A Model-Derivation Framework for Software Analysis

Bugra M.Yildiz[1] Arend Rensink[1] Christoph Bockisch[2] Mehmet Aksit[1]

[1]Formal Methods and Tools Group, University of Twente

[2]Faculty of Mathematics and Computer Science, Philipps-Universität Marburg

{b.m.yildiz, arend.rensink, m.aksit}@utwente.nl

bockisch@mathematik.uni-marburg.de

Model-based verification allows to express behavioral correctness conditions like the validity of execution states, boundaries of variables or timing at a high level of abstraction and affirm that they are satisfied by a software system. However, this requires expressive models which are difficult and cumbersome to create and maintain by hand. This paper presents a framework that automatically derives behavioral models from real-sized Java programs. Our framework builds on the EMF/ECore technology and provides a tool that creates an initial model from Java bytecode, as well as a series of transformations that simplify the model and eventually output a timed-automata model that can be processed by a model checker such as UPPAAL. The framework has the following properties: (1) consistency of models with software, (2) extensibility of the model derivation process, (3) scalability and (4) expressiveness of models. We report several case studies to validate how our framework satisfies these properties.

1 Introduction

One of the main challenges in developing a software system is to ensure that it fulfills the specifications. Validation of software systems by testing is generally considered to be a labor-intensive and tedious task [5]. For this reason, model-based verification techniques have been introduced, which aim at verifying software systems through the use of models, instead of testing at the implementation level [23]. Such approaches naturally require the existence of expressive models of the systems being considered.

Unfortunately, deriving expressive models for software systems for the purpose of verification is not a trivial task [21]. Firstly, models are typically defined through a manual effort. The modeler must be an expert in the adopted modeling technique, must have a deep understanding of the software being modeled and must have skills for abstracting away the unnecessary details. These challenges make the model building process a labor-intensive and error-prone task. As a result, models of the same system can vary depending on the skills and preferences of the modeler. Secondly, software systems evolve continuously. Models must be maintained in parallel or else they become outdated [18]. Keeping models consistent with software is tedious work.

To counter these challenges, we propose the use of automatic model derivation from program code. This paper presents a framework that automatically derives models from Java programs that can be used by a model checker. There are a number of similar proposals [3, 7, 4, 6] that share the overall aim of our framework. However, no evidence is reported that they are capable of handling large programs. We show that our framework can, for instance, derive a model of a Java program with around 1500 classes in a reasonable time. (Note that the actual analysis of the generated models is not the focus of this paper.)

We have built the implementation of this framework upon the MDE technology called Epsilon/EMF [16]. As part of our framework, we have developed an ECore metamodel of Java bytecode. As target formalism for the derived models, we have chosen timed automata, since we want to conduct (among other

H. Hermanns, P. Höfner (Eds.): Models for
Formal Analysis of Real Systems (MARS 2017)
EPTCS 244, 2017, pp. 217–229, doi:10.4204/EPTCS.244.9

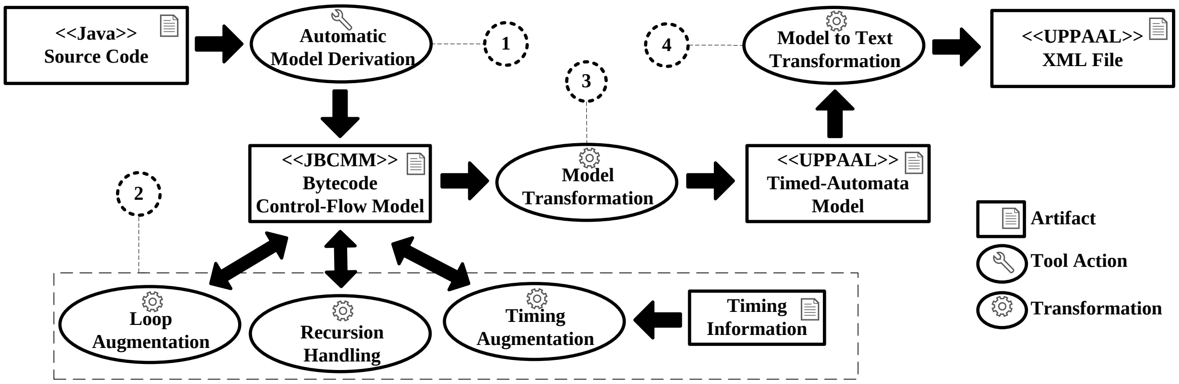

Figure 1: The automatic model-derivation framework

things) timing analysis in future work of our project [22]. Our framework produces models compatible with the UPPAAL model checker [2].

