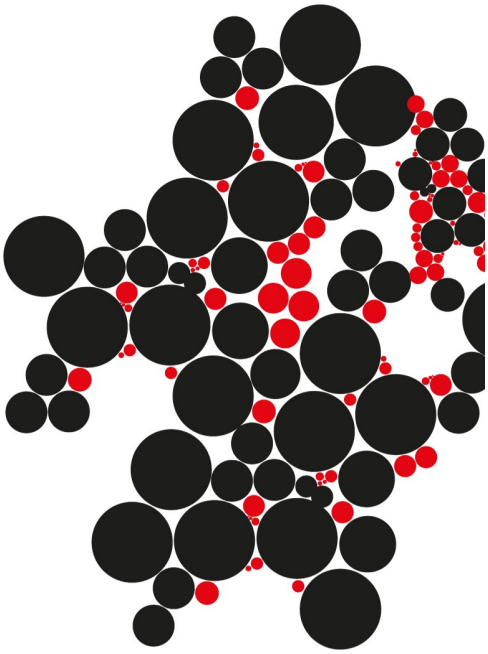


# A JAVA BRIDGE FOR LTSMIN

Ruben Oostinga

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE (EEMCS)
FORMAL METHODS AND TOOLS (FMT) RESEARCH GROUP

**EXAMINATION COMMITTEE**
prof.dr. J.C. van de Pol (1st supervisor)
prof.dr.ir. A. Rensink
dr.ing. C.M. Bockisch

UNIVERSITY OF TWENTE.

18 JULI 2012

# Contents

# 1 Introduction

## 1.1 Background

With today's advancing technology we become more and more dependent on automated systems. We even trust our lives to those systems functioning properly. Think of the fly-by-wire system of a modern airliner. The system makes sure that the instructions of the pilot are translated into the correct movement of the plane. A fly-by-wire system is a lot more complicated than a mechanical system, therefore there is more opportunity for things to go wrong. To use such complicated systems it must be certain that it will function properly or otherwise it could have fatal consequences. Other systems, although lives may not depend on them as directly, should also function properly at all times.

To ensure this the system must be tested. Systems are tested on various situations to verify they behave as expected. However, the problem with this is that of the sheer number of possible situations it is often impossible to test for every situation that could be encountered. Besides this it can also be very labour-intensive to test every situation even if tests are automated. Automated model checking attempts to solve this problem. A system will be abstracted to a model which behaves in the same way as the system. This model will be checked by looking at every situation and state the model can find itself in. This will then confirm that the system also behaves as intended. The intended behavior is specified as a certain property. There are two property types, safety properties and liveness properties. Safety properties specify that nothing "bad" will happen and liveness property specify something "good" will eventually happen.

Although model checking looks like the solution to finding errors in automated systems it also has limits. The more complicated the model, the more states it can reach. The total number of states will rise exponentially with each new state variable. For example, when a variable is added which can have 2 values, the total number of states will double. Variables that can have more values with will increase the total number of states even more. Because in order to prove that a property holds, every state has to be visited, it can take too long to visit every state. In other words exponential growth in execution time does not scale well.

Model checking is a powerful technique to verify models of, for example, integrated circuits or computer algorithms / protocols. The circuit or computer algorithm is called the system of which a model is made which is validated. A system is modelled as a state-transition graph. Nodes of the graph represent the state of the system, the edges of the graph represent transitions. One state is designated as the initial state. The collection of the states of the system is called the statespace. Some model specifications allow labels to be added to the transitions and / or to the states.

An example of a state-transition graph is shown in figure 1. It shows a simplified communication protocol. The labels on the nodes representing the states, show the state label above the state vector. The transitions also have labels describing the action that takes place. The arrow labelled *"start"* points to the initial state.

