A Model-Derivation Framework for Timing Analysis of Java Software Systems

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

[1] Formal Methods and Tools Group,
University of Twente,
7522 NB Enschede, The Netherlands
{b.m.yildiz,arend.rensink,m.aksit}@utwente.com
[2] Open Universiteit,
Postbus 2960, 6401 DL, Heerlen The Netherlands
bockisch@acm.org

Abstract. One of the main challenges in developing a software system is to assure that its properties fulfill the specifications. In the context of this paper, we are especially interested in timing properties. Model-based software verification is one of the approaches to achieve this. However, model-based verification requires expressive models of software systems and deriving such models is not a trivial task. Although there are a few model derivation tool proposals for the purpose of model-checking timing properties, these are dedicated tools supporting a selected set of verification techniques and as such they are not explicitly designed for coping with new demands. This paper presents a framework that derives models from Java programs in an automated way for analyzing timing properties. The framework has the following properties that are not provided by the previous proposals: (1) Efficiency in model development, (2) consistency of models with software, (3) expressiveness of models, (4) scalability and (5) extensibility of the model derivation process.

Keywords: Automatic model derivation, timing analysis, model checking, model transformation, model-driven software engineering.

1 Introduction

Today, software is adopted in many products and almost every product is produced by a system which adopts software. This fact makes software a crucial asset in our daily life. One of the main challenges in developing a software system is to assure that its properties fulfill the specifications. Traditionally, dynamic testing approaches are used for this purpose [26][14][30]. However, due to the complexity and scale of today's software systems, dynamic testing can be too time consuming and labor intensive. In particular, systems with timing constraints are difficult to test efficiently and effectively [19]. Moreover, testing software after its construction can be also too inefficient, especially if structural changes are required as a result of testing. 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 [22]. Such approaches naturally require the existence of expressive models of the software systems being considered.

Unfortunately, defining expressive models for software systems for the purpose of verification is not a trivial task [30]. First, models are generally defined by 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 very labor intensive and error-prone task. Moreover, models of the same system can vary depending on the skills of the modeler.

Second, software systems evolve continuously due to changes in requirements, bug fixes, etc. Accordingly, models must be adapted in parallel to software evolution or else they may become outdated [34][28]. Adaptation of models must be carried out in a timely manner, otherwise models can deviate from the corresponding software system too much. Keeping models consistent with software is tedious work.

There has been a considerable number of tool proposals in the past to derive models from software for various purposes. Many of these tools are used to derive models that express static properties of programs [5][17][25]. Our focus is on deriving models with timing properties which can be utilized in the verification of dynamic properties of programs using model-checking [8][31][10]. To provide sufficient detail, we define the model at the level of language instructions.

We consider three important challenges along this line. First, the model must include sufficient detail so that the verification process can be carried out effectively. Second, the tool must be able to derive models for varying size of programs in an acceptable time span. Last but not least, the tool must be extensible to cope with new demands. For example, due to size and complexity of programs, various model pruning techniques can be introduced when needed. Moreover, extensions to the program to be analyzed and/or demand for new verification techniques may require considerable extensions to the tool.

There are a few model derivation tool proposals for model-checking timing properties of software systems [11][18][12]. These are dedicated tools for the purpose of carrying out a selected set of verification techniques and as such they are not explicitly designed for coping with new demands. This paper presents a framework that derives models from Java programs in an automated way for analyzing timing properties with the following features:

– *Efficiency in model development*: The models are automatically derived from Java programs with minimal human interaction. Automation reduces errors that can be caused by manual processes. Furthermore, variations in models which are caused by subjective decisions of modelers are eliminated.
– *Consistency of models with software*: Since the model derivation process is automated, it is easy to re-create models after changes occur in the code. Thus, it is always possible to have models in hand that are consistent with the software.

- *Expressiveness of models*: From Java bytecode instructions of a program, the framework derives timed-automata models that can be used as input for the UPPAAL model checker [9]. These models contain details necessary for the timing analysis; such as loops, branching points and method calls.
- *Scalability of the model derivation process*: The framework can handle large Java programs in an acceptable time span. The model of a Java program with around 1000 classes can be derived under 85 minutes.
- *Extensibility of the framework*: The extensibility of the framework is provided by the adoption of Model-Driven Engineering (MDE) techniques [33].

This rest of the paper is structured as follows: Section 2 describes the architecture of the framework with the explanation of the design choices and a motivating example to show how the framework basically functions. Section 3 explains the actions in the framework in detail. Section 4 shows some experimental results to illustrate how the framework handles realistic-sized Java applications. Section 5 provides the related work. Section 6 concludes the paper with a summary and final remarks.

2 The Model-Derivation Framework

A timed automaton is basically a finite automaton extended with real-valued clocks. The clocks can be used in location[1] invariants and edge guards to restrict and guide the behavior of a timed automaton. The clocks can be also reset. The UPPAAL model checker allows to construct a network composed of multiple templates, which are blueprints of timed automata that need to be instantiated, through shared variables and synchronization actions. In the rest of the paper, we refer to such a network as a *timed-automata model* (of the system to be analyzed).

Timing analysis of software systems is generally applied to check if a system satisfies the timing constraints specified by the requirements. A timing constraint of a software system is generally a required maximum (or minimum) time restriction on a task duration [16]. A timing constraint can be expressed as an inverse reachability property of the timed-automata model of the software system. Such a property can be expressed formally as a query of the form “Do all the paths from $\mathtt{t_s}$ to $\mathtt{t_f}$ take less than $\mathtt{x}$ units of time?”, where $\mathtt{x}$ is the time restriction, and $\mathtt{t_s}$ and $\mathtt{t_f}$ are the starting and ending locations of the task, respectively. The model checker can either confirm that the model satisfies the property, in which case one can conclude that the system satisfies the constraint; or return a trace as a counter-example showing how the property is not satisfied, which can be back-traced to the implementation to see how the system violates the timing constraint.

[1] Following UPPAAL, we use the term *location* to refer to what is typically called a ”state” in a single finite atutomaton while we use the term *state* to refer to the global state of a timed-automata model in this paper.

In this paper, we propose a model-derivation framework for timing analysis of Java software systems. The framework derives a bytecode control-flow model from the source code and subsequently applies a group of transformations to this model to obtain a timed-automata model. Basically, a single Java class is transformed into a single UPPAAL template. Method calls are mapped to synchronization actions between these templates. Each bytecode instruction is transformed into a location in the timed-automata model. The transitions between successive bytecode instructions in the control-flow model are transformed to edges in the timed-automata model. The approximate minimum and maximum time spent to complete each instruction is added in the timed-automata model by arranging location invariants and edge guards as clock conditions. Although the bytecode control-flow model preserves the data and expressions in the source code, this data is abstracted away in these transformations.

