possible:

- Transitions between two states, or from a state to a probabilistic decision node, may comprise a guard expression, a synchronization channel and a set of assignments.
- Transitions from a probabilistic decision node to a state may comprise a probability weight and a set of assignments.

We can generate UPPAAL models in XML format if the following restrictions apply:

- The model is only a PTA.
- Synchronizations are pairwise or broadcast, with input and output modalities.

Example 2. From the model presented in Fig. 3 we generate the UPPAAL model of Fig. 5. The semantics of colors in this model are the following: invariants are drawn in purple, guards in green, synchronization in light blue, assignments in blue, probability weights in brown and exponential rates in red.

4.4 Promela

PROMELA (short for Process Meta Language) is the input language for the SPIN software model checker [15]. Due to the popularity of SPIN, various other tools

have been adapted to accept Promela input as well (for instance LTSmin [29]), making it an attractive choice especially for the comparison of different model checking tools.

The syntax of Promela closely resembles that of the C programming language, augmented by inter-process communication constructs such as buffered or unbuffered channels, atomic sections etc. Moreover, unlike C, Promela allows for non-deterministic choices via statements with non-disjoint guards.

A Promela model consists of one or more *process types* or *process behaviors* (`proctypes`), which may be instantiated to form running processes. These processes can communicate via (buffered or unbuffered) channels or shared (global) variables. A process type can be thought of in analogy to a C function that, when instantiated, is executed in parallel with other running processes. Like C functions, process type declarations may be parameterized, with the actual arguments supplied at instantiation time.

While this would theoretically allow for an unbounded number of instantiated processes (technically, the number of simultaneously running processes is limited to 255 in Spin), we do not make use of the possibility of run-time process instantiations: the notion of *modules* as set out in Sec. 2.2 requires that there exists an a-priori known, *fixed* set of parallel components (processes). For process type declarations which are used for a single process instance only, Promela provides the `active` keyword as a prefix to `proctype` declarations to denote that the corresponding process type is to be instantiated once at the beginning of the program.

Unlike the languages presented in the previous sections, Promela neither supports time nor probabilities. While there do exist extensions for both aspects (Probmela [30] for probabilistic processes and RT-Promela [31] for real-time properties), for this case study we chose to focus on the original Promela due to the popularity of the Spin model checker.

In order to generate Promela files, the model has to fulfill the following prerequisites:

- no clocks, invariants, or exit rates occur in the model,
- synchronizations are pairwise only, with input and output modalities,
- each guard only contains either an expression over variables or a synchronization via a specific channel, but not both simultaneously.

The last restriction is due to the fact that in Promela, evaluating the synchronization expression `chan ? MSG` is not side-effect free, and thus cannot be performed in conjunction with a simple expression such as `x >= 10`. Note that we do not forbid probabilistic choices in the model. However, as probabilities cannot be expressed in Promela, those will be realized as simple non-deterministic choices.

The translation from an automaton-like structure, as is the modeling formalism for process behavior in our tool, to Promela code can be realized in a fashion similar to that for Prism described in Sec. 4.2: each process has a (local) integer variable indicating the current state of the process, initialized to the value corresponding to the respective initial state. The process body then consists of a

`do...od` block, executing transition statements (assignments, non-deterministic choices, and state changes) according to the current state and satisfied guards. In order to be consistent with the RML notion of *updates* (cf. Sec. 4.2), each sequence of transition statements is wrapped in an `atomic` block.

5 Model-to-Model Transformations

Cinco allows for the easy creation of many different graphical modeling tools. Thus, the so far presented MGL specification of the modeling language PSM can simply be stripped down to the needed parts, fed into the Cinco tool generator, resulting in the automatic creation of a dedicated modeling tool for any language $L \subseteq PSM$. This could, for instance, result in an LTS modeling tool that only contains states and transitions with channels. In fact, the required changes to the code generators would only be marginal, as one only needs to remove those parts of the generator code that handle the no longer present artifacts, like clocks, assignments, variables etc.

The resulting tools look very similar to the full PSM modeler as presented in Fig. 3. The only immediate difference is that the components palette on the right contains fewer elements and that the elements when configured in the Properties view contain fewer parameters. Even though these differences might look marginal at first sight, they may drastically ease the working with the specialized tools.

However, manually changing the code generators is impractical, especially if extensions to PSM are made that would require all derived code generators to be manually adapted again. Therefore, as already introduced before (cf. also Fig. 1), we use only one code generator base and provide model-to-model (M2M) transformations that translate other model types into PSM. As PSM is designed to provide all the required features, these transformations turn out to essentially be simple injective mappings, which may e.g., require renaming channels into alphabet symbols for labeled transition systems.

Figure 6 illustrates this concept in more detail for labeled transition systems. There exist two different MGL specifications and thus two different metamodels (PSM.ecore and LTS.ecore) are generated. As explained before, the code generators operate on the PSM metamodel. Thus the domain-specific LTS models need to be transformed into PSM models, a fact which is hidden to the user, who is only confronted with states and transitions.

Of course, the here depicted LTS instance (Trains.lts) does not contain time, probabilities or variables. Thus, the resulting PSM instance (Trains.psm) semantically differs from the one presented in Fig. 3. It just models the gate as a semaphore preventing both trains to enter simultaneously. It can, however, now be translated into all three target languages.

