How to Efficiently Build a Front-End Tool for UPPAAL: A Model-Driven Approach

Stefano Schivo[1], Buğra M. Yildiz[1], Enno Ruijters[1], Christopher Gerking[2], Rajesh Kumar[1], Stefan Dziwok[3], Arend Rensink[1], and Mariëlle Stoelinga[1]

[1] Formal Methods and Tools, University of Twente,
{s.schivo, b.m.yildiz, e.j.j.ruijters,
r.kumar, a.rensink, m.i.a.stoelinga}@utwente.nl

[2] Software Engineering, Heinz Nixdorf Institute, Paderborn University,
christopher.gerking@upb.de

[3] Software Engineering, Fraunhofer IEM, Paderborn,
stefan.dziwok@iem.fraunhofer.de

Abstract. We propose a model-driven engineering approach that facilitates the production of tool chains that use the popular model checker Uppaal as a back-end analysis tool. In this approach, we introduce a metamodel for Uppaal's input model, containing both timed-automata concepts and syntax-related elements for C-like expressions. We also introduce a metamodel for Uppaal's query language to specify temporal properties; as well as a metamodel for traces to interpret Uppaal's counterexamples and witnesses. The approach provides a systematic way to build *software bridging tools* (i.e., tools that translate from a domain-specific language to Uppaal's input language) such that these tools become easier to debug, extend, reuse and maintain. We demonstrate our approach on five different domains: cyber-physical systems, hardware-software co-design, cyber-security, reliability engineering and software timing analysis.

1 Introduction

Uppaal [3] is a leading model checker for real-time systems, allowing one to verify automatically whether a system meets its timing requirements. Uppaal and its extensions have been applied to a large number of domains, ranging from communication protocols [28], over planning [4] to systems biology [31]. As such, Uppaal is a popular back-end for various other real-time analysis tools, such as ANIMO [31], sdf2ta [13] and STATE [19]. Typically such tools take their inputs in a domain-specific language (DSL) and translate these inputs into timed automata, which are then fed into Uppaal to perform the analysis. In this way, domain experts can write their models in a DSL that they are familiar with, while still using Uppaal's powerful analysis algorithms behind the scenes.

A disadvantage of this approach is, however, that the tools that translate from a DSL to Uppaal's input language, i.e., *software bridging tools*, are often implemented ad hoc, and hence difficult to debug, reuse, extend and maintain.

To overcome this problem, we advocate to develop these tools with *model-driven engineering* (MDE) techniques, which studies [26] have demonstrated can lead to faster software development, with higher levels of interoperability and lower cost. MDE is an approach that uses models as first-class citizens, rather than as by-product of intermediate steps. In MDE, a *metamodel* captures core concepts and behavior of a certain domain. Then, domain-specific models are instances of this metamodel and can be transformed to other models, formats or formalisms via *model transformations*.

In this paper, we propose an MDE approach for tools that use Uppaal as a back-end. In the context of our approach, we introduce metamodels for Uppaal timed automata, Uppaal's query language and its diagnostic traces, in order to transform the domain-specific models to Uppaal, analyze them and transform the results back to a domain-specific representation, respectively. Our metamodels also support Uppaal's extensions with cost [4] and probability [7].

We show our approach on five diverse application domains: cyber-physical systems, namely, coordination protocols of MechatronicUML; hardware-software co-design, namely, scheduling of synchronous dataflow graphs; cyber-security, namely, analysis of attack trees; reliability engineering, namely, analysis of fault trees; and software timing analysis, namely, timing analysis of Java applications.

Our contributions. To summarize, our main contribution is an MDE approach for building software bridging tools around the Uppaal model checker. Concretely, we introduce (1) metamodels[1] for Uppaal's timed automata, queries and traces, providing all the ingredients needed to construct Uppaal models, verify relevant properties and interpret the results; (2) model transformations from several domain-specific models to the Uppaal models and back; and (3) five case studies demonstrating how the approach is applied in practice and supports a wide range of application domains.

Overview of our MDE approach. The proposed approach can be seen in Figure 1. Taking into consideration the analysis of a (generic) domain-specific model, the most important steps involving a bridging software tool that implements our approach are the following:

- In *Step 1*, a domain-specific model is generated/created by the domain expert. This model is an instance of the metamodel of a particular domain of interest. Such a metamodel defines the concepts and their relationships in that domain. For some domains, it may be more convenient to define multiple related metamodels targeting distinct concerns.
- In *Step 2*, the domain-specific model is transformed to a timed-automata model, conforming to the Uppaal Timed Automata metamodel (uta) we propose as part of the contribution of this paper. A snippet of such a transformation can be found in Figure 7.

[1] The metamodels are available at `https://github.com/uppaal-emf/uppaal`.

– In *Step 3*, the property against which the domain-specific model is to be checked is specified in a query language specific to the domain.
– In *Step 4*, the query specified in the domain-specific query language is transformed to a corresponding UPPAAL query, in turn conforming to the UPPAAL Query metamodel (UQU) we propose as part of the contribution of this paper.
– In *Step 5*, UPPAAL checks if the timed-automata model (a UTA model) satisfies the property specified by the generated query (a UQU model). The result of this operation is usually a diagnostic trace. As part of this step, the UTA and UQU models are transformed into the native UPPAAL input formats; moreover, the diagnostic trace natively produced by UPPAAL is transformed into yet another model, conforming to the UPPAAL Trace metamodel (UTR) that we also propose as part of the contribution of this paper.
– In *Step 6*, the UTR model is transformed back to a domain-specific representation. This representation can conform to a metamodel that is designed to express the analysis results in an understandable way by the domain experts.