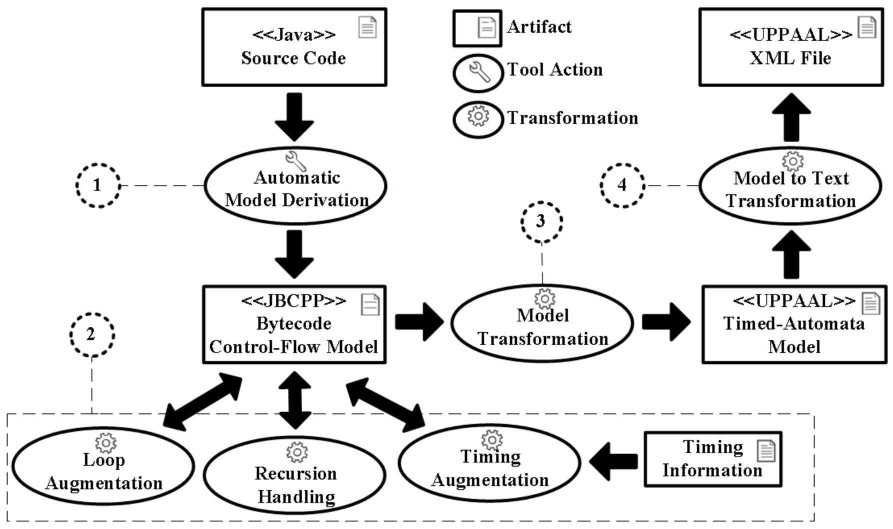


Fig. 1. Timing analysis framework

The current elements of the framework are shown in Fig. 1. We automatically derive the bytecode control-flow model of the software system from its source code in Action 1. As Action 2 (shown in a dashed rectangle), the bytecode control-flow model can be processed by various model transformations, which modify or enrich it with necessary properties aiming to set up prerequisites for the model-based timing analysis: *a*) loop augmentation aims to detect the loops in the control-flow model and annotates them with repetition limits, *b*) recursion handling modifies 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 the timed-automata model and

Action 4 transforms this timed-automata model to the format accepted by the model checker.

Due to the adoption of MDE techniques, the timing analysis framework is extensible. One can extend the framework in various ways depending on the analysis needs using the following mechanisms:

- Introducing new models with related metamodels,
- Introducing new transformations,
- Extending existing transformations or metamodels, and
- Changing the action application order accordingly.

To demonstrate the extensibility of the framework, we will describe three extensions besides the elements shown in Fig. 1: *a*) Statistical Analysis extension provides statistics such as number of various elements in the model, *b*) Unconnected Node Deletion extension removes the unreachable instructions from a control-flow model, and *c*) Node Grouping extension allows to reduce the state-space size in model checking. The details of the first two extensions can be found in Section 3.2. Node Grouping, the scalability of the framework with respect to the varying input code size and the scalability of the output models for model checking are discussed in Section 4.

2.1 Design Choices

Although any tool that supports timed-automata models could be selected, we have chosen UPPAAL since it is a mature tool and it has been proven to be useful in many case studies.

For metamodeling purposes, we have used ECore metamodels in the Eclipse Modeling Framework (EMF) [1]. The model transformations have been implemented using Epsilon Transformation Language (ETL), which is one of the domain-specific languages provided by the Epsilon framework [27]. ETL supports many-to-many model transformation using multiple input and output models conforming to different metamodels. Furthermore, ETL allows the users to inherit, import and reuse other Epsilon modules in the transformation. These properties of ETL support the requirements of reusability and extensibility of the model transformation framework. Both EMF and the Epsilon framework have large developer communities working with MDE techniques [32][13].

For the automatic extraction of a bytecode control-flow model, we have developed the Java Bytecode++ (JBCPP) plug-in for the Eclipse IDE. It comes with an ECore metamodel for Java bytecode, which we use to present the bytecode instructions and the control-flow relations between them [2]. We have chosen the compiled bytecode as a starting point for our framework. This choice offers us the following advantages:

- The bytecode of a Java system includes all the control-flow information to be used in the timing analysis. If desired, Java source code can almost completely be recovered from its bytecode.

- We increase the applicability of our framework through usage of bytecode, because we can safely assume that bytecode is always available also for the third-party components and libraries that are merely re-used but not developed in a project.
- Using bytecode frees us from tasks like resolving unqualified names or calls to overloaded methods. Also the compiler normalizes the code structure; for example it adds a default constructor which is not always written explicitly in source code.
- Although the bytecode is a low-level programming language, the required abstractions can be later introduced to remove redundant details of the models by using the extension mechanisms of our framework.

A challenge arising from the usage of bytecode is to recognize block structures such as loops in the source language. While such blocks are explicitly represented in the source code, this is no longer true in the bytecode. Nevertheless, by analyzing branching instructions and control flow, loops can still be identified as we will show in Section 3.2.

2.2 Example

Throughout this paper, we adopt an explanatory example to demonstrate how the framework works. Fig. 2 shows the source code of the example Java program. In the example, `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. This is considered as an expressive example, since it includes typical imperative object-oriented language structures such as loops, branching points and method invocations on objects that are explicitly represented in our modeling notation in UPPAAL.

Applying actions 1-4 in Fig. 1 to this program results in the timed-automata model in Fig. 3. The first template (first row) 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 this example case, we want to check if the main method finishes its execution within `x` time units. This corresponds to check whether the timed-automata model always reaches `finish` in less than `x` time units and is expressed by the following UPPAAL query: `A[] main.finish and globalClock<=x`. This query means that all paths (expressed through `A[]`) that begin in the "start" location and terminate at the `main.finish` location the condition must hold that `globalClock` is less than or equal to x when the end-state is reached (expressed through `globalClock<=x`).

When UPPAAL evaluates the query, it reports that the property represented by the query it is not satisfied. This means that there is an execution path taking more than `x` time units. In the example, the trace visiting all the locations in the timed-automata model except the locations `l_23_66` and `l_23_9` (since the system follows the flow through the branching point from the location `l_21_5`

```java
import java.util.Random;

public class Main
{
  public static void main(String[] args)
  {
    Math math=new Math();
    int rdmNmbr=new Random().nextInt();
    System.out.println(rdmNmbr+
      " is prime:"+math.isPrime(rdmNmbr));
    System.out.println(rdmNmbr+
      " is even:"+math.isEven(rdmNmbr));
  }
}
```

```java
public class Math
{
  public boolean isPrime(int nmbr)
  {
    int curDivider = 2;
    boolean isPrime = true;
    while(curDivider<nmbr)
    {
      if(nmbr%curDivider==0)
      {
        isPrime = false;
        break;
      }
      curDivider++;
    }
    return isPrime;
  }

  public boolean isEven(int nmbr)
  {
    if( nmbr % 2 == 0)
    {
      return true;
    }
    else {
      return false;
    }
  }
}
```

Fig. 2. Example Java source code

to the location `l_26_65`) is reported as a counter-example for `x`=20 and after setting the time spent at each location to 1 and the loop limit to 5.