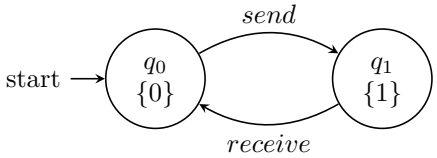



Figure 1: A model a of simplified communication protocol

Liveness and safety properties can be specified using temporal logic. Temporal logic can specify properties qualified in terms of time. It can reason about terms like next, eventually and always. A property or a specification made in temporal logic can be verified by exploring the various states of the system [2]. Verification by visiting the states one at a time is called enumerative verification.

As described previously, model checking is often limited by the exponentially growing amount of states with increasing complexity of the model. This problem is called the statespace explosion problem and techniques have been developed to alleviate this problem. One of the techniques is called symbolic model checking [12]. This makes it possible to consider multiple states and transitions at once. There is also multi-core and distributed model checking. They can be used to perform enumerative model

checking. Both attempt to use parallel processing to speed up the statespace exploration. Multi-core model checking uses parallel processing on the same machine. Distributed model checking performs processing on multiple machines. Another technique is partial order reduction and symmetry reduction that try to avoid having to visit the entire statespace. They exploit instances where when a property holds for certain states, it will also hold for other similar states.

This makes it clear that although model checking is useful there are limitations that have to be taken into account. Research in model checking has tried to resolve the limitations by inventing new techniques and optimizing existing techniques. This project is targeted towards making research in model checking tools easier and more flexible. We made a Java implementation of LTSmin which is easier to use and to extend. It allows to interface with LTSmin and make additions to LTSmin using Java while the existing features are still available.

## 1.2 LTSmin

LTSmin is a modular model checking toolset. It has modules to provide multi-core, distributed, enumerative and symbolic model checking. It can also perform partial order reduction and verify specifications in temporal logic. It is modular in the sense that it has language modules which can read varying types of models and analysis algorithms which can validate models provided by the language modules [1]. The advantage of this modular design is that new types of models and analysis algorithms can be added while not having to build an entirely new tool. This allows new techniques to be implemented and tested faster. Now we will provide a more detailed description of the various modules.

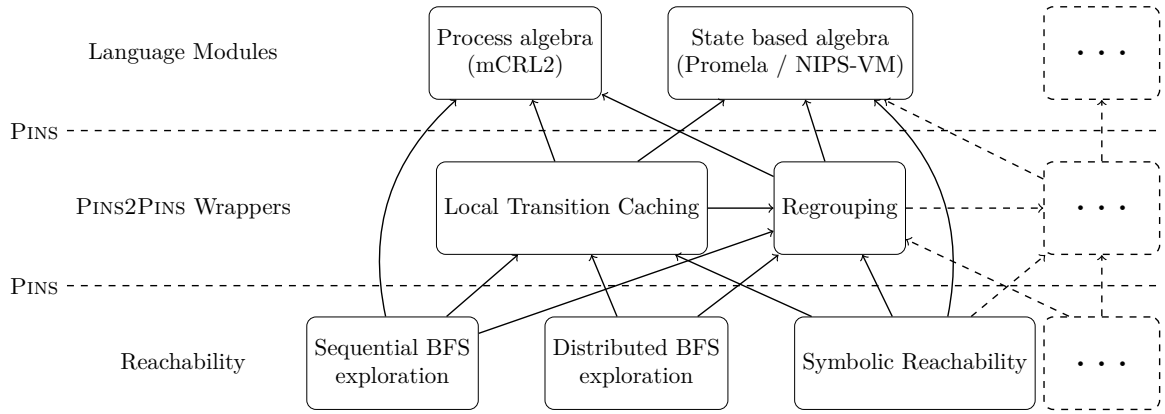



Figure 2: Architectural overview of LTSmin tools (some paths are omitted for clarity)

### 1.2.1 Language modules

Figure 2 shows the architecture of the LTSmin toolset. The top row shows the various language modules. Each module can read its own modelling language. These languages can be used to model any transitions systems like for example software, electronic circuits, puzzles and board games. LTSmin can read models which are specified for the following existing tools: muCRL [9], mcrl2 [8], DiVinE [**?**], SPIN [10], NIPS [15] and CADP [4]. Each of these modules provide the same interface which can be used by the analysis algorithms.