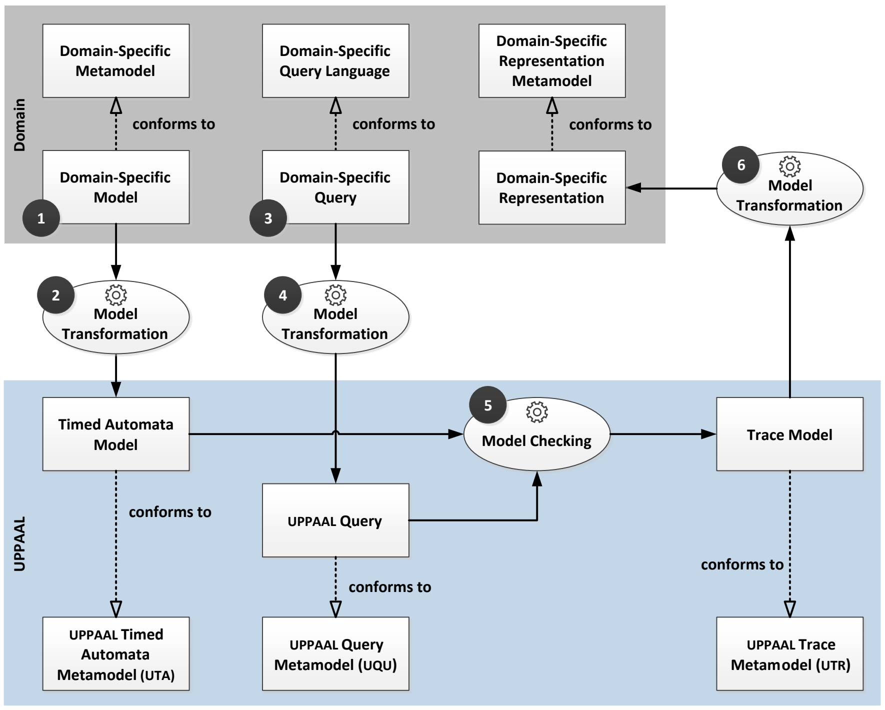


Fig. 1. The generic model-driven engineering approach for building front-end tools that use UPPAAL as a back-end analysis engine.

Organization of the paper. Section 2 provides some background information about MDE and the timed-automata formalism. Section 3 introduces the three metamodels and their transformations. Section 4 discusses the case studies. Section 5 discusses the related work and Section 6 concludes the paper.

Acknowledgements. This research was partially funded by STW and ProRail under the project ArRangeer (grant 12238), STW project SEQUOIA (15474), NWO projects BEAT (612001303) and SamSam (628.005.015), and EU project SUCCESS.

2 Background

In this section, we provide some background information about model-driven engineering (cf. Sect. 2.1) and the timed-automata formalism (cf. Sect. 2.2).

2.1 Model-Driven Engineering

Models are powerful tools to express structure, behavior and other properties in domains such as engineering, physics, architecture and other fields. *Model-Driven Engineering (MDE)* is a software engineering approach that considers models not only as documentation, but also adopts them as basic abstractions to be used directly in development processes [33].

To define models of a particular domain, we need to specify their language. In MDE, such a language (often referred to as a *domain-specific language*, DSL) is also specified as a model at a more abstract level, called a *metamodel*. A metamodel captures core concepts and behavior of a certain domain, and defines the permitted structure and behavior, to which its instances (models) must adhere. Another way of saying this is that metamodels describe the syntax of models [34]. Following the common terminology, we will write that a model *conforms to* or *is an instance of* its metamodel.

MDE provides interoperability between domains (and tools in these domains) via *model transformations*. The concept of model transformation is shown in Figure 2. Model transformations are usually defined in a language designed specifically to this aim and map the elements of a source metamodel to the elements of a target metamodel. The transformation engine executes the transformation definition on the input model and generates an output model.

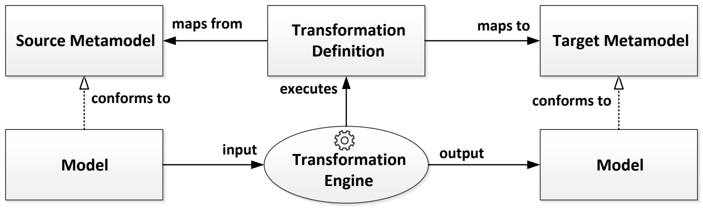


Fig. 2. The concept of *model transformation*.

Benefits of MDE. MDE provides a range of important benefits [36], some of which we briefly discuss below:

- *Interoperability:* As we have mentioned before, there can be multiple domains in a project where various tools are used, each with its own I/O formats. MDE provides interoperability between these domains (and tools in these domains) via model transformations.
- *Higher level of reusability:* The metamodels, models and tools from a domain can be reused by many projects targeting the same domains. Such reuse also increases the quality of the final product since the reused units are revised and improved continuously.
- *Faster tool development:* Domain experts only focus on the concepts of the domain while creating models. Transformations on these models are implemented using languages designed specifically for model transformations rather than using general-purpose languages. Because of these advantages of MDE, the development time of tools decreases.

Tool Choice. There are a number of tools for realizing MDE. The case studies presented in this paper are implemented using the Eclipse Modeling Framework (EMF) [35], a state-of-art tool for implementing MDE techniques. EMF provides the *Ecore* format for defining metamodels and many plug-ins to support various functionalities, such as querying, validation and transformation of models.