Further details related to the construction of a timed-automata model from source code are given in Section 3.

3 The Actions of the Framework

In this section, we explain each of the actions shown in Fig. 1 in more detail. We want to remind the reader that the framework itself is extensible, the currently implemented actions may be seen as typical examples.

3.1 Action 1: From 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 a dedicated metamodel for Java bytecode. A partial view of this metamodel is given in Fig. 4. Some of the important concepts are:

- A `project` includes multiple classes, one of which is a selected main class where the execution starts in its main method.

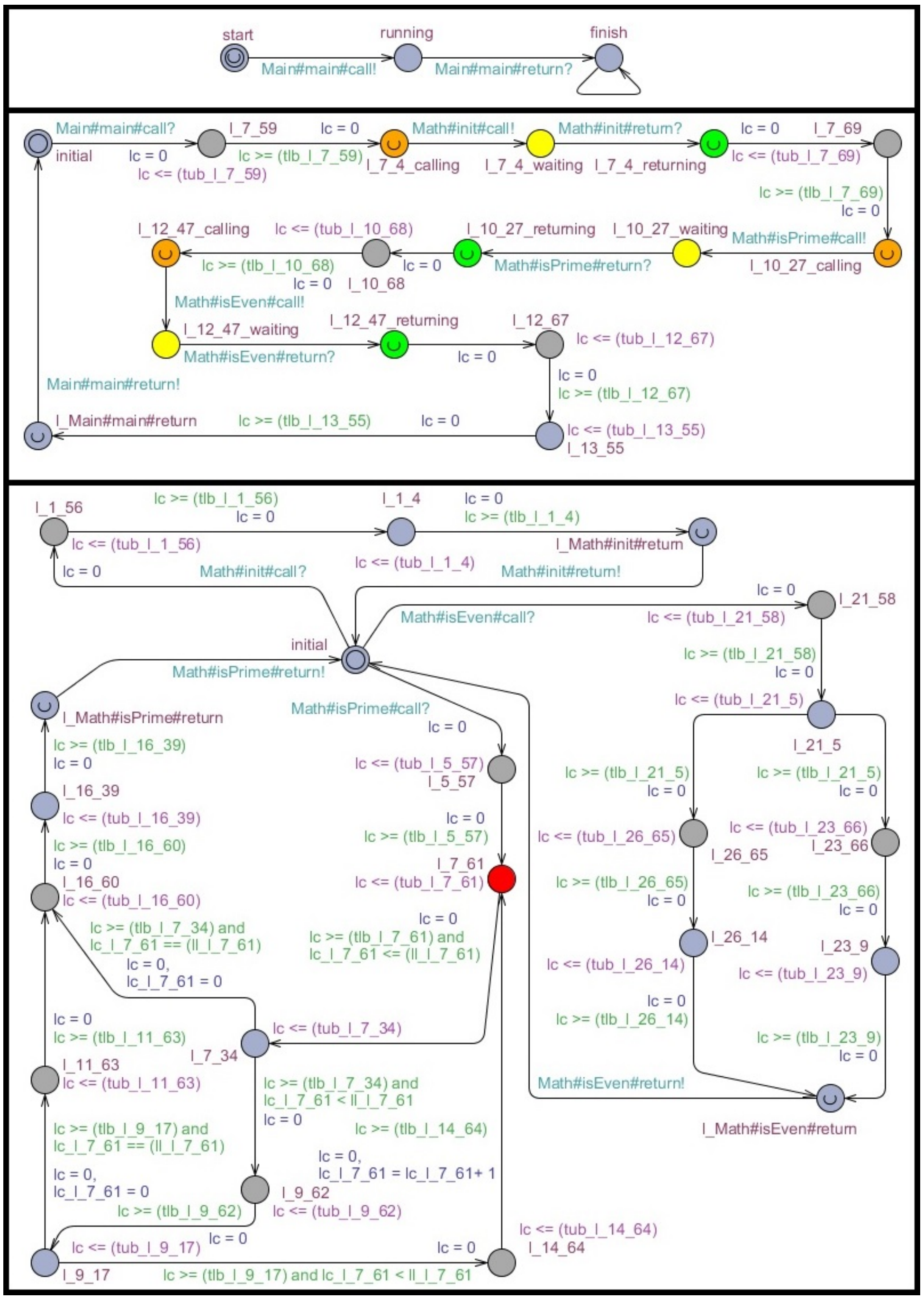
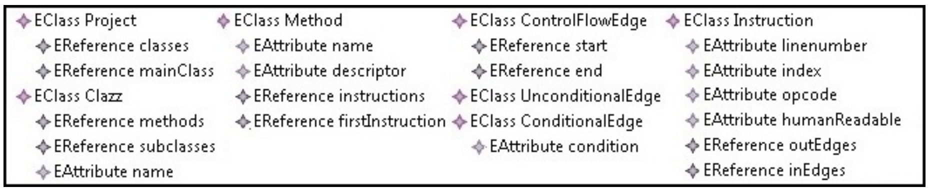


Fig. 3. The generated timed-automata model

- A `class` has multiple `methods`. Every method's signature is specified by the name and descriptor properties.
- The `instructions` are contained in methods and there is a pointer from the method to its first instruction. Each instruction has a line number showing the position in the source code and an index to identify itself uniquely inside the method.
- `Control flow edges` represent the flow of the execution of two consecutive instructions specified by the start and end pointers.



- EClass Project
  - EReference classes
  - EReference mainClass
- EClass Clazz
  - EReference methods
  - EReference subclasses
  - EAttribute name
- EClass Method
  - EAttribute name
  - EAttribute descriptor
  - EReference instructions
  - EReference firstInstruction
- EClass ControlFlowEdge
  - EReference start
  - EReference end
- EClass UnconditionalEdge
- EClass ConditionalEdge
  - EAttribute condition
- EClass Instruction
  - EAttribute linenumber
  - EAttribute index
  - EAttribute opcode
  - EAttribute humanReadable
  - EReference outEdges
  - EReference inEdges

Fig. 4. Java Bytecode Control-flow Metamodel

3.2 Action 2: Enriching the Bytecode Control-Flow Models

We have provided the following bytecode-to-bytecode model transformations:

Loop Augmentation: Loop structures frequently occur in Java source code. When starting with a piece of source code with loops, the generated timed-automata model will contain cycles. Due to the data abstraction mentioned before, there is no way to precisely predict the number of repetitions for those cycles in general. However, if we do not limit the number of repetitions in the UPPAAL model, then the model checker can create unbounded execution paths on which the global clock value increments infinitely. As a consequence, queries about worst case execution time on finitely executing software systems will not give any meaningful results. This leads to the need for *a*) detecting the loops in the JBCPP model and *b*) inserting loop-related information into the transformed timed-automata model.