Overview. The elements of the framework are shown in Figure 1. The actions, described in some more detail in Section 3, are as follows:

- Action 1 automatically derives a bytecode control-flow model of the software system from its bytecode.
- Action 2 modifies and enriches the bytecode control-flow model with properties serving, e.g., as a prerequisite for the model-based timing analysis: *a*) loop augmentation, to detect the loops in the control-flow model and annotate them with repetition limits; *b*) recursion handling, to modify the control-flow model for feasible model checking by detecting the recursive calls via a derived call graph and handling them, *c*) timing augmentation inserts the predictions of execution times into the control-flow model.
- Action 3 transforms the control-flow model to a timed-automata model. This is shown on an example in Section 2.
- Action 4 transforms this model to the input format of the model checker.

Contribution and roadmap. We claim the following benefits for our model derivation framework:

1. *Consistency*: Our framework ensures that the derived models are consistent with the original software. Since all the individual actions of the framework are fully automated, it is easy to re-create models after changes occur in the code. We elaborate on this in Section 3.
2. *Extensibility*: Our framework can be extended in a systematic way to adapt to various analysis needs. We discuss this further in Section 4, by providing some example extensions.
3. *Scalability*: Our framework can handle large Java programs in an acceptable time span. The time needed to check a model is known to grow exponentially with the size of the model. For this reason, Action 2 of our framework provides model transformations that simplify the control-flow model before the UPPAAL model is derived in Action 3. Furthermore, we generate timed-automata models such that optimizations applied by UPPAAL are applicable. We elaborate this in Section 5.

4. *Expressiveness*: The derived models are expressive enough for analysis purposes. Essentially, bytecode instructions are transformed into locations, in such a way that UPPAAL queries can be formulated about the timing properties of the model that reflect real properties of the original system. We discuss this further in Section 6.

For more detailed information, please refer to our technical report [24]. The link to the repository of the framework is available at [20].

Related work. There are a number of automatic model derivation tools for analysis of software systems [3, 7, 4, 6], which share the overall aim of our framework. However, no evidence is reported for these tools showing that they are capable of handling large programs.

Corbett et al. introduced Bandera [6], an integrated collection of analysis components for Java programs. Bandera produces finite-state models in the input language of several verification tools from the source code of Java programs. One major difference with our framework is that timing analysis is not aimed by Bandera through its generated models. Another difference is that Bandera derives its models from the source code but not from the bytecode of Java programs. As a result of this, it does not offer to extend its analysis to include the third-party components whose source code is not available or it does not analyze other non-Java language programs which compile to bytecode. Lastly, the study reports the timing performance of model-derivation for a small example, however there is no information reported on the scalability of the model-derivation process for large code sizes.

Frost et al. presented the tool called TetaJ [7] for static analysis of Java programs using model checking and Luckow et al. presented the tool called TetaSARTS in [14] as a continuation of TetaJ to address schedulability of Java programs. They use three layers of models that are presented as separate templates in a timed-automata model in UPPAAL: the program, the virtual machine and the hardware. The program model is automatically derived from the bytecode and the mapping of bytecode elements to the timed-automata model is done in a similar approach as we do. Although their tool includes some optimizations coming with it to reduce the state-space size, it does not offer mechanisms for extensions. They report the performance results of the model checking some examples up to 18 classes and 44 methods, but the papers include no information about the performance and scalability of the automatic model derivation process for large programs.

Bucaioni provides a tool developed using MDE techniques to generate models of a vehicular embedded application in order to perform timing analysis [4]. However, neither are there an explicit extension mechanisms offered nor do the authors report about the scalability of timing performance.

Bernat et al. proposed a WCET analysis scheme based on Java bytecode and a tool called Javelin to support this scheme [3]. They use static method calls to inject the missing data information in the bytecode such as loop iterations. They focus on WCET analysis only and do not use model checking for this purpose. Javelin does not offer any extension mechanisms. There is also no explanation on how inheritance and polymorphism are handled and nor do they report on the scalability of their approach.

2 Model Derivation Framework

The core transformation of our framework is Action 3, which derives an UPPAAL model from an (enriched) bytecode control-flow model. We explain the basics of this transformation very briefly; for more information please refer to [24].