2.2 Timed Automata and UPPAAL

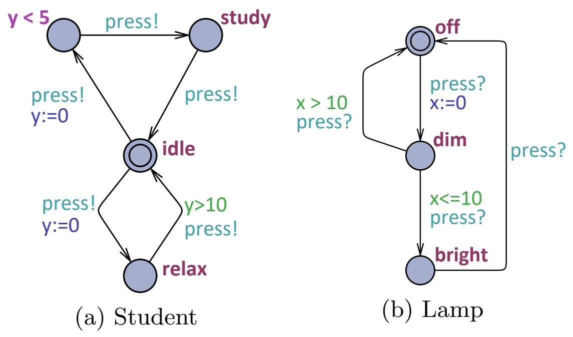
Timed automata are finite-state automata with the addition of real-valued *clocks* and synchronization *channels*. In Figure 3, we show an example timed-automata model (from [5]), with clocks x and y. *Locations* are indicated by circles (double circle for the initial location), and *transitions* are represented by edges. Conditions on clocks can enable transitions (e.g., $x > 10$ in Figure 3b, from dim to off) or allow residence in locations ($y < 5$ in Figure 3a). Synchronizations can occur when two automata perform complementary actions on the same channel: in the example, outputs press! synchronize with inputs press?. When taking a transition, clocks can be reset ($x{:=}0$, $y{:=}0$).

Timed-automata models are verified with Uppaal [3] through queries expressed in a subset of CTL [12]. In Figure 3c, we show the trace resulting from the verification of the reachability query E<>lamp.bright, which asks whether a state where the lamp automaton is in the bright location is reachable. The verification returns a positive outcome, together with a witness trace, listing the sequence of states and transitions leading to the desired target.

In addition to the standard version of Uppaal, some of the models presented in this paper are intended for analysis by Uppaal CORA [4], which allows to compute cost-optimal traces (see Section 4.2), and Uppaal-SMC [7], which allows to perform statistical model checking (see Section 4.4).

3 Metamodels for the Approach

We use metamodeling to represent the domain of timed automata and enable the backend analysis of domain-specific models. Our approach extends the work by Greenyer and Rieke [17] towards full-fledged metamodels, covering all language features accepted by the Uppaal model checker. Thereby, we make sure that model transformations may freely use any of Uppaal's concepts when translating domain-specific models into timed-automata models.



(a) Student (b) Lamp

```
Verifying formula 1: E<> lamp.bright

 -- Formula is satisfied.
Showing example trace.

State:
( student.idle lamp.off )
student.y=0 lamp.x=0

Transitions:
student.idle->student._id5 { 1, press!, y := 0 }
lamp.off->lamp.dim { 1, press?, x := 0 }

State:
( student._id5 lamp.dim )
student.y=0 lamp.x=0

Transitions:
student._id5->student.study { 1, press!, 1 }
lamp.dim->lamp.bright { x <= 10, press?, 1 }

State:
( student.study lamp.bright )
student.y=0 lamp.x=0
```

(c) Trace

Fig. 3. An example of a timed-automata model (a, b) and the textual output (c) of verifying the reachability query E<>lamp.bright as provided by UPPAAL's command-line tool.

In Section 3.1, we present the metamodel for UPPAAL timed automata (UTA). Section 3.2 describes a metamodel extension for UPPAAL's query language (UQU). A metamodel for traces obtained from UPPAAL (UTR) is given in Section 3.3.

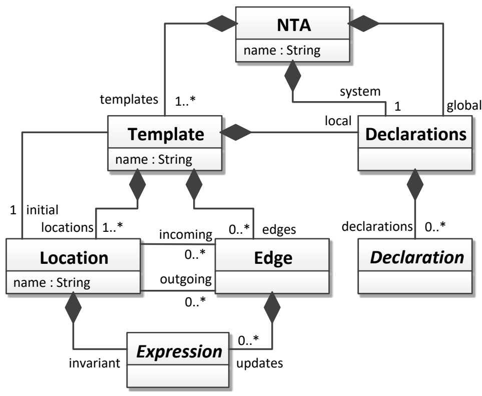
3.1 The UPPAAL Timed Automata Metamodel

Figure 4a shows an excerpt from our UPPAAL Timed Automata metamodel (UTA), extending the metamodeling approach proposed in [17]. This metamodel reflects the basic structure of timed automata accepted by UPPAAL.

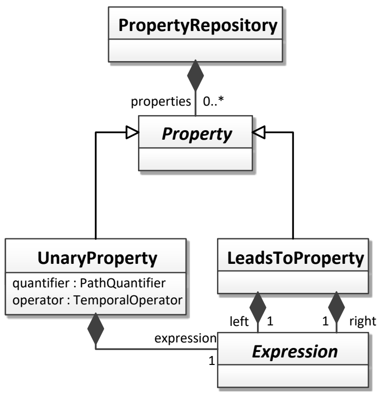
At the core of UTA is a network of timed automata (NTA). An NTA includes a set of global Declarations, containing instances of the abstract base class Declaration. A declaration is used to introduce elements such as clocks or synchronization channels. Primarily, an NTA includes a non-empty set of templates where each Template represents a type of timed automaton. Moreover, an NTA contains a separate set of system declarations. These are specific TemplateInstances (omitted from the figure), which constitute the set of concrete timed automata that make up the system to be model-checked.

Templates include locations and edges, and every Template refers to one particular initial location. Templates may also include local declarations (e.g., for clocks that should not be reset from outside the automaton). Every Location refers to its incoming and outgoing edges. In addition, a Location specifies an invariant which is a boolean expression as an instance of the abstract base class Expression. An Edge may contain expressions as well to specify updates of variables (e.g., clock resets). The metamodel also contains syntax-related elements for the C-like expressions supported by UPPAAL.