Loop Detection: The loop detection takes a JBCPP model as input and generates another JBCPP model as output. The generated JBCPP model conforms to an extended version of the JBCPP metamodel to represent the loop-related information.

In our framework, we detect only so-called *natural loops*. A *natural loop* is a cycle in the control-flow graph which has a single entry point, called the loop head [6]. In other words, there is no way to reach intermediate instructions in the loop without visiting the loop head first. The bytecode generated from Java source code has only natural loops and thus in the JBCPP models there are only natural loops.

We use a dominator analysis algorithm [24] based approach for loop detection. The algorithm works basically as follows: Given a graph with a dedicated entry node (more commonly called start node), a node `d` dominates a node `n` if all paths from the entry node to `n` go through `d`. The dominator analysis algorithm derives the dominator list of all the nodes in the graph. Then, every back edge, defined as an edge from a node `s` to one of its dominators (`dh`), shows the existence of a loop whose head is the dominator. Therefore, we do back-tracing by following the edges in the reverse direction from the node `s` through `dh` to detect which nodes the loop consists of. Once the back edge, the head node and the loop nodes are detected, those branching nodes whose one edge goes out of the loop are detected. Such edges that go out of the loop are called *exiting* edges, and the edges that continue to a loop node are called *continuing* edges. The detected information is later used for insertion of the necessary information related to loops (see below).

Currently, we use a default value as the loop limit for all the loops. The framework can be extended to accept interactive user inputs or formatted data for providing the loop limits with dedicated values or to heuristically predict the loop limits.

Insertion of Loop Information: Once the loops are detected and the extended JBCPP model is generated, loop-related information is inserted to the timed-automata model during Action 3. For this purpose, we have implemented a transformation module, which is called *Loop Information Insertion.* The details related to this module are explained in Section 3.3.

Fig. 5 (The figure is excerpted from Fig. 3 and the annotations other than loop-related ones are removed.) shows how the loop-related information is inserted to the timed-automata model using annotations in UPPAAL.

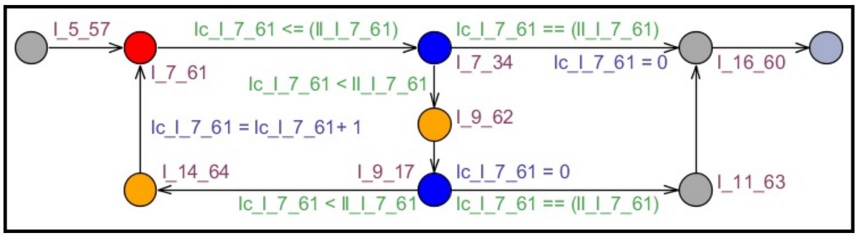


Fig. 5. Example Timed-Automata Model with Loop Annotations

In the transformation, loops are named after their head locations. In the example, the location `l_7_61`, which is shown in red color, is the head of the loop. The loop counter and the limit for loop repetition are named as `lc_l_7_61` and `ll_l_7_61`, respectively. The blue locations, `l_7_34` and `l_9_17` show the possible exiting points from the loop and orange locations, `l_9_62` and `l_14_64` show the remaining loop nodes. On the back edge, which is from the location `l_14_64` 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.

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 fixed and known before using the model checker for the timing analysis, since the model checker UPPAAL does not allow to create new instances on the fly. If we start with fewer number of instances than the possible call stack size, then we will end up in a deadlock state. Even if UPPAAL was able to create new template instances on the fly (This feature has been offered as a development snapshot in Uppaal 4.1.17), we couldn't create infinitely many new instances since after a while we will have memory problems due to state-space explosion.

Unhandled recursive methods in the transformation lead to unboundedness in the number of template instances needed, especially in the case of data abstraction. There are some alternatives about how to treat recursive method calls:

1. No treatment: If the recursive calls do not occur very frequently or their role is not very important from an analysis perspective, then we have an option of not doing specific treatment. This option still requires to inform and warn the user of the framework about the existence of any recursive calls so that s/he can decide what to do manually.
2. Removing recursion: Some calls that cause the recursion can be removed. If the timing of the recursive calls is not significant from an analysis perspective, this timing information can be approximated and the model under analysis can be modified accordingly to tolerate the information loss.
3. Transforming recursions to loops: The recursive calls can be transformed to loop structures using existing algorithms in the literature [21].
4. Estimating an approximate call depth: If we know the number of recursive calls beforehand, we can create sufficient template instances in the timed-automata model for a feasible analysis. This also requires the templates to be modified specifically not to allow more than a limited number of synchronization calls.

Currently, we have chosen the recursion removal alternative for direct recursion (which refers to the methods calling themselves) (option 2) and just reporting other forms of recursion (option 1). The other options can later be added to the framework using extension mechanisms, if desired. For removing the direct recursive calls, we have implemented a transformation called *Recursion Removal*. The *Recursion Removal* transformation takes a JBCPP model and outputs a new JBCPP model in which the direct recursive call instructions are replaced by some dummy instructions.

For reporting the other forms of recursion, we have implemented a transformation to derive the call graph of the JBCPP model. A call graph shows calling relationships between methods in the software system [7]. Nodes correspond to methods and the directed edges correspond to calling relationships. A loop in such a graph shows the existence of the recursive call. The transformation generates a call graph from a JBCPP model and reports any recursive call structures by using a depth-first search on this call graph.

Timing Augmentation: We have extended the JBCPP metamodel to include the timing information for each instruction. This extension keeps the minimum and maximum time spent for each instruction's execution.

The *Timing Augmentation* transformation takes a JBCPP model as input and enhances it with the timing information. Currently, we use a default value for each instruction type. The framework can be extended to accept interactive user input or dedicated timing data as a separate input file.

Other extensions: In the framework, we have implemented two more modules: *Statistical Analysis* and *Unconnected Node Deletion*.

Statistical Analysis gives some statistical information related to the JBCPP model such as the number of instructions, method calls, edges, methods, etc. This extension reads the input model in a read-only manner to report the desired information to the framework user. We have used this information to analyze the complexity of the models.