A timed-automata model in UPPAAL consists of synchronized instances of templates, where each template defines a single timed automaton; the synchronization is based on transition labels.

```
 1 import java.util.Random;
 2
 3 public class Main
 4 {
 5   public static void main(String[] args)
 6   {
 7     Math math = new Math();
 8     int rdmNmbr=new Random().nextInt();
 9     System.out.println(rdmNmbr +
10       ” is prime:” + math.isPrime(rdmNmbr));
11     System.out.println(rdmNmbr +
12       ” is even:” + math.isEven(rdmNmbr));
13   }
14 }
```

```
 1 public class Math
 2 {
 3   public boolean isPrime(int nmbr)
 4   {
 5     int curDivider = 2;
 6     boolean isPrime = true;
 7     while(curDivider < nmbr)
 8     {
 9       if(nmbr % curDivider == 0)
10       {
11         isPrime = false;
12         break;
13       }
14       curDivider++;
15     }
16     return isPrime;
17   }
18
19   public boolean isEven(int nmbr)
20   {
21     if( nmbr % 2 == 0)
22     {
23       return true;
24     }
25     else {
26       return false;
27     }
28   }
```

Figure 2: Example Java source code

- The UPPAAL model derived from a Java program consists of a template for every Java class, plus one global template to maintain a global clock and kick off the program.
- Each class-derived template consists of a loop for each method, starting with an action corresponding to a call of that method, expressed as `<class_name>#<method_signature>#call?` and ending with the corresponding `<class_name>#<method_signature>#return!`.
- The locations correspond to nodes of the bytecode control flow graph and are labeled `l_<line_number>_<index>`; the transitions correspond to flow graph edges.
- If a location corresponds to a method call, each incoming transition is labeled with the call actions, and each outgoing transition with the return actions.
- A local clock per template, `lc`, is used in invariant and guard expressions to force the system to spend between `tlb_<location_name>` and `tub_<location_name>` time units at a particular location.

We use a running example to demonstrate the framework. Figure 2 shows the source code of the example Java program. Here, `Main.main` generates a random integer, then calculates first if this integer is prime and afterwards if it is even by using methods of the `Math` class. We consider this a representative example, since it includes typical imperative object-oriented language structures such as loops, branching points and method invocations.

The timed-automata model derived from this program is shown in Figure 3. The first template keeps the global clock, calls the main method of the program and goes to location `finish` when the call returns. The second and third templates correspond to the `Main` and `Math` classes, respectively. For instance, location `l_7_59` in the second template (see label `A`) corresponds to the bytecode instruction compiled from line 7 of the class `Main` and represents an instruction allocating memory for the object which is constructed at this line. The time spent on the location `l_7_59` is limited between `tlb_l_7_59` and `tub_l_7_59` time units, which are specified by the outgoing edge guard and the location invariant, respectively. The method call for the `Math` object construction is mapped to three locations `l_7_4_calling`, `l_7_4_waiting` and `l_7_4_returning`. The call and return of this method are transformed as a pair of synchronization actions, which are expressed as `Math#init#call!` and `[`keywords=]Math# init#return?, on the corresponding edges. These synchronization actions pass the control flow to the `Math` template for the method execution, then take the control back as soon as the method finishes and returns.