UTA models are not the native input format of UPPAAL and, therefore, are not directly processable. We have implemented a model-to-text transformation, which takes a UTA model as input and transforms it into UPPAAL native XML.



(a) UTA (UPPAAL Timed Automata)



(b) UQU (UPPAAL Queries)

Fig. 4. Partial views from the UTA and UQU.

3.2 The UPPAAL Query Metamodel

Figure 4b depicts an excerpt from our UPPAAL Query metamodel (UQU). Queries are temporal logic properties to be verified using model checking. Multiple queries are bundled by a PropertyRepository, which is the root class of the metamodel. A repository contains a set of properties, where every Property represents one query. Every property is either a UnaryProperty or a LeadsToProperty.

A UnaryProperty is a temporal formula that conforms to the *computation tree logic* (CTL, [12]). First, such a property includes a quantifier (one of *universal* or *existential* quantification) to describe whether the property must hold on all execution paths, or at least one path. Second, it consists of a modal operator (one of *globally* or *finally*) to describe if the property needs to hold in all states of a certain execution path, or needs to hold eventually in some state. Third, unary properties include an expression to be evaluated in the context of the quantifier and the operator. For example, this expression could represent an active location inside an automaton, or a clock value. To this end, UQU extends UTA and reuses the Expression class introduced in Section 3.1.

A LeadsToProperty represents a binary property connecting two expressions by means of the *leads-to* operator supported by UPPAAL. Please note that, according to the restrictions imposed by UPPAAL on the set of CTL formulas supported, our metamodel does not allow nested properties. However, we introduce dedicated classes for logical connections of expressions (omitted from Figure 4b), precisely reflecting the range of functions actually supported by UPPAAL.

Like UTA models, also UQU queries have to be transformed to UPPAAL's native format before they can be actually processed. For this purpose, we provide another model-to-text transformation.

3.3 The UPPAAL Trace Metamodel

The outcome of evaluating a query in UPPAAL can be twofold: either a simple "yes" (for a universally quantified query claiming that a given property holds for all paths) or “no” (for an existentially quantified query asking whether a path with a given property exists), and possibly a trace through the state-space of the timed-automata model along which the query fails to hold (for a universal query) or that is a witness (for an existential query). Queries are very often formulated in such a way that it is known *a priori* whether they hold or not, the interesting part of the outcome is then that diagnostic trace.

Uppaal outputs its traces in a native textual format that is not too well documented. From [6], we have taken a metamodel (utr) to capture the information in a tractable way and a parser that produces utr models from Uppaal’s output. Like uqu, also utr depends on uta itself, so that the traces can refer back to their constituent components. Figure 5 gives a high-level overview of utr.

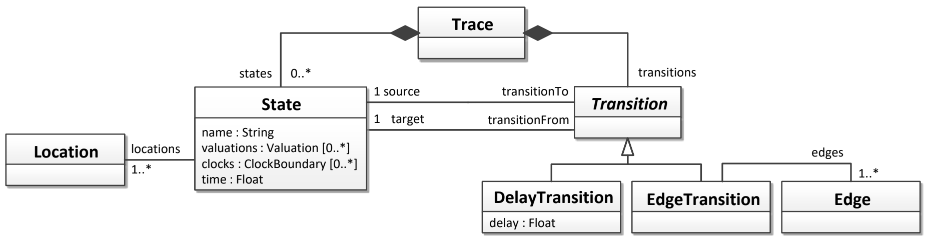


Fig. 5. A partial view from utr (Uppaal Trace metamodel).

A Trace consists of States and Transitions; every State except the final one has a single outgoing Transition. A State refers to a set of Locations (one for every Template-Instance in the system, though that cannot be seen from the provided metamodel fragment), together with Valuations, i.e., bindings for all the variables to concrete values, as well as boundaries for all the clocks in the system (the Valuation and ClockBoundary classifiers are omitted from the figure). Finally, a State stores the absolute time at which the system arrived in that state. A Transition can either be a DelayTransition, in which only time passes, or an EdgeTransition, in which a number of Edges (one for every TemplateInstance involved) fire in synchrony. Location and Edge are imported from uta.

4 Case Studies

The general MDE approach we propose for bridging software tools has been introduced in Section 1. In this section, we present five case studies that have put this approach into practice.

In Table 1, an overview of these case studies is given. After the section number, the second column shows to which domain the approach is applied. The third column contains the list of the metamodels that are used to describe that domain. The fourth column gives the motivation why model checking is used for the particular case study. The fifth column shows which steps from the approach (given in Figure 1) are implemented in the particular case study. The following subsections describe these case studies in more detail.

The transformations for the cyber-physical systems case study are specified in the QVTo [27] language, for the other cases in the Epsilon Transformation Language [21]. Translation of the timed-automata models to the XML input files for Uppaal is performed via the Xtend [11] language, using its template expressions for model-to-text transformations.

Table 1. An overview of the case studies applying the proposed approach.