Unconnected Node Deletion removes normally unreachable instructions from a JBCPP model. The exception handling mechanism in Java creates an alternative group of instructions (in try-catch blocks), which are not reachable through the normal (exception-free) execution of a method. These instructions are executed only if some exception occurs and are caught. We assume that the timing analysis is carried out only on the parts of the source code that are reachable through an execution without exceptional flows, and the catching of exceptions are mostly coded for reporting-logging purposes. The transformation takes a JBCPP model and removes all the unreachable nodes and related edges from it and outputs the updated model.

3.3 Action 3: From Bytecode Control-Flow Models to UPPAAL Models

The core part of the framework is the transformation definition from JBCPP control-flow models to UPPAAL timed-automata models. This corresponds to Action 3 in Fig. 1. The UPPAAL models conform to a metamodel developed by the Software Engineering Group at the University of Paderborn [4]. 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 templates, locations, edges and clocks; it also contains syntax graphs for the C-like expressions supported by the UPPAAL model checker.

The transformation maps the entities in the JBCPP model to the timed-automata entities of the UPPAAL model. We create one UPPAAL system with global and system declaration areas for a JBCPP model. For each class in the JBCPP model, a template is created in the UPPAAL model with an initial location. Each instruction in the JBCPP model corresponds to a location in the corresponding UPPAAL template, and, the control-flow edges in the JBCPP model are mapped to edges between the corresponding locations. The locations are named in the form `l_<line_number>_<index>`.

The methods in the JBCPP model are transformed to UPPAAL in the following way: The template corresponding to the class that owns the called method receives the call via a synchronization action expressed as `<class_name># <method_signature>#call?`. This synchronization expression is on the edge from the initial location of this template to the location corresponding to the first instruction of the called method. After the instructions in the method are executed, corresponding to a path from the first instruction of the method through one of the return instructions, the template has a return edge back to the initial location with the return synchronization expression `<class_name># <method_signature>#return!`.

This mapping of methods can be explained with the following reasoning: Remember that method execution frames are realized through template instances in our approach as outlined in Section 3.2. A template instance can be only at one particular location in a certain time, which has motivated us to use multiple instances of templates to present the status of the program stack at that certain moment in time.

Due to the polymorphism, static analysis is not always able to detect which method implementation will be invoked by a method call in the bytecode. For this reason, each method call instruction is mapped to multiple pairs of call-return synchronization actions where each pair corresponds to one possible implementation for the called method. The choice of which method implementation to synchronize is decided nondeterministically such that all possibilities are tried during the computation of query results by UPPAAL.

A global clock is generated in UPPAAL to measure the time since the start of the execution. A local clock per template is used to specify the time on the locations which actually correspond to instructions. 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.

As an example transformation rule, we provide the rule in Fig. 6 for creating the UPPAAL locations from the JBCPP instructions. The `@greedy` tag forces this rule to be applied to all of the instruction subtypes. The rule starts with the rule name, `jbcpp_Instruction2uppaal_location`. The `transform` keyword, which defines from which input elements to which output elements this rule does the mapping. Following this, the rule body is defined. The `guard` keyword specifies under which condition this rule should be applied. For this case, an instruction is transformed to a location unless it is a method invocation instruction. In line 11, the created location is added to the relevant template, which is implicitly expected to be already created by another rule `jbcpp_Class2uppaal_Template`.

```
 1 @greedy
 2 rule jbcpp_Instruction2uppaal_location
 3   transform in_instruction:In!Instruction to out_location:Out!Location
 4 {
 5   guard:in_instruction.isKindOf(In!MethodInstruction)=false or in_instruction.getMethod()=
         null
 6
 7   //Setting the name of the location
 8   out_location.name="l_"+in_instruction.linenumber+"_"+in_instruction.index;
 9
10   //Adding location to the related template
11   out_location.parentTemplate=in_instruction.getOwnerClass().equivalent("
         jbcpp_Class2uppaal_Template");
12   out_location.parentTemplate.location.add(out_location);
13 }
```

Fig. 6. Example ETL Rule

Fig. 7 (excerpted from Fig. 3) shows how the UPPAAL model looks like after the transformation. This example covers a small part of the `Main` template, which corresponds to the 7th line of the `Main class` in Fig. 2. 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 as 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, expressed as `Math#init#call!` and `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. Please note that if more than one method implementation was possible for the called method due to polymorphism, then there would be more than one pair of call-return synchronization actions (with a waiting location for each possible method implementation).

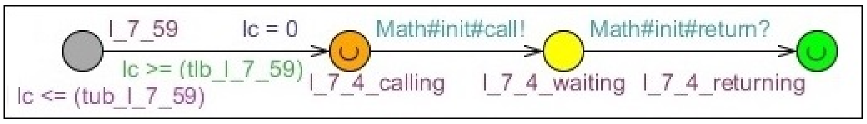


Fig. 7. Small Example: How UPPAAL Model Looks Like After JBCPP to UPPAAL Mapping

As described in Section 3.2, Action 3 also inserts the loop related information to the timed-automata model using the Loop Information Transformation module. This is achieved by extending the framework in the way as shown in Fig. 8.

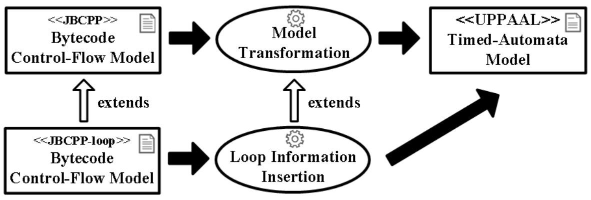


Fig. 8. Framework Extension for Loop Information Insertion

3.4 Action 4: Transformation of UPPAAL Models to Input-Compatible XML 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 Fig. 1.

4 Experiments

We have already illustrated that the framework is extensible in Section 3 with Statistical Analysis, Unconnected Node Deletion and Node Grouping modules. 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 JBCPP models after the Unconnected Node Deletion and Loop Augmentation transformations have been applied. The columns A through E show the counts of corresponding elements in the model. The column F shows the count of the method call instructions whose invokable method implementations are included in the model. The column G shows the total number of possible method implementations invocable by method calls. The column H shows the count of the return instructions.