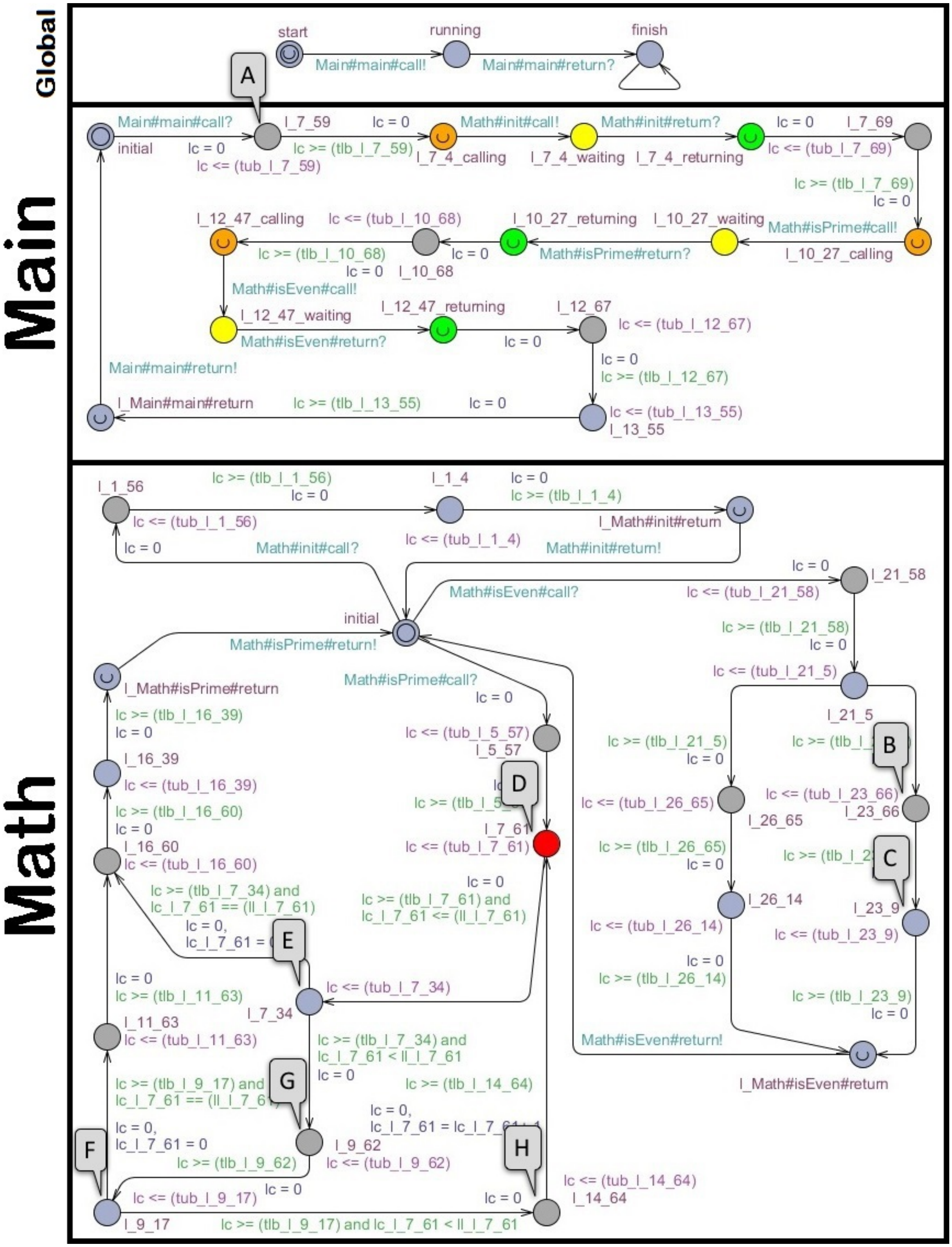

Figure 3: Generated timed-automata model from Figure 2

For this example, suppose that we want to check if the main method finishes its execution within `x` time units. This corresponds to checking whether the timed-automata model always reaches `finish` in less than `x` time units, and is expressed by the UPPAAL query

```
A[] main.finish and globalClock <= x
```

This query means that for all paths (expressed through `A`) that begin in the `start` location and terminate at the `finish` location, the condition that `globalClock` is less than or equal to `x` must hold (expressed through `globalClock <= x`).

Suppose that the time spent at each location is 1, the loop limit is 5 and `x` is set to 20. When UPPAAL evaluates the query, it reports that the property represented by the query is not satisfied. This means that there is an execution path taking more than `x` time units. In this case, the reported trace visits all the locations in the timed-automata model except the ones with the labels `B` and `C`. This corresponds to an execution of the program where the `else` case of the method `isEven` is executed (lines 25–27).

3 Consistency

In this section, we explain each of the actions shown in Figure 1 in more detail. By applying these actions, one can easily re-create timed-automata models for analysis after any changes occur in the code. This helps to have models that are consistent with the software at the time of analysis.

Action 1: Java Source Code to Bytecode Control-Flow Models. To implement Action 1, we have developed JBCPP, which we publish as an Eclipse plug-in. JBCPP generates models from Java bytecode that include an explicit representation of control flow. The models conform to JBCMM (Java Bytecode Metamodel)[1], a dedicated metamodel for Java bytecode. The root of a JBCMM model is called `Project` and it contains models of all classes whereby the main class is distinguished. A class model in turn contains models for its methods which contain models for all bytecode instructions. The control flow between instructions is modeled through control flow edges.

Action 2: Enriching Bytecode Control-Flow Models. The model resulting from Action 1 faithfully reflects the bytecode. This means, on one hand, it covers all the information needed to execute the program; on the other hand, some information is only presented implicitly. The purpose of Action 2 is therefore twofold. First, details irrelevant for the analysis should be abstracted away. Second, additional information, like loop limits or timing information, should be injected.

Action 3: Bytecode Control-Flow Models to UPPAAL Models. The core part of the framework is the transformation definition from JBCMM models to UPPAAL timed-automata models. This corresponds to Action 3 in Figure 1. The UPPAAL models conform to a metamodel developed by the Software Engineering Group at the University of Paderborn [15]. This metamodel consists of all the elements and their relationships of any timed-automata model definable using the UPPAAL tool. The metamodel contains the conceptual elements such as locations, edges and clocks; it also contains syntax graphs for the C-like expressions supported by the UPPAAL model checker.

[1] While the term *JBCMM* refers to the metamodel itself, we use *JBCMM model* and *bytecode control-flow model* terms interchangeably to refer to an instance of JBCMM.

Action 4: Generating UPPAAL Files. The model conforming to the UPPAAL metamodel itself is not directly processable by the UPPAAL model checker. To make the model usable by the tool, we have implemented a model-to-text transformation, which takes a UPPAAL model as the input and transforms it into an XML file compatible with the UPPAAL XML format. The transformation corresponds to Action 4 in Figure 1.