Sect.	**Domain**	**Domain Metamodels**	**Motivation for using a model checker**	**Steps of the approach**
4.1	Cyber-Physical Systems	Protocol, Query	To verify whether a coordination protocol fulfills all stated properties	1, 2, 3, 4, 5, 6
4.2	Hardware-Software Co-Design	Synchronous Data Flow Graph, Hardware Platform, Allocation	To obtain a schedule for the execution of the tasks considering optimization objectives of resource and energy	1, 2, 5, 6
4.3	Cyber-Security	Attack-Fault tree	To obtain a schedule of attack steps optimizing objectives like time and cost, or stochastic values, e.g., probability of attack within mission time.	1, 2, 5, 6
4.4	Reliability Engineering	Attack-Fault tree	To obtain the probability of failure within mission time.	1, 2, 5
4.5	Software Timing Analysis	Java Bytecode, Timing Analysis Extension	To validate Java applications to ensure that they fulfill their timing specifications.	1, 2, 5

4.1 Coordination Protocols of CPSs

Future cyber-physical systems (CPS; e.g., cars, railway systems, smart factories) will heavily interact with each other to contribute to aspects like safety, efficiency, comfort and human health. They may achieve this by coordinating their actions via asynchronous message exchange. However, such a coordination must be safe and has to obey hard real-time constraints because any (timing) error may lead to severe damage and even loss of human life. Consequently, the development of so-called coordination protocols that specify the allowed message exchange sequences requires formal verification like model checking to guarantee the functional correctness of the coordination.

Model checkers like Uppaal are appropriate for verifying such coordination protocols but their language has no built-in support for domain-specific aspects like asynchronous communication including message buffers and quality-of-service (QoS) assumptions (e.g., message delay and reliability). Consequently, the domain expert has to encode these aspects manually, which is a complex and error-prone task. Therefore, the model-driven method MechatronicUML [10] defines a DSL for specifying coordination protocols of CPS at a more abstract level. Among others, this DSL enables to specify hierarchical state machines, real-time constraints, message buffers and the QoS assumptions of the protocol. Furthermore, MechatronicUML defines a domain query language to ease the specification of formal verification properties that a coordination protocol of MechatronicUML shall fulfill. For example, the requirement "At least one instance per message type of the coordination protocol can be in transit" may be specified as follows: `forall(m : MessageTypes) EF messageInTransit(m)`.

In [15,9], we have achieved to fully hide the model checker Uppaal from the domain expert by specifying domain-specific model checking for coordination protocols of MechatronicUML using Uppaal. Our approach requires all six steps that we introduce in Section 1. In particular, we assume that the coordination protocol and its domain queries are specified in Steps 1 and 3. Then, in Step 2, we transform a coordination protocol of MechatronicUML into a set of timed automata that conform to uta. Moreover, in Step 4, we transform our domain query language into properties that conform to uqu. We automate Uppaal in Step 5 and parse the textual trace into a model that conforms to the utr metamodel. Finally, in Step 6, we apply a model transformation to translate the trace back to the level of MechatronicUML in order to show the trace to the domain expert. We have implemented our concepts successfully into the MechatronicUML Tool Suite.

4.2 Synchronous Dataflow Graphs

Hardware-software (HW-SW) co-design is an engineering approach to simultaneously design the hardware and software components of a system to meet optimization objectives. *Synchronous dataflow (SDF) graphs* [25] are a frequently used formalism in the HW-SW co-design domain to represent streaming and dataflow applications in terms of their computation tasks and the data relationships among them. Tasks are represented as nodes, and data input-output relationships between these tasks are represented as edges. SDF graphs can be used to calculate an (energy- or time-) optimal schedule of an application allocated on a particular hardware platform.

In [1], we have applied the generic approach presented in this paper for scheduling analysis of SDF graphs with an energy-optimization objective. Three metamodels are introduced as domain metamodels: The SDF metamodel representing SDF graphs, the hardware platform metamodel representing multi-processor hardware platforms on which SDF graphs can be mapped, and the allocation metamodel representing such mappings. The domain-specific model, which consists of one instance of each metamodel, is transformed to a timed-automata model and is analyzed with Uppaal CORA [4]. The trace resulting from this analysis, which is an instance of the trace metamodel given in Section 3.3, represents an energy-optimal schedule. In order to make the result available to the domain experts, we have implemented a model transformation from trace models to *schedule* models. Schedule models conform to the *Schedule* metamodel (see Fig. 6) that we have developed and described below.

Schedule is the root of the metamodel. It consists of Executors, Executables and Tasks. An Executor represents a processing unit (which is usually a processor or a core) that executes a task. An Executable is a computation unit that can be executed while a Task is one execution instance of an Executable. A Task has a start time and an optional end time, which are both Time references. The end time is optional since a Schedule may contain Tasks that have not finished.

4.3 Attack Tree Analysis

Modern day infrastructures are frequently faced with cyber attacks. A key challenge is to identify the most dangerous security vulnerabilities, estimate their likelihood and prioritize investments to protect the system from the most riskful scenarios. Security

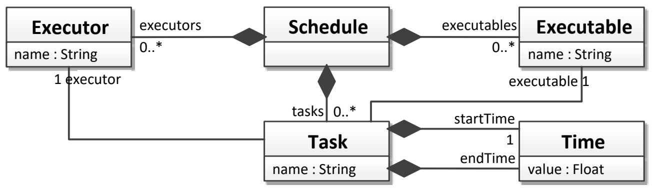


Fig. 6. The Schedule metamodel.

experts often model threat scenarios and perform quantitative risk assessment using attack trees (ATs). These describe how atomic *attack steps* (the tree leaves) combine into complex attacks (intermediate nodes, also called *gates*), leading to the *security breach* represented by the root of the tree. Over the years, numerous formalisms inspired by ATs have been proposed [22]. As they all share the same basic structure, we have developed a metamodel [20] to support interoperability between the different tools made to analyze attack trees. Furthermore, as attack trees resemble fault trees, we enriched the AT metamodel with fault tree constructs, resulting in the *attack-fault tree* (AFT) metamodel [29].