A language model is initialized when it is given a file containing the specification of the model. This file is then parsed and interpreted by the code of the existing tool. The language module of LTSmin will then act as a translator of the interface offered by the existing tool to the generic Pins interface each language module offers. The Pins interface is used by the various modules of LTSmin to communicate. In figure 2 this interface is represented by the dotted horizontal lines. Pins will be explained in more detail in section 1.2.3. It is also possible to implement new language modules which are only defined for LTSmin.

The language module must provide information about what the model looks like. Using this information it is possible to request states and transitions which are part of the model. When this is done

```
int x=1;
process p1 (){
  do
  ::atomic{x>0 -> x--; y++}
  ::atomic{x>0 -> x--; z++}
  od }
```

```
int y=1;
process p2 (){
  do
  ::atomic{y>0 -> y--; x++}
  ::atomic{y>0 -> y--; z++}
  od }
```

```
int z=1;
process p3 (){
  do
  ::atomic{z>0 -> z--; x++}
  ::atomic{z>0 -> z--; y++}
  od }
```

Figure 3: Promela specification of three processes

repeatedly it is possible to explore the full state space of the model.

### 1.2.2 Analysis algorithms

The bottom row of Figure 2 shows some analysis algorithms. Analysis algorithms use the Pins interface to retrieve information about the model as well as the states and transitions of the model itself. These states are then used to explore the model and to prove certain properties. Reachability tools in particular try to determine whether certain states are reachable. This proves the safety property that is being checked for. Examples are checking for deadlocks or proving properties specified in temporal logic.

The statespace can for example be explored in a breadth first search (BFS) or depth first search (DFS) order. The visited states can then be stored in an enumerative way or a symbolic way. Exploration can also take place distributed on multiple machines. There are also multi-core reachability modules to take advantage of parallel execution in multi-core systems.

### 1.2.3 Pins

Model checking uses the states of the model to verify whether a property holds for these states. A model can transition from one state to the next. In the world of enumerative model checking the *next-state function* is commonly used. It returns the successor states of a given state. In LTSmin the language module provides the intitial state. This state can be used to call the next-state function to discover new states. By requesting the next-state repeatedly using the newly discovered state it is possible to explore the full statespace. LTSmin also uses this next-state function. It does this by providing the Pins interface. Pins is an Interface based on a Partitioned Next-State function [1]. In figure 2 the Pins interface is represented by the horizontal dotted lines. The communication between the different modules takes place via Pins.

The goal of this interface is to provide access to the states and transitions of a model while exploiting what is known as *event locality*. *Event locality* refers to the fact that not the entire state is required to perform a transition. Only certain variables are accessed and modified. To make use of this, LTSmin provides a way for analysis algorithms to know which variables are read and which are written.

For example, when transitions from one state to another state are stored, it is possible to only store the values of the variables that are changed during this transition for the destination state. The values that were not stored can be taken from the preceding state which is stored fully. This is allowed because the values that were not stored did not change.

Another example is a caching layer where it is possible to only store the variables that are read to determine whether a cache hit is found. This will limit the amount of transitions that are stored in the cache. It also makes caching of transitions easier because only the variables that are read have to be checked in order to find a cache hit. Caching will be discussed in more detail in section 1.2.4.

To provide the information about which variables are used, a structure called the dependency matrix is introduced. The dependency matrix is a binary matrix in which the columns represent the variables and the row represents a group of transitions. The dependency matrix is part of the LTSmin specific information which must be provided by language modules described in section 1.2.1.

We use the example from [1]. It has three processes communicating with a shared variable. The processes are specified using the Promela modelling language [10] in Figure 3. These processes share three global variables. When we group each atomic transaction as a transaction group we get the dependency matrix in Table 1. The three variables are represented in the three columns and each process has two rows representing the two atomic transitions. As can be seen from this diagram there are only two variables used for each atomic transaction.

| | x | y | z |
|---|---|---|---|
| p1.1 | 1 | 1 | 0 |
| p1.2 | 1 | 0 | 1 |
| p2.1 | 1 | 1 | 0 |
| p2.2 | 0 | 1 | 1 |
| p3.1 | 1 | 0 | 1 |
| p3.2 | 0 | 1 | 1 |

Table 1: Dependency matrix of three processes

Because the dependency matrix is part of the Pins interface, this matrix will be available for any model. Analysis algorithms can use this information to provide an optimized statespace exploration.