4 Extensibility

One can extend the framework in various ways depending on the analysis needs using the following mechanisms: introducing new models with related metamodels or new transformations; extending existing transformations or metamodels; and changing the transformation application order accordingly. In this section, we present several extensions as examples.

4.1 Loop Augmentation

When starting with a piece of source code with loops, the generated timed-automata model will contain cycles. If the number of repetitions would not be limited in the UPPAAL model, then the model checker can create unbounded execution paths. In particular this is true if timing is considered in the analysis, as the global clock value would increment infinitely. As a consequence, queries over all execution paths, such as worst case execution time (WCET), will generally not give any meaningful results. To compensate this, we apply two steps:

Loop Detection: The loop detection takes a JBCMM model as input, detects the loops in it and generates an extended JBCMM model as output which additionally represents loop-related information such as instructions in the loop and loop limit. We use a dominator analysis [13] for loop detection.

The framework provides an interface through the extension of JBCMM to allow its user to insert loop limits. The loop limits to be inserted can be obtained using manual annotations [3] or automated analysis techniques [12]. Currently, we use a default value as the loop limit for all the loops.

Insertion of Loop Information: Once the loops are detected and the extended JBCMM model is generated, loop-related information is inserted into the timed-automata model during Action 3. For this purpose, we have implemented a transformation module, which is called *Loop Information Insertion*. This module extends the transformation from bytecode control-flow models to UPPAAL timed-automata models (Action 3) by simply adding one additional transformation rule for loops and reusing all the other rules in the transformation. As an example how this module works, consider Figure 3. The location `l_7_61` (label `D`) is the head of the loop. The counter and the repetition limit are named after the loop head as `lc_l_7_61` and `ll_l_7_61`, respectively. The locations `l_7_34` (label `E`) and `l_9_17` (label `F`) are the exiting points from the loop. The locations `l_9_62` (label `G`) and `l_14_64` (label `H`) are the remaining loop nodes. On the back edge, which is from the location `l_14_64` (label `H`) to the head location, the loop counter is incremented. The exiting edges are guarded by the condition `lc_l_7_61==ll_l_7_61`, which checks if the loop counter already reached the limit; the continuing edges are guarded by the condition `lc_l_7_61 <ll_l_7_61`, which checks if the loop counter is still below the limit. The loop counter is reset on the exiting edges.

4.2 Recursion Handling

A template instance in the UPPAAL model represents a stack frame in a Java program execution. The number of template instances for each template needs to be known before using the model checker for the timing analysis since UPPAAL does not allow to create new instances on the fly. If we start with fewer instances than the possible call stack size, then we will end up in a deadlock state. Therefore, our framework currently handles recursive calls by removing direct recursion (calls to the method containing the invocation) and reporting other forms of recursion.

For removing the direct recursive calls, we have implemented a transformation called *Recursion Removal*. The *Recursion Removal* transformation takes a JBCMM model and outputs a new JBCMM model in which the direct recursive call instructions are replaced by some dummy instructions.

For reporting other forms of recursion, we have implemented a transformation to derive the call graph of the JBCMM model. A cycle in such a graph shows the existence of a recursive call. The transformation generates a call graph from a JBCMM model and reports any recursive call structures by analyzing this call graph.

4.3 Timing Augmentation

The *Timing Augmentation* transformation takes a JBCMM model as input and enhances it with the timing information, i.e., the minimum and maximum time spent for each instruction's execution. Currently, we use default values for the timing information. The framework provides an interface through slots in JBCMM models for the timing values to allow its user to insert timing information. The user can acquire timing information to be inserted by using various techniques such as profiling [17] or JVM Timing Models [10, 9].

5 Scalability

In this section, we will present evidence for the scalability of the framework. For our purpose, the framework should be able to cope with realistic software sizes. To assess whether this is the case, we have chosen three real-life open source Java programs of different sizes as input.

Table 1 shows the characteristics of their derived JBCMM models. The columns A through E show the counts of corresponding elements in the model. Column F shows the count of the method call instructions whose invokable method implementations are included in the model. Column G shows the total number of possible method implementations invocable by method calls. Column H shows the count of the return instructions. The size of a program can be determined by the number of model elements that its JBCMM model contains: A+B+C+D+E.