Technically, the realization of the M2M transformations is a special case of the code generator concepts presented in Sec. 4.1. The only difference is that instead of generating the domain-specific library of components for *one* MGL model, we now have *two*, and that instead of reading one model type and writing text,

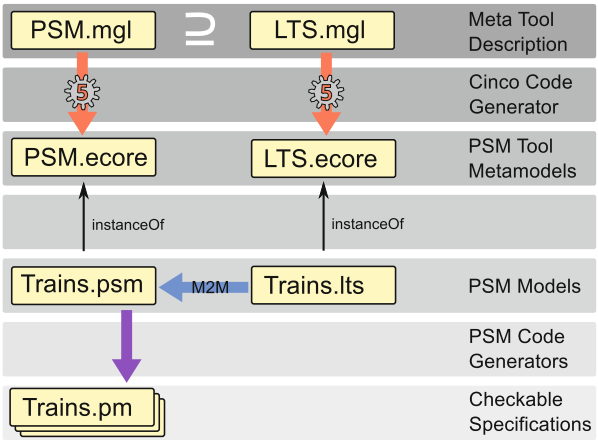


Fig. 6. Multiple products of the PSM family realized as subsets of the full language

we now read one model type and write the other. We are currently investigating ways how these transformations can also be created automatically.

6 Conclusion

We have presented the PSM case study, a generic conceptual framework to rapidly construct custom graphical interfaces with corresponding code generators for multi-faceted, concurrent systems that is based on the Cinco meta tooling suite. The point of the Cinco project is the explicit support of domain-specificity in order to simplify the tailored tool development. The impact of this approach has been illustrated by generating graphical interfaces for timed automata, probabilistic timed automata, Markov decision processes, and simple labeled transition systems, and the corresponding metamodel-based construction of code generators for Uppaal, Spin, PLASMA-lab, and Prism.

Key to the case study is the development of a 'unifying' super-set parallel systems modeling language (PSM) which serves as a 'mediator' between the multiple 'source' modeling languages and the various targeted input formats for model checking tools. This does not only allow for the generation of the domain-specific tools, but it also provides a means for systematically studying the differences and commonalities of the various system scenarios.

PSM is designed to allow the easy specification of model-to-model transformations from the source models. Moreover, our Eclipse-based framework provides automatically generated domain-specific code that frees the developer of the code generator from dealing with intricate Eclipse APIs. We envision that these features can be combined to further increase the potential of automatic code generation in order to also automatically generate the required model-to-model transformations into the intermediate PSM language.

References

1. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool (2012)
2. Xtend, `https://www.eclipse.org/xtend/` (Online; last accessed April 23, 2014)
3. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Boston (2008)
4. MetaEdit+ Version 4.5 Workbench Users Guide - 5.3 MERL Generator Definition Language, `https://www.metacase.com/support/45/manuals/mwb/Mw-5_3.html` (Online; last accessed July 31, 2014)
5. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press, Hoboken (2008)
6. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 1–21. Springer, Heidelberg (1996)
7. Kastens, U., Pfahler, P., Jung, M.T.: The Eli System. In: Koskimies, K. (ed.) CC 1998. LNCS, vol. 1383, pp. 294–297. Springer, Heidelberg (1998)
8. Schmidt, C., Cramer, B., Kastens, U.: Generating visual structure editors from high-level specifications. Technical report, University of Paderborn, Germany (2008)
9. Naujokat, S., Lybecait, M., Steffen, B., Kopetzki, D., Margaria, T.: Full generation of domain-specific graphical modeling tools: A meta^{2}modeling approach (under submission, 2014)
10. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
11. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theoretical Computer Science 282, 101–150 (2002)
12. Howard, R.A.: Dynamic Programming and Markov Processes. MIT Press (1960)
13. Katoen, J.-P.: Labelled Transition Systems. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 615–616. Springer, Heidelberg (2005)
14. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Uppaal – a Tool Suite for Automatic Verification of Real-Time Systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996)
15. Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley (2004)
16. Boyer, B., Corre, K., Legay, A., Sedwards, S.: Plasma-lab: A flexible, distributable statistical model checking library. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 160–164. Springer, Heidelberg (2013)
17. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
18. Jörges, S.: Construction and Evolution of Code Generators. LNCS, vol. 7747. Springer, Heidelberg (2013)
19. Lybecait, M.: Entwicklung und Implementierung eines Frameworks zur grafischen Modellierung von Modelltransformationen auf Basis von EMF-Metamodellen und Genesys. diploma thesis, TU Dortmund (2012)

20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
21. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-Driven Development with the jABC. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 92–108. Springer, Heidelberg (2007)
22. McAffer, J., Lemieux, J.M., Aniszczyk, C.: Eclipse Rich Client Platform, 2nd edn. Addison-Wesley Professional (2010)
23. Margaria, T., Steffen, B.: Simplicity as a Driver for Agile Innovation. Computer 43(6), 90–92 (2010)
24. Kopetzki, D.: Model-based generation of graphical editors on the basis of abstract meta model specifications. Master's thesis, TU Dortmund (2014)
25. Parr, T.: String Template, `http://www.stringtemplate.org/` (Online; last accessed July 18, 2014)
26. The Apache Software Foundation: Apache Velocity Site, `https://velocity.apache.org/` (Online; last accessed July 17, 2014)
27. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design 15(1), 7–48 (1999)
28. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST, pp. 125–126. IEEE Computer Society (2006)
29. Blom, S., van de Pol, J., Weber, M.: Ltsmin: Distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
30. Baier, C., Ciesinski, F., Grosser, M.: PROBMELA: A Modeling Language for Communicating Probabilistic Processes. In: Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE 2004, pp. 57–66. IEEE (2004)
31. Tripakis, S., Courcoubetis, C.: Extending PROMELA and SPIN for Real Time. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 329–348. Springer, Heidelberg (1996)