A piece of the transformation from attack trees to UPPAAL can be seen in Fig. 7. This section produces the overall structure (i.e., *system declaration* in UPPAAL) from the class called `AttackTree` in the metamodel `AFT`. The `.equivalent()` function transforms each node into an UPPAAL template and declaration, automatically selecting the transformation rule for that node.

Traditional ATs are static, and their leaves are decorated with single attributes like cost or time. In order to account for multiple attributes and temporal dependencies we defined transformations from AFT models to UTA models. The security properties that can be checked require either *optimization*, like “What is the cost-optimal path taken

```
rule Base transform at : AFT!AttackTree to out : Uppaal!NTA {
     out.systemDeclarations = new Uppaal!SystemDeclarations();
     out.systemDeclarations.system = new Uppaal!System();
     var iList = new Uppaal!InstantiationList();
     out.systemDeclarations.system.instantiationList.add(iList);
     for (node : AFT!Node in at.Nodes) {
          var converted = node.equivalent();
          if (converted <> null) {
               out.template.add(converted.get(0));
               out.systemDeclarations.declaration.add(converted.get(1));
               iList.template.add(converted.get(1).declaredTemplate);
          }
     }
     out.addTopLevel(at.Root);
}
rule andGate transform node : AFT!Node to ret : List {
     guard : node.nodeType.isKindOf(AFT!AND)
...
```

Fig. 7. Snippet of the translation from the Attack Tree metamodel to UTA.

by an attacker? [24]", or the use of *stochastic values*, like "What is the probability of an attack within m months? [23]". Similar to what we did for Synchronous Dataflow models, the results of optimization queries are computed using Uppaal CORA. The outcome of such analysis is a trace which is automatically parsed, obtaining a UTR model. A trace obtained from this analysis can additionally be transformed into a schedule, represented by an instance of the Schedule metamodel described in Section 4.2. The adoption of MDE allows us to reuse the Schedule metamodel to describe results from the attack tree domain, as they are semantically close to the SDF results. The stochastic values are computed using Uppaal-SMC. Plotting these results over time yields graphs similar to the one in Fig. 8.

Currently, the optimization and stochastic security properties are expressed as queries specific for Uppaal CORA and Uppaal-SMC, making them incompatible with the current query metamodel.

4.4 Fault Tree Analysis

As society becomes ever more dependent on complex technological systems, the failure of these systems can have disastrous consequences. The field of reliability engineering uses various methods to analyze such systems, to ensure that they meet the required high standards of dependability.

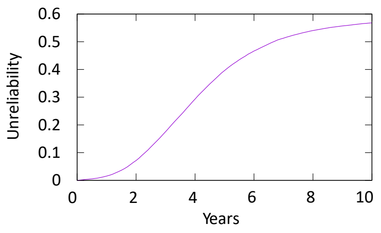


Fig. 8. Example plot of reliability over time as produced by automatic analysis of a fault tree using the Uppaal-SMC metamodel.

A popular formalism to perform such an analysis is *fault tree* analysis. Faults trees (FTs) are similar to attack trees (described in Sect. 4.3), however rather than modeling deliberate steps in executing an attack, they model component failures (called *basic events*) that may combine to cause system failures or other undesired events.

Standard FTs were developed in the 1960s and describe only boolean combinations of faults. Since then, a large number of variations and extensions of fault trees have been developed [30], covering aspects such as timing dependencies, uncertainty, and maintenance. Most of these extensions were developed independently and traditional tools do not support combinations. MDE simplifies the combination of models of different kinds and the analysis of those aspects that are shared between the different formalisms.

Fault trees are described in a unified attack-fault tree (AFT) metamodel also used for attack trees. The main difference from ATs is in the attributes of the basic events. Where attack steps are controlled by an external attacker who makes deliberate decisions based on factors such as cost, faults are inherently stochastic in nature: The failure time is not externally decided, but rather governed by a probability distribution attached to the fault.

The AFT metamodel supports basic events governed by hypoexponential distributions, and gates from standard fault trees, dynamic fault trees [8] and fault maintenance trees [29], as well as gates from attack trees.

As one of the analysis back-ends of the AFT metamodel, we provide a model transformation to a UTA model. Unlike most applications described in this paper, the analysis of this model does not result in a trace or a schedule, nor can its queries be expressed in the current query metamodel. Queries are usually probabilistic in nature, asking questions such as “What is the probability of the system failing within 5 years”. Results are then numeric values answering such queries. While it is possible to extract a trace from an FT, its value is limited due to the stochastic nature of the fault tree.

Instead, the typical use of the fault tree metamodel is to produce one UPPAAL-SMC model and automatically query the failure probability at different times. The results of these queries can than be used to produce a plot of the system reliability over time, such as the one shown in Figure 8.

4.5 Analysis of Java Programs

Model-based verification techniques for software applications require the existence of expressive models. Typically, these models are derived manually, which is a labor-intensive and error-prone task. Also, models need to be maintained and kept consistent with the software application, lest they become outdated.

The framework we have introduced in [38] adopts the generic approach presented in this paper for automatically deriving timed-automata models to validate Java applications, timing requirements in particular, using model checking. In this framework, the bytecode metamodel [37] and its timing analysis extension are introduced as the domain metamodels. The instance of the bytecode metamodel (*bytecode model*) is generated from the target Java application automatically using the JBCPP plug-in. Following this, the bytecode metamodel is enriched through a number of model transformations with additional information necessary for analysis; this includes recursion handling, loop detection, loop iteration bounding, timing information, etc. The additional information is represented as an instance of the extension metamodel. The enriched bytecode model is then transformed to a UTA model to be analyzed with UPPAAL.