### 1.2.4 Pins2Pins Wrappers

Pins2Pins wrappers are layers between the language module and the analysis algorithm. They provide the same Pins interface to the analysis algorithm as the language modules while they are a wrapper around a language module or another wrapper. The user can determine which wrappers will be used and all wrappers are optional. The purpose of wrappers is, for example, optimization or new features. Figure 2 shows two Pins2Pins wrappers which we will use as examples of the features a wrapper can offer.

- The *local transition caching* wrapper can store transitions in a data structure called a cache. When a request is made for a transition that has been requested before and is stored in its cache, the wrapper can retrieve the stored transition. The advantage of this is that the language module does not have to provide the transition. This can be faster when determining the transitions by the language module takes long enough and there are enough cache hits. However, it does not have to be faster because caching itself does take some time. It could be that the language module is fast enough or there might not be enough cache hits. We will discuss the performance of this caching algorithm in section 5.3.1.
- The *regrouping* wrapper attempts to optimize the dependency matrix. It does this for example by combining identical rows or columns and changing the order of the variables. Changing the order of the variables may greatly optimize the data structures and thereby the performance of the symbolic reachability tools. The regrouping wrapper has various strategies which can be picked by the user. When the dependency matrix is optimized, the regrouping layer will ensure the transitions that are presented to the analysis algorithm match the changed dependency matrix.

## 1.3 A Java Interface

Currently LTSmin is written in C. This is because the language modules it currently supports are also programmed in C or C++, which is easy to link to from C. However C is a low level programming language which requires memory management and does not feature object oriented programming. This makes it difficult to work with for newcomers, especially if they are used to a higher level programming language.

An object oriented programming interface in a modern programming language would make it easier to work with. Interestingly LTSmin already has data structures which looks very similar to objects as used in object oriented programming languages. LTSmin features various implementations of the same functions. This is implemented by using structures with function pointers which can point to the various implementations. This looks very similar to an object oriented approach with interfaces and implementations. Therefore an object orientated interface is a natural fit to LTSmin.

LTSmin links to existing model checking tools to make their models available via the Pins interface. It makes sense to continue this when an object oriented programming language is used. Therefore the language that is picked to provide an object oriented interface should be able to access the code of existing model checking tools. Many tools are developed in Java. The Formal Methods and Tools group at the University of Twente is working on three Java model checking tools and there also other tools

developed in Java. Therefore it makes sense to make a Java Pins interface. There are also many libraries implemented in Java which can be used.

In the preliminary research we looked at what this interface should look like and what should happen with the existing tool. We discuss this on more detail in section 1.4.1.

## 1.4 Project overview

The goal of this research project is to make it easier to add new modules by adding a new, easier to use interface. A higher level of abstraction provided by a high level programming language would allow more rapid development. This, in turn, will lower the barrier of making an implementation as well as decrease the time it takes to do this, which will speed up research in model checking. However an important condition to this is that the performance of the resulting tool is adequate, otherwise the time saved during implementation will be lost during testing and execution.

In the preliminary study we decided to make a Java bridge to LTSmin and we picked the ideal technique to make the bridge as well as a tool to help to develop it. This thesis will describe the design, implementation and evaluation of the Java interface.

Now we will discuss the research questions this project will try to answer in more detail. The goal of this project is to implement new features of an existing software project. This means the main research questions ask what the ideal way would be to implement these new features.