1. *LiveGraph* is a real-time graph and chart plotter to represent large amounts of data [3]. This program includes 131 classes and 350 methods and so it can counted as a small-scale program compared to the other two programs.
2. *Groove* is a tool for modeling the design-time, compile-time and runtime structure of object-oriented systems by the use of graphs, and supports transformations of these graphs for automatic analysis [29]. We have examined

	Class (A)	**Method (B)**	**Loop (C)**	**Instruction (D)**	**Edge (E)**	**Method Call (F)**	**Method Invocation (G)**	**Return Inst (H)**	**Total (A+B+C+D+E)**
LiveGraph	131	350	33	11795	11740	665	687	440	24049
Groove Generator	930	5392	756	99738	98634	9790	12718	7114	205450
Groove Simulator	1482	9232	1454	203030	203071	20198	25272	12101	418269
Weka	1041	8322	4072	367774	374854	30124	108570	10820	756063

Table 1. Characteristics of the JBCPP Models of the Example Programs

the Simulator and Generator components of Groove. The model of the Simulator component (1482 classes and 9232 methods) has around twice the size of the model of the Generator component (930 classes and 5392 methods).

3. *Weka* offers a large collection of machine learning algorithms with preprocessing of the data and visualization of the results [20]. 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.

Section 4.1 verifies that the UPPAAL model generated by the actions of the framework consist of correct number of model elements. Section 4.2 discusses the time performance of the framework actions with respect to varying code sizes. Section 4.3 presents two scalability techniques for optimization of the model checking process of the generated UPPAAL models.

4.1 Verification of Model Element Counts of the Generated UPPAAL Models

Table 2 shows the characteristics of the generated UPPAAL models. The formulations for the expected number of model elements in the generated UPPAAL models can be described as the following (We use the notation $< column\_label >$ $_{\text{JBCPP/UPPAAL}}$ for referring to the values in the columns in Table 1 and Table 2.):

1. *Templates:* One template per class is generated. In addition to this mapping, one *extra template* is generated to keep the global clock and to call the main method of the program.

$$T_{\text{UPPAAL}} = A_{\text{JBCPP}} + 1.$$

2. *Locations:* An initial location is generated per class. A return location is generated per method. One location is generated per instruction which is not a method call. 3 additional locations are generated for the *extra template.* 2 locations (calling and returning) and one location (waiting) for each possible method implementation to be invoked are generated per method call instruction.

$$L_{\text{UPPAAL}} = A_{\text{JBCPP}} + B_{\text{JBCPP}} + D_{\text{JBCPP}} - F_{\text{JBCPP}} + 3 + 2 * F_{\text{JBCPP}} + G_{\text{JBCPP}}.$$

3. *Edges:* The edges in the JCBPP model are mapped one-to-one to the edges in the UPPAAL model. In addition the this mapping, one call and one return edge are generated per method. One edge per return instruction is generated. Two edges (one call and one return edge) are generated for per possible method invocation. Finally, 3 edges generated for the *extra template*, presenting call and return to-from the main method with an additional deadlock avoidance edge.

$$E_{\text{UPPAAL}} = E_{\text{JBCPP}} + 2 * B_{\text{JBCPP}} + H_{\text{JBCPP}} + 2 * G_{\text{JBCPP}} + 3.$$

4. *Synchronization Points:* One call and one return synchronization point are generated per method. In addition to this, to call these method implementations, two synchronization points (one call and one return point) are generated per method invocation. Also, two more synchronization points are generated for the call to and return from the main method in the *extra template*.

$$S_{\text{UPPAAL}} = 2 * B_{\text{JBCPP}} + 2 * G_{\text{JBCPP}} + 2.$$

5. *Channels:* One call and one return channel are generated per method.

$$C_{\text{UPPAAL}} = 2 * B_{\text{JBCPP}}.$$

As an example calculation, the number of locations in Weka can be calculated using the values in Table 1 on the given formula: 1041 + 8322 + 367774 - 30124 + 3 + 2* 30124 + 108570, which is equal to 515834 as given Table 2. Once the other expected results are calculated, it can observer that the other results are also consistent with the values presented in Table 2. This provides a weak validation of the correctness of the transformations themselves.

	Templates (T)	**Locations (L)**	**Edges (E)**	**Synchronization Points (S)**	**Channels (C)**
LiveGraph	132	13631	14257	2076	700
Groove Generator	931	128571	141971	36222	10784
Groove Simulator	1483	259217	284183	69010	18464
Weka	1042	515834	619461	233786	16644

Table 2. Characteristics of the UPPAAL Models of the Example Programs

4.2 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. The size of a program can be determined by the number of model elements that its JBCPP model contains, which consists of the columns A, B, C, D and E in Table 1. The sizes of the models of each program are approximately

$$|\,\mathit{Weka}|{\approx}1{,}8{\times}|\,\mathit{Groove\ Simulator}|{\approx}3{,}7{\times}|\,\mathit{Groove\ Generator}|{\approx}31{\times}|\mathit{LiveGraph}|.$$

Prediction on outcome: Although there is an expectancy to get a linear association 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 these example Java programs. For Action 2, we have used a practically fast implementation of the dominator analysis algorithm of the complexity $\mathrm{O}({D_{\mathrm{JBCPP}}}^2)$, so we expect the timing performance to have at worst a quadractic association with respect to the model size (but in practice, it can run faster). 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 3. 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)**	**JBCPP to UPPAAL Transformation (Action 3)**	**Model-to-Text Transformation (Action 4)**	**Total Time (sec)**
LiveGraph	24049	18	51	12	35	117
Groove Generator	205450	1414	86	194	364	2058
Groove Simulator	418269	1480	300	538	977	3295
Weka	756063	764	803	1069	2402	5037

Table 3. 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 have tried to find a correlation with various possible factors related to this Action, but currently we cannot say what is the determining factor. On the other hand, the timing performance is still acceptable for large projects like Weka and Groove.

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 around only 13 minutes even for the Weka program. This is due to the following fact: If the algorithm we have used is run over a connected control-flow graph, then the complexity becomes $\mathrm{O}({D_{\mathrm{JBCPP}}}^2)$. For JBCPP models, each method corresponds to a connected graph of instructions which is not connected to the other methods. Our algorithm runs the dominator analysis algorithm for each method, which results in the complexity $B_{\mathrm{JBCPP}} * (\ D_{\mathrm{JBCPP}}\ /\ B_{\mathrm{JBCPP}})^2$ that is much smaller than the complexity of the algorithm for a connected control-flow graph (The part in parentheses is the average number of instructions per method). This decreases the complexity by a factor of B_{JBCPP}. If desired, one can replace this implementation with the algorithms available in the literature with lower complexity by using complex data structures [23][15].

The last two columns of Table 3 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 with respect to the varying code sizes, which confirms our claim about the scalability of the framework. Assuming that timing analysis is not a frequently occurring task in a software development process, we can conclude that the framework is usable for programs like Weka with large code sizes.