	Class (A)	Method (B)	Loop (C)	Instruction (D)	Edge (E)	Method Call (F)	Method Invoc. (G)	Return Instr. (H)	Total (A+B +C+D+E)
LiveGraph	131	350	33	11795	11,740	665	687	440	24,049
Groove Generator	930	5,392	756	99,738	98,634	9,790	12,718	7,114	205,450
Groove Simulator	1,482	9,232	1,454	203,030	203,071	20,198	252,72	12,101	418,269
Weka	1,041	83,22	4,072	367,774	374,854	30,124	108,570	10,820	756,063

Table 1: Characteristics of the JBCMM Models of the Example Programs

LiveGraph is a real-time graph and chart plotter to represent large amounts of data [11]. *Groove* is a tool for modeling and analyzing object-oriented systems through graphs and graph transformations [19]. We have examined the *Simulator* and *Generator* components of Groove. *Weka* offers a large collection of machine learning algorithms with pre-processing of the data and visualization of the results

[8]. Although the class and method counts are close to the Groove components, the total model size is around 1.8 times as much as the Groove Simulator due to the large instruction and edge counts.

5.1 Scalability of the Framework Actions

We define the scalability of the framework actions as the ability to get acceptable time performance measurements with the increase in the input size.

Prediction on outcome: Although there is an expectancy to observe a linear dependency between the timing performance and the derived model sizes for Action 1, the performance can depend on various factors such as the number of processed classes on the classpath (which is different from the number of classes included in the model) in the example Java programs. For Action 2, we have implemented a practically fast implementation of the dominator analysis algorithm of the complexity $O(D^2)$ for loop detection, so we expect the timing performance to have at worst a quadratic dependency with respect to the model size (but in practice, it can run faster) [13]. We expect to get a linear association between the timing performance and the model sizes for the actions 3 and 4 since these model transformations are direct mappings of input elements to output elements and do not have any special algorithmic computations used, unlike Action 2.

Outcome: We have applied the actions 1 through 4 to obtain the UPPAAL textual model of each program. The timing results of these experiments are presented in Table 2. Each action has been repeated 10 times for each program, the table shows the averages. The experiments have been carried out using an Intel i7-3520M 2.90 GHz CPU with 4 cores and 16 GB RAM.

	Model Size	Automatic Model Derivation (Action 1)	Loop Augmentation (Action 2)	JBCMM to UPPAAL Transformation (Action 3)	Model-to-Text Transformation (Action 4)	Total Time (sec)
LiveGraph	24,049	18	51	12	35	117
Groove Generator	205,450	1,414	86	194	364	2,058
Groove Simulator	418,269	1,480	300	538	977	3,295
Weka	756,063	764	803	1,069	2,402	5,037

Table 2: Experiment Results (in seconds, averaged over 10 runs)

Evaluation of the outcome: For Action 1, the results show no particular relationship with the derived model sizes. We tried to find a correlation with various possible factors related to this Action, but currently we cannot say what is the determining factor. Nevertheless, the timing performance is still acceptable for large projects like Weka.

Although the algorithmic complexity of the dominator analysis algorithm that we have used is quadratic with respect to the number of instructions in the input models, the loop augmentation transformation with the practically fast implementation of the dominator analysis algorithm still runs only around 13 minutes for the largest of our input programs. The last two columns of Table 2 show the timing performance of the actions 3 and 4. The figures support the hypothesis that the performance of the model-to-model and model-to-text transformations are linear with respect to the input size of the models. The experiments show that the timing performance of the framework scales well with respect to the varying code sizes.

5.2 Scalability of Model Checking of the Generated UPPAAL Models

Although model checking itself is outside the scope of this paper, let us discuss some ways to improve scalability of model checking of the generated models.

It is a major challenge to adjust the correct abstraction level for models to avoid the state-space explosion problem in model checking. The more detailed the models are, the more accurate the results one can get. However, increasing the detail level of models can cause intractable state-space sizes. Raising the abstraction level, in our approach, can be achieved by extending our framework with new transformations. We have implemented such an extension that abstracts away some details by grouping nodes.

Furthermore, UPPAAL provides some generic mechanisms to reduce the state-space size or to optimize the state-space generation/exploration by removing redundancy. We have defined our strategy for transforming JBCMM models to UPPAAL timed automate models such that these mechanisms are applicable. Both optimizations are detailed below, followed by a discussion of their effects.

Node Grouping: This extension allows to reduce the state-space size by decreasing the number of locations in the timed-automata model, by replacing sequences of bytecode instructions (connected with control-flow edges) with a *group* instruction that accumulates the timing characteristics of its instructions. Method calls and branches are not grouped since they affect the execution flow.

Symmetry Reduction: When a template instance T_A has to synchronize with an instance of template T_B, it needs to choose with which instance it wants to synchronize. UPPAAL generates the same state-space for any choice of instances of template T_B if all instances of the same template are identical. For such a case, UPPAAL can be guided to not to generate redundant states, which is called symmetry reduction optimization. For this reason, we define templates such that no extra states are introduced for identical cases, enabling the symmetry reduction optimization of UPPAAL in our default transformation definition from JBCMM models to UPPAAL models.