Queries are currently manually written and results of the model checking process are not translated back to a domain-specific representation such as to a source-code view. However, the implementation of these points using MDE is suggested in the generic approach is a future direction of the study in [38].

5 Related Work

There are many studies that use UPPAAL to verify systems. We limit this section to the studies that automatically transform domain-specific models to timed automata, or map the results of model checking back to the domain of interest.

The tool ANIMO (Analysis of Networks with Interactive MOdeling) [32] has been introduced to analyze complex biological processes in living cells. ANIMO transforms the domain-specific models defined by biologists to UPPAAL models; then the results of the model checking process are presented back in a domain-specific fashion. The transformations in ANIMO are implemented in a general-purpose language, i.e., Java, whereas the case studies reported in this paper use languages specifically designed for model management tasks.

Frost et al. [14] have introduced a tool for static analysis of timing properties of Java programs. The tool transforms the domain-specific model, which consists of the program, the virtual machine, and the hardware models, to an UPPAAL timed-automata model. The paper does not report any use of MDE techniques.

A toolset to support design-space exploration of embedded systems was introduced by Basten et al. [2]. It aims for the reuse of models between various domains, by providing Java libraries to read design models written in its own specification language and then transform them for use with other tools including Uppaal for design-space exploration. If the toolset needs to support a new tool, one has to implement new transformations using these libraries. Using a language not specifically designed for such transformations leads to challenges in maintaining the toolset, which are in fact stated as a future direction of research.

In the study by Fakih et al. [13], a tool named sdf2ta has been introduced for analyzing timing bounds of SDF graphs. The tool takes an SDF graph defined using the tool SDF3 and a hardware model defined separately, and automatically generates an Uppaal timed-automata model. Similar to our tooling choice, they have used EMF for the implementation of sdf2ta, however, it is not reported how the generation of the timed-automata model is achieved.

Herber and Glesner [19] proposed a framework to verify hardware-software co-designs using timed automata. It translates the co-design implemented in SystemC to Uppaal's timed automata format. This translation is automatically achieved by the SystemC Timed Automata Transformation Engine (STATE) that is specifically designed for SystemC-to-Uppaal transformations. STATE is implemented directly in Java, which limits interoperability with other tools.

In the work by Hartmanns and Hermanns [18], a toolset has been introduced to facilitate the reuse of various model checkers targeting the stochastic hybrid automata formalism. The toolset uses a high-level compositional modeling language that serves as an interoperability point among existing languages and tools. Conceptually, this language is similar to a metamodel and the transformations from/to this language are implemented using traditional compiler techniques.

The study by Glatz et al. [16] uses model checking to test distributed control systems. The authors mathematically define a mapping from concepts in the control systems domain to the timed-automata domain. In their approach, they suggest implementing this mapping as a translation between the XML formats of these domains, which can be seen as a textual model-based transformation.

6 Conclusions

We have demonstrated the use of MDE in the development of software bridging tools that use Uppaal as a back-end analysis tool. Our approach uses metamodels as the foundation to translate domain-specific concepts into timed-automata models and queries; the results delivered by Uppaal are similarly translated back to the original domain, providing experts with access to formal analysis techniques without requiring additional training. We have presented five case studies in different domains to demonstrate how our approach has been applied in practice with the aim of a higher level of interoperability, faster software development and easier maintainability.

The principles we have presented here can be applied to formalisms and analysis tools different from timed automata and Uppaal by replacing the central metamodels uta, uqu and utr with suitable counterparts. Thus we expect our approach to be generally applicable in the development of more software bridging tools which act between DSLs and formal methods.

References

1. Ahmad, W., Yildiz, B.M., Rensink, A., Stoelinga, M.: A Model-Driven Framework for Hardware-Software Co-design of Dataflow Applications. In: Proc.6th Int. Wks.