4.3 Scalability of the Model Checking of the Generated UPPAAL Models

One of the common problems in model checking is the state-space explosion due to complexity of models in terms of the number of parallel components or the size of data domains. It is a major challenge to adjust the correct abstraction level of models to be checked. The more detailed models are, the more accurate results one can get. However, increasing the detail level of models can cause intractable state-space sizes.

UPPAAL provides some mechanisms to reduce the state-space size or to optimize the state-space generation/exploration by removing redundancy. One can extend our framework to raise the level of abstraction of the generated timed-automata models and use the optimization mechanism of UPPAAL. We have provided two such extensions in our framework:

Node Grouping: This extension allows to reduce the state-space size in model checking by decreasing the number of locations in the generated timed-automata model, by in turn decreasing the instruction count in the input JBCPP model. The *Node Grouping transformation* replaces a sequence of bytecode instructions (connected with control-flow edges) by a *group* instruction. This idea is similar to how compilers form control-flow graphs with *basic blocks*, which are sequences of instructions with a single entry point and single exit point. For our case, method calls are not grouped since they can invoke other methods with instructions, and also branches are not grouped since they affect the execution flow.

A *group* instruction's timing information is assigned a cumulative calculation of the timing characteristics of its instructions. The cumulative values of maximum and minimum time of a group instruction does not affect the result of the computation of worst and best case timing analysis.

The decrease in the count of the JBCPP model can be estimated in the following way:

1. *Simple case:* If there is no branching or method call in a method, we will replace all the consecutive instructions by one group instruction.
2. *Method call:* If there is a method call in a sequence of instructions, then this case can be replaced by two group instructions with the method call in the middle. If we generalize this case, each method instruction causes one more group instruction to be created with respect to the simple case.
3. *Branching:* Each branching instruction causes as many group instructions to be created as the number of its outgoing edges.

The ratio of the total count of method call and branching instructions over the total count of the instructions is between 0.08 and 0.14 for the example programs. This means approximately 85-90% of the instructions are other types of instructions which can be grouped and so the model size can be dramatically reduced by the Node Grouping transformation. As an example, if we apply the Node Grouping transformation to the Weka model (This transformation itself takes around 540 seconds), the number of instructions decreases from 367774 to 118139 and the number of edges decreases from 374854 to 125219, which leads to a size reduction in the model by 66%. Please note that the number of classes, methods, method calls, branch and return instructions do not change during this transformation.

Symmetry Reduction: When a template instance TA1 has to synchronize with an instance of template TB, it needs to choose with which instance it wants to synchronize. UPPAAL generates the same state-space for any choice of instances of template TB since all instances of the same template are identical. This fact helps us to use the symmetry reduction optimization in UPPAAL to reduce the state-space size by recognizing this situation and by not creating extra states for identical cases.

The default transformation definition from JBCPP models to UPPAAL models enables the symmetry reduction optimization of UPPAAL.

Experiment setup: We have used the example Java program given in Section 3 to show how Node Grouping and Symmetry Reduction help to reduce the state-space size created by UPPAAL using the generated timed-automata models by the framework.

The number of model elements of the JBCPP models with and without Node Grouping for this program is shown Table 4. As it can be seen in the figure, the grouping transformation reduced the number of instructions by 63% by replacing 14 groups of instructions with group instructions.

We have used the two JBCPP models whose details are provided in Table 4 to generate twelve UPPAAL models that cover the combination of the parameters Node Grouping (with/without), Symmetry Reduction (with/without) and the loop limit (5, 10 or 15). We have included the loop limit as a parameter to observe its impact on the state-space size.

	Without Grouping	With Grouping
Class	2	2
Method	4	4
Loop	1	1
Instructions in Loops	8	5
Instruction	67	25
Group Instruction	0	14
Edge	65	23
Method Call	3	3
Branch	3	3

Table 4. Model Element Counts of the JBCPP Models of the Example Program with and without Grouping

One way to check the maximum possible size of the state-space of an UPPAAL model is to ask the model checker to evaluate a reachability query for a state that is not reachable. To this aim, we have chosen the query that checks if the model has a deadlock. Since the generated models do not have any deadlock, we expect the model checker to create and search the whole state-space of the models.

We have used four instances (three of them are redundant) of the `Math` template to be able to observe the impact of Symmetry Reduction. The `Main` template will choose an arbitrary one of these instances for synchronization.

Experiment outcome: Table 5 shows the results of the experiment. In this table, the columns ”State-Space” show the number of states created by UPPAAL. The columns ”Improvement” show the percentage of improvement achieved by the optimization combinations compared to the state-space size created without any optimization for the same loop limit.

	Loop Limit = 5		**Loop Limit = 10**		**Loop Limit = 15**	
	State-Space	**Improvement**	**State-Space**	**Improvement**	**State-Space**	**Improvement**
Without Any Optimization	304	0%	464	0%	624	0%
Only Node Grouping	182	40%	282	39%	382	39%
Only Symmetry Reduction	109	64%	149	68%	189	70%
Node Grouping and Symmetry Reduction	56	82%	81	83%	106	83%

Table 5. State-Space Size Generated by Various Optimization Configurations

We can derive the following formula from the results and the structure of the models to calculate the total state-space, $\mathrm{SS}_{\mathrm{total}}$, for this experiment:

$$\mathrm{SS}_{\mathrm{total}} = \mathrm{SS}_{\mathrm{const}} + \mathrm{N}_{\mathrm{inst}} * (\text{Loop Limit} * \mathrm{SS}_{\mathrm{loop}} + \mathrm{SS}_{\mathrm{src}})$$

where the left-hand side of the summation, $\mathrm{SS}_{\mathrm{const}}$, is the state-space size not affected by the Symmetry Reduction optimization and the right-hand side is the affected state-space by it. N_{inst} is the number of template instances available for synchronization, SS_{loop} is the size of the state-space created for the loop itself if the loop counter is 1 (one state for each instruction in the loop), and SS_{src} is the size of the state-space created for the part of the model outside the loop.

We can make the following conclusions from this experiment:

– *Loop Limit:* Increasing the loop limit affects only the part of the model with the loop in it. The increase in the size of the state-space on this part has a constant factor. This factor is calculated by the number of instructions in the loop, the loop limit itself and the number of available template instances for synchronization (if the Symmetry Reduction optimization is disabled). As an example, the number of states without any optimization increases by 160 if the loop limit increases by 5. This increase can be calculated like the following: 8 (number of instructions in the loop) * 4 (number of available template instances for synchronization) * 5 (loop limit increase).
– *Symmetry Reduction:* Since the Symmetry Reduction optimization is based on synchronization points and available template instances, it affects some part of the model. The impact of the optimization on the state-space size has some constant factor depending on the number of instances available for synchronization. For the example, N_{inst} is 4 if this optimization is applied and is 1 if this optimization is not applied.
– *Node Grouping:* The values of SS_{const}, SS_{loop} and SS_{src} depend on if the Node Grouping optimization has been applied. For example, for the models without this optimization, SS_{const} is 44, SS_{loop} is 8 and SS_{src} is 25. When this optimization is applied, SS_{const} becomes 14, SS_{loop} becomes 5 and SS_{src} becomes 17. The size of the state-space created based on the affected parts of the model with the Node Grouping optimization is reduced with a percentage of how many instructions are grouped.