### 1.4.1 Preliminary research

As preparation of this final project we performed some preliminary research [13]. In this preliminary study it was decided to make a bridge from Java to the existing tool. It was not an option to only reimplement the whole tool in Java because it has many C / C++ based dependencies. These dependencies include the original interpreters for various modelling tools. Without these dependencies the tool would not be able to interpret any modelling language. Of course, interpreters could be implemented in Java as well, but reusing the implementation in the existing tool avoids a lot of extra work. Especially when the maintenance of the interpreters is also considered part of the work.

Another research question that was answered is: Which techniques should be used to implement the Java bridge? We picked JNI [11] as the bridging technique that will be used and Jace as the tool to generating the bridge [3]. We did this by defining various criteria and evaluating various techniques and tools. One of the criteria was performance and we evaluated this by running performance tests. It became clear that language bridging calls can be very expensive in terms of performance. The more language bridging calls can be avoided the better.

### 1.4.2 How to design and implement the bridging architecture?

The research for this question will focus on which features will be on the Java side and what will be on the native side. Besides this, a technical design will be made describing how the Java bridge should be implemented. The result of this research will be a class diagram and a description of the design.

This design also has criteria and requirements which it must meet. These will be taken into account during the design and implementation. Whether these criteria and requirements are met sufficiently has to be answered after the implementation is completed. The criteria are similar to the ones which the bridging technique had to meet. However, in this context they have different stakeholders. Instead of the developers that are working on the Java bridge, the stakeholders this time are the researchers which will add new modules to LTSmin using the Java interface. The criteria are the following.

1. Performance

   Because model checking is often limited by the time it takes to complete, performance is an important concern. The implementation will not increase the order of complexity of the tool because there are no fundamental changes in the way the algorithms work. Therefore only a linear increase in time of completion is acceptable. How much of a decrease in performance would be acceptable is hard to determine because it is up to the user of the tool to define which amount of time is acceptable. It should at least be possible to prototype new analysis algorithms or language modules and compare these prototypes to existing algorithms. This means the Java bridge should not

be so slow that is impossible to test new techniques properly. This can be the case when even a model with a small statespace takes days to validate. How to measure whether the performance is sufficient will be discussed in section 2.

2. Ease of use

   Ease of use is also one of the criteria of the final implementation. This refers to the ease of use for the developers using the Java interface. The native side of the new `LTSmin` tool should be invisible to the Java developer. The same should be true for C developers who should not have to deal with calls to the Java side.

3. Maintainability

   Maintainability is also a criterion shared by both the bridging technique and the final implementation of the bridge itself. Assuming changes have to be made to either the Java side or the native side, the question is: "How much work is it to integrate these changes into the other side?" Section 2 describes how to evaluate whether the criteria are met.

### 1.4.3 How to implement Local transition caching?

As explained in section 1.2.4, LTSmin has a wrapper which caches transitions. Because bridging calls are costly, it is interesting to also have such a caching wrapper in Java. When a transition is cached on the Java side, a bridging call is not needed anymore to retrieve the transition. When there are enough cache hits, the amount of bridging calls could decrease drastically.

The performance improvement of this Java caching layer will be measured. When a C analysis algorithm is using a Java language module, the existing LTSmin caching layer will also avoid making bridging calls. The performance improvement of this existing layer to language bridging runs will also be measured.

# 2 Evaluation approach

This section will describe the approach to measuring and evaluating the criteria discussed above. The measurements will also be used to perform optimizations during the implementation. The goal is to make an implementation which scores as well as possible on the proposed criteria. This will also help to gain insight of which parts of the implementation influence the criteria the most.

## 2.1 Performance

Measuring performance is only useful when there are two or more measurements which can be compared. The goal is to compare the performance of exploring state spaces of models when the Java bridge is being used to when the bridge is not used. To do this in a fair way for real world cases, we need to compare the same model using the same algorithm but in either Java or C. This will give an indication of the performance penalty of using the Java bridge.

There are various performance measurements which can be made: Startup time, time of a single transition and overall runtime.