Cyber Physical Systems. Design, Modeling, and Evaluation. pp. 1–16. Cham (2017)
2. Basten, T., Hamberg, R., Reckers, F., Verriet, J.: Model-Based Design of Adaptive Embedded Systems. Springer Publishing Company (2013)
3. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Petterson, P., Yi, W., Hendrink, M.: Uppaal 4.0. In: Proc. 3rd Int. Conf. Quantitative Evaluation of Systems (QEST). pp. 125–126 (2006)
4. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal Scheduling Using Priced Timed Automata. SIGMETRICS Perform. Eval. Rev. 32(4), 34–40 (Mar 2005)
5. Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools, LNCS, vol. 3098, pp. 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
6. Brandt, J.: Understanding attacks: Modeling the outcome of attack tree analysis. In: 25th Twente Student Conference on IT. vol. 25. University of Twente (2016), BSc. Thesis; see `http://referaat.cs.utwente.nl/conference/25/paper`
7. Bulychev, P., David, A., Larsen, K.G., Mikučionis, M., Poulsen, D.B., Legay, A., Wang, Z.: Uppaal-SMC: Statistical Model Checking for Priced Timed Automata. In: Proc. 10th Wks. Quantitative Aspects of Programming Languages (2012)
8. Dugan, J.B., Bavuso, S.J., Boyd, M.A.: Fault trees and sequence dependencies. Proc. Annu. Reliability and Maintainability Symposium pp. 286–293 (Jan 1990)
9. Dziwok, S., Gerking, C., Heinzemann, C.: Domain-specific Model Checking of MechatronicUML Models Using Uppaal. Tech. Rep. tr-ri-15-346, Paderborn University (Jul 2015)
10. Dziwok, S., Pohlmann, U., Piskachev, G., Schubert, D., Thiele, S., Gerking, C.: The MechatronicUML design method: Process and language for platform-independent modeling. Tech. Rep. tr-ri-16-352, Software Engineering Dep., Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute (Dec 2016), version 1.0
11. Eclipse Foundation, Inc.: XTend – modernized Java. `https://www.eclipse.org/xtend/index.html`
12. Emerson, E.A., Clarke, E.M.: Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. Sci. Comput. Program. 2(3), 241–266 (1982)
13. Fakih, M., Grüttner, K., Fränzle, M., Rettberg, A.: State-based Real-time Analysis of SDF Applications on MPSoCs with Shared Communication Resources. J. Syst. Archit. 61(9), 486–509 (Oct 2015)
14. Frost, C., Jensen, C., Luckow, K.S., Thomsen, B.: WCET Analysis of Java Bytecode Featuring Common Execution Environments. In: Proc. 9th Int. Wks. Java Technologies for Real-Time and Embedded Systems. pp. 30–39. ACM (2011)
15. Gerking, C., Schäfer, W., Dziwok, S., Heinzemann, C.: Domain-specific model checking for cyber-physical systems. In: Proc. 12th Wks. Model-Driven Engineering, Verification and Validation (MoDeVVa 2015). Ottawa (Sep 2015)
16. Glatz, B., Cleary, F., Horauer, M., Schuster, H., Balog, P.: Complementing testing of IEC61499 function blocks with model-checking. In: Proc. 12th IEEE/ASME Int. Conf. Mechatronic and Embedded Systems and Applications (MESA) (2016)
17. Greenyer, J., Rieke, J.: Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In: Revised Selected and Invited Papers 4th Int. Symp. Applications of Graph Transformations with Industrial Relevance (AGTIVE). LNCS, vol. 7233, pp. 222–237 (2012)
18. Hartmanns, A., Hermanns, H.: The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In: Proc. 20th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems. pp. 593–598 (2014)
19. Herber, P., Glesner, S.: A HW/SW Co-verification Framework for SystemC. ACM TECS 12(1s), 61:1–61:23 (Mar 2013)

20. Huistra, D.: A unifying model for attack trees. Research Project, University of Twente http://essay.utwente.nl/69399/ (2015)
21. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Proc. 1st Int. Conf. Theory and Practice of Model Transformations (ICMT). pp. 46–60 (2008)
22. Kordy, B., Piètre-Cambacédès, L., Schweitzer, P.: DAG-based attack and defense modeling: Don't miss the forest for the attack trees. Comput. Sci. Review 13-14, 1–38 (2014)
23. Kumar, R., Stoelinga, M.: Quantitative Security and Safety Analysis with Attack-Fault Trees. In: Proc. IEEE 18th Int. Symp. High Assurance Systems Engg. (HASE). pp. 25–32 (Jan 2017)
24. Kumar, R., Ruijters, E., Stoelinga, M.: Quantitative Attack Tree Analysis via Priced Timed Automata. In: Proc. 13th Int. Conf. Formal Modeling and Analysis of Timed Systems (FORMATS). pp. 156–171 (2015)
25. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE 75(9), 1235–1245 (Sep 1987)
26. Mohagheghi, P., Dehlen, V.: Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Proc. European Conf. Model Driven Architectures - Fountations and Applications (ECMDA-FA). LNCS, vol. 5095 (2008)
27. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.2. OMG Document Number formal/2015-02-01 (http://www.omg.org/spec/QVT/1.2) (Feb 2015)
28. Ravn, A.P., Srba, J., Vighio, S.: A Formal Analysis of the Web Services Atomic Transaction Protocol with Uppaal. In: Proc. 4th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). LNCS, vol. 6415, pp. 579–593 (2010)
29. Ruijters, E., Guck, D., Drolenga, P., Stoelinga, M.: Fault maintenance trees: reliability contered maintenance via statistical model checking. In: Proc. IEEE 62nd Annu. Reliability and Maintainability Symposium (RAMS). IEEE (Jan 2016)
30. Ruijters, E., Stoelinga, M.: Fault tree Analysis: A survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Review 15–16, 29–62 (2015)
31. Schivo, S., Scholma, J., Wanders, B., Camacho, R.A.U., van der Vet, P.E., Karperien, M., Langerak, R., van de Pol, J., Post, J.N.: Modeling Biological Pathway Dynamics With Timed Automata. IEEE J. Biomed. Health Inform. 18(3), 832–839 (May 2014)
32. Schivo, S., Scholma, J., van der Vet, P.E., Karperien, M., Post, J.N., van de Pol, J., Langerak, R.: Modelling with ANIMO: between fuzzy logic and differential equations. BMC systems biology 10(1), 56 (2016)
33. da Silva, A.R.: Model-driven engineering: A survey supported by the unified conceptual model. Comput. Languages, Systems & Structures 43, 139–155 (2015)
34. Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G.: Metamodelling. In: Model-Based Engineering of Embedded Real-Time Systems, pp. 57–76. Springer (2010)
35. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse modeling framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)
36. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-driven software development: technology, engineering, management. John Wiley & Sons (2006)
37. Yildiz, B.M., Bochisch, C.M., Rensink, A., Aksit, A.: An MDE approach for modular program analyses. In: Proc. Modularity in Modelling Workshop (2017)
38. Yildiz, B.M., Rensink, A., Bockisch, C., Aksit, M.: A Model-Derivation Framework for Software Analysis. In: Proc. 2nd Wks. Models for Formal Analysis of Real Systems (MARS) (2017)