The optimizations have impact areas related to where they can be applied in the model. As a result of this fact, the reduction of the state-space size depends on the structure of the underlying model. Considering method calls and sequences of instructions are commonly found in a Java program, these optimization techniques provide significant improvements in model checking performance by reducing the state-space size. For this example case, we have achieved to get an improvement over 80% when both optimizations have been applied.

5 Related Work

Bernat et. al. have proposed a worst-case execution time (WCET) analysis scheme based on Java bytecode and a tool called Javelin to support this scheme [11]. They use annotations based on 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. Their approach does not offer any extension mechanisms. There is also no explanation on how inheritance and polymorphism are handled and no report on the scalability of their approach.

Frost et. al. have presented a tool called TetaJ for statical analysis for WCET analysis from Java bytecode [18]. They use three layer of models: the program, the virtual machine and the hardware model. These models are presented as separate templates in a timed-automata model in UPPAAL. 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 implemented optimizations coming with it to reduce the state-space size, it does not offer mechanisms for extension to help with custom optimizations. Although they report the performance results of the model checking on an example with 18 classes and 44 methods, the paper includes no information about the performance and scalability of the automatic model derivation process from the bytecode for large programs. They also do not explain how inheritance and polymorphism are handled by their tool.

Bucaioni uses Model-Driven Engineering techniques to generate a set of candidate models with timing elements from a high-level model of a vehicular embedded application in order to perform timing analysis [12]. Bucaioni provides a tool for end-to-end timing analysis specifically for EAST-ADL models. However, they do not yet offer any explicit extension mechanisms. There is also no report about the scalability of timing performance of the tool.

6 Conclusion

In this paper, we have presented a framework that derives models from Java programs in an automated way for analyzing timing properties. The framework provides the following features:

- Extensibility of the framework: Due to adoption of MDE techniques, the users can modify, adapt and extend the framework conveniently. For example, it may be necessary to optimize the models in domain-specific ways to speed-up certain analysis techniques. Along this line, we have extended the framework with Statistical Analysis, Unconnected Node Deletion and Node Grouping modules. In this paper, model-analysis techniques are considered to be out of the scope.
- Efficiency in model development: The models are automatically derived from Java programs with minimal human interaction. This has been described in 3.1.
- Consistency of models with software: Since the model derivation process consists of multiple actions that are automated. it is easy to re-create models after changes occur in the code. This ensures that it is always possible to have models in hand that are consistent with the software.
- Expressiveness of models: Typical imperative object-oriented language structures such as instructions, loops, branching points and method invocations on objects are detected in Java bytecode of a program and are represented

in the derived timed-automata models that are interoperable with the UPPAAL model checker.

– Scalability of model derivation process: The framework is able to cope with realistic software sizes in an acceptable time span. We have shown this property in Section 4 using four example real-sized software programs.

One direction for future work is to extend the framework to include the abstracted data. This can provide us a more flexible way for handling the missing information such as loop limits and timing values for the instructions. Static analysis of the source code and/or semi-automatic ways such as profiling can be used to obtain the missing information.

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

7 Acknowledgement

Thanking UPPAAL metamodel owners and also Steven te Brinke for the Node Grouping transformation algorithm.

References

1. Eclipse Modeling Framework website. `https://www.eclipse.org/modeling/emf/`, May 2015.
2. Jbcpp plug-in website. `http://sourceforge.net/projects/jbcpp/`, May 2015.
3. Livegraph website. `http://www.live-graph.org/`, May 2015.
4. Software Enginering Group website, University of Paderborn. `https://www.hni.uni-paderborn.de/en/software-engineering/`, May 2015.
5. I. Aghav, V. Tathe, A. Zajriya, and M. Emmanuel. Automated static data flow analysis. In *Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference on*, pages 1–4. IEEE, 2013.
6. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques.* Addison wesley, 1986.
7. F. E. Allen. Interprocedural data flow analysis. In *Proceedings of IFIPS Conference (Software)*, pages 398–402, 1974.
8. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
9. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, Oct. 1995.
10. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124. Springer, 2004.
11. G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using java byte code. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 81–88. IEEE, 2000.

12. A. Bucaioni12. Raising abstraction in timing analysis for vehicular embedded systems through model-driven engineering.
13. F. Budinsky. *Eclipse modeling framework: a developer's guide.* Addison-Wesley Professional, 2004.
14. I. Burnstein. *Practical software testing: a process-oriented approach.* Springer Science & Business Media, 2006.
15. K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10, 2001.
16. B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *Software Engineering, IEEE Transactions on*, (1):80–86, 1985.
17. G. Fischer, J. Lusiardi, V. Gudenberg, and J. Wolff. Abstract syntax trees-and their role in model driven software development. In *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pages 38–38. IEEE, 2007.
18. C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen. 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*, pages 30–39. ACM, 2011.
19. S. Gardiner, editor. *Testing Safety-Related Software: A Practical Handbook.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.
20. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.
21. S. Himpe, F. Catthoor, and G. Deconinck. Control flow analysis for recursion removal. In *Software and Compilers for Embedded Systems*, pages 101–116. Springer, 2003.
22. P. S. Kaliappan. Model based verification techniques. 2008.
23. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.
24. E. S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, 1969.
25. S. Nair, R. Jetley, A. Nair, and S. Hauck-Stattelmann. A static code analysis tool for control system software. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 459–463. IEEE, 2015.
26. M. A. Ould and C. Unwin. *Testing in software development.* Cambridge University Press, 1986.
27. R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 162–171, Washington, DC, USA, 2009. IEEE Computer Society.
28. J. Rech. *Emerging Technologies for the Evolution and Maintenance of Software Models.* IGI Global, 2011.
29. A. Rensink. The groove simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.
30. R. G. Sargent. Verification and validation of simulation models. In *Proceedings of the 2011 Winter Simulation Conference (WSC).*

31. J. Springintveld, F. Vaandrager, and P. R. D'Argenio. Testing timed automata. *Theoretical computer science*, 254(1):225–257, 2001.
32. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.
33. M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.
34. C. Wagner. *Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems*. Springer Science & Business Media, 2014.