- The interesting part of the startup time is the time it takes to parse and to interpret a model and the initialization that is required to do this. The time it takes to load the Java virtual machine or the C executable is less interesting because it is a constant time. Because there is no language module which is implemented in both Java and C, it is not possible to compare the time it takes to parse a model. Making this would require writing a language parser and interpreter which is beyond the scope of this project. Therefore measuring startup time is not very interesting. Also, the startup time is only a small portion of the overall runtime, which means it is not a limiting factor and therefore not that important because only the model has to be parsed and the interpreter has to be initialized. Generating the statespace takes many times longer. Typically the startup time will be less than 1% of the overall runtime.

- Measuring the time it takes to perform a single transition is very difficult. The Java virtual machine and the CPU perform optimizations during runtime which make it impossible to make meaningful and accurate measurements [7]. Therefore the time it takes to perform a single transition is also not what is going to be measured.

- The performance measurement that will be measured is the overall runtime of the reachability check in seconds. With a complex model which has a large statespace the time it takes to verify it is often a limiting factor. A model will be loaded and a full statespace exploration will be performed. The time it takes to do this will be measured.

- Of course the overall runtime is also affected by the non-deterministic nature of Java program execution. This is because of the same optimizations that make it impossible to make meaningful measurements of a single transition. To avoid making conclusions based on erroneous data we will apply a statistically rigorous methodology [6]. In this case it means we run each benchmark multiple times and calculate the 95% confidence interval. Conclusions will be made based on this interval instead of a single measurement.

Now it is clear what will be measured and how the results will be used, we need to determine which results can be compared fairly. The easiest two ways of execution which can be compared is a standard LTSmin run compared to a run of a Java analysis algorithm using a C language module. The analysis algorithm must be the same and the statespace should also be stored in a similar way. The difference between the two measurements will show the performance of the Java analysis algorithm as well as the overhead caused by the language bridging calls. Section 5.1 will discuss the measurement setup in more detail.

## 2.2 Ease of use

Ease of use is subjective by definition, but to give a guideline of the usability we can measure the lines of code needed to add an analysis algorithm or a language module. As stated in the introduction adding a

language module to the native LTSmin code currently requires 200-500 lines of code. This amount will be compared to the required lines of code when the Java interface is used.

Another way to measure usability is to look at the number of steps that have to be taken when a new module is added. This can be number of methods that have to be implemented or the amount of values that have to be defined. Using the number of steps and lines of code as a measurement of ease of use, we can say the Java interface is easier to use when it takes fewer steps and fewer lines of code to implement a language module.

Because BFS and DFS algorithms already have to be made in order to test the performance they can also be used to measure the lines of code and required steps. The caching layer itself will be used as an example of a language module. This is because, just like a language module, a caching layer allows the retrieval of transitions. To allow the analysis algorithms to make the same calls when the caching layer is used, it will support the same interface as regular language modules. Therefore it can be used as an example of a language module.

## 2.3 Maintainability

There are various maintenance scenarios. A likely scenario is that a change in the Pins interface is made. When this occurs it should not be required to change a lot of code to make sure the bridge still functions. Another scenario would be possible improvements to the Java bridge implementation itself. This also should be as easy to accomplish as possible.

To judge maintainability, we look at the following:

- The number of steps that are needed to make a change and the amount of work each step takes

  To measure which steps are needed to take and how much work this is we will make changes to the Pins interface and then update the Java bridge to be compatible with these changes. We will look at the following possible changes that could be made to Pins: A method is added, a parameter is added, a method is removed and a parameter is removed. After we update the Java bridge we will list what was changed and evaluate how much work it was to change it.

- Alternative implementations to evaluate whether the chosen solution is ideal.

  With alternative implementations we refer to implementations that could have been made when different design choices were made. We will look at the following alternative implementations: A reimplementation of LTSmin in Java which does not bridge to the existing tool, an implementation of the Java bridge with a different technique for bridging languages or one which bridges directly to the language module instead of the Pins interface. We will do this by comparing the steps that would need to be taken in theory to the steps that currently need to be taken. This is possible because the steps that need to be taken are the same for every implementation using the same technique.

# 3 LTSmin

As said in the introduction LTSmin provides an Interface based on a Partitioned Next-State function. Practically this