Experiment setup and outcome. We have used the example Java program given in Section 3 to test how much Node Grouping and Symmetry Reduction help to reduce the state space. To check the maximum possible size of the state-space of the UPPAAL model, we have chosen a query that checks whether synchronization points in UPPAAL can be blocking each other. As we only use synchronization actions to represent method invocation, they are properly nested in our case and deadlocks cannot appear. Therefore, such a query will lead to exploring the whole state space of the model. For this example case, we have achieved a reduction of around 80% in the size of the generated state-space when both optimizations have been applied. The details of this experiment can be found in the technical report [24].

6 Expressiveness

We have conducted an example analysis of LiveGraph to show how we can use the models of real-life programs generated by our framework for analysis. LiveGraph stores the data to be plotted in the `DataCache` class. The changes in the instances of this class trigger firing of an event via the `fireEvent` method of the same class, which notifies observers.

As an example analysis, we want to check if resetting of data labels, `DataCache.resetLabel`, eventually triggers firing of an event. This kind of check is common in program analyses. The following query is the formulation of this property (note that we simplified the names for readability):

```
(exists (id:DataCacheTemplateId) DataCacheTemplate(id).l_resetLabelF) -->
(exists (id:DataCacheTemplateId) DataCacheTemplate(id).l_fireEventF)
```

The location `l_resetLabelF` is the first to be visited when `resetLabel` is called. Similarly, `l_fireEventF` is the first to be visited when `fireEvent` is called. Both locations are in the same template, DataCacheTemplate, which corresponds to the `DataCache` class. The `-->` sign specifies the *eventually leads to* statement. The **`exists`** statements are used to make sure that these locations can be reached by any instance of the template. The query asks if the visit of the location `l_resetLabelF` eventually leads to the visit of the location `l_fireEventF`, which corresponds to what we want to check.

This example shows that the models generated by the framework provide a practical way for conducting desired analyses through defining small and intuitive queries in a simple way.

7 Conclusion

In this paper, we have presented a framework that derives models from Java programs in an automatic way for analysis. We have shown that the framework provides the following features:

1. *Consistency*: Since the model derivation process consists of multiple actions that are fully automated, it is easy to re-create models after changes occur in the code. Therefore, there is no need to separately maintain code and models, and this prevents inconsistency.
2. *Extensibility*: Due to adoption of MDE techniques, users can adapt and extend the framework conveniently. Metamodels as well as transformations can easily be extended. We have demonstrated this by developing four different transformations to enrich the models derived by our framework.
3. *Scalability*: We have derived models with our framework from large Java programs with up to 1,482 classes and up to 367,774 bytecode instructions. The largest model that we derived in our study contained 756,063 nodes. The longest time our framework took to derive a model was 84 minutes, which is still sufficient to re-create models on a nightly basis. For a smaller project of 131 classes and 11,795 instructions, model derivation took only 2 minutes.
4. *Expressiveness*: We have described a translation scheme to create UPPAAL timed-automata models from the ECore-based model that we derive. The timed-automata model faithfully reflects the execution semantics of the Java program, in particular for method invocation and branching instructions. The translation scheme is also provided as model transformation as part of our framework.

In addition to these features, the automation of the model derivation process reduces errors that can be caused by manual processes; and eliminates variations in models caused by subjective decisions.

With our Java bytecode metamodel, we have presented a generic way of modeling compiled Java code. By using EMF-technology, we facilitate a high degree of interoperability between multiple static Java bytecode analyses.

Future work. One direction for future work is to include support for data-flow analysis to increase the precision of derived information such as loop limits or timing values.

Another direction is to include stochastic information to allow users of the framework to do probabilistic model checking with the derived timed-automata models.

Yet another direction is to provide support for concurrent Java programs. The instances of templates in UPPAAL models are time automata (processes) running in parallel, by definition. In the future, we plan to adapt the model transformation in Action 3 (from bytecode models to UPPAAL models) to treat concurrency-related constructs in bytecode models, such as new tread creations or synchronization directives, specifically to analyze concurrent programs by benefiting from the parallelism of time automata in UPPAAL models.

Finally, we plan to provide a convenient front-end for using UPPAAL together with models created by our framework; this will allow to perform analyses and interpret results in terms of the source code rather than locations in the timed-automata model.

References

[1] R. Alur & D. L. Dill (1994): *A Theory of Timed Automata*. *Theoretical Computer Science* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[2] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson & W. Yi (1995): UPPAAL— *a Tool Suite for Automatic Verification of Real–Time Systems*. In: *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, *Lecture Notes in Computer Science* 1066, Springer–Verlag, pp. 232–243, doi:10.1007/BFb0020949.

[3] G. Bernat, A. Burns & A. Wellings (2000): *Portable Worst-Case Execution Time Analysis Using Java Byte Code*. In: *12th Euromicro Conference on Real-Time Systems, 2000. Euromicro RTS 2000.*, IEEE, pp. 81–88, doi:10.1109/EMRTS.2000.853995.

[4] A. Bucaioni (2015): *Raising Abstraction in Timing Analysis for Vehicular Embedded Systems through Model-Driven Engineering*. In: *Software Technologies: Applications and Foundations*, doi:10.13140/RG.2.1.5120.0883.

[5] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov & G. Candea (2010): *Cloud9: A Software Testing Service*. *ACM SIGOPS Operating Systems Review* 43(4), pp. 5–10, doi:10.1145/1713254.1713257.

[6] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby & H. Zheng (2000): *Bandera: Extracting Finite-State Models from Java Source Code*. In: *Proceedings of the 2000 International Conference on Software Engineering*, IEEE, pp. 439–448, doi:10.1145/337180.337234.

[7] C. Frost, C.S. Jensen, K. S. Luckow & B. Thomsen (2011): *WCET Analysis of Java Bytecode Featuring Common Execution Environments*. In: *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM, pp. 30–39, doi:10.1145/2043910.2043916.

[8] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann & I. H. Witten (2009): *The WEKA Data Mining Software: An Update*. *SIGKDD Explorations Newsletter* 11(1), pp. 10–18, doi:10.1145/1656274.1656278.

[9] E. Y. Hu, A. J. Wellings & G. Bernat (2003): *Deriving Java Virtual Machine Timing Models for Portable Worst-Case Execution Time Analysis*. In: *Workshops On The Move to Meaningful Internet Systems*, Springer, pp. 411–424, doi:10.1007/978-3-540-39962-9_48.

[10] J. M. Lambert & J. F. Power (2008): *Platform Independent Timing of Java Virtual Machine Bytecode Instructions*. *Electronic Notes in Theoretical Computer Science* 220(3), pp. 97–113, doi:10.1016/j.entcs.2008.11.021.

[11] LiveGraph (2016): *LiveGraph website*. Available at `http://live-graph.sourceforge.net`.

[12] P. Lokuciejewski, D. Cordes, H. Falk & P. Marwedel (2009): *A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models*. In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE Computer Society, pp. 136–146, doi:10.1109/CGO.2009.17.

[13] E. S. Lowry & C. W. Medlock (1969): *Object Code Optimization*. *Communications of the ACM* 12(1), pp. 13–22, doi:10.1145/362835.362838.

[14] Kasper Søe Luckow, Thomas Bøgholm, Bent Thomsen & Kim Guldstrand Larsen (2013): *TetaSARTS: A Tool for Modular Timing Analysis of Safety Critical Java Systems*. In: *Proceedings of the 11th International*

Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '13, ACM, New York, NY, USA, pp. 11–20, doi:10.1145/2512989.2512992.

[15] UPPAAL Metamodel (2016): *Software Engineering Group website, University of Paderborn*. Available at `https://www.hni.uni-paderborn.de/en/software-engineering`.

[16] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos & F. A. C. Polack (2009): *The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering*. In: *Proceedings of the 2009 14th IEEE Int. Conference on Engineering of Complex Computer Systems*, IEEE Computer Society, Washington, DC, USA, pp. 162–171, doi:10.1109/ICECCS.2009.14.

[17] P. Puschner & A. Burns (2000): *Guest Editorial: A Review of Worst-Case Execution-Time Analysis*. *Real-Time Systems* 18(2), pp. 115–128, doi:10.1023/A:1008119029962.

[18] J. Rech (2011): *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global.

[19] A. Rensink (2004): *The GROOVE Simulator: ATool for State Space Generation*, pp. 479–485. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-25959-6_40.

[20] Repository (2015): *The repository of the Model-Derivation Framework*. Available at `https://bitbucket.org/bmyildiz/model-derivation-framework.git`.

[21] R. G. Sargent (2011): *Verification and Validation of Simulation Models*. In: *Proceedings of the 2011 Winter Simulation Conference (WSC)*, doi:10.1109/WSC.2011.6147750.

[22] TIPS (2016): *Project website*. Available at `http://fmt.cs.utwente.nl/research/projects/aselsan/`.

[23] M. Utting & B. Legeard (2010): *Practical Model-based Testing: A Tools Approach*. Morgan Kaufmann.

[24] B. M. Yildiz, A. Rensink, C. M. Bockisch & M. Aksit (2015): *A Model-Derivation Framework for Timing Analysis of Java Software Systems*. Technical Report TR-CTIT-15-08, Centre for Telematics and Information Technology, University of Twente. Available at `http://eprints.eemcs.utwente.nl/26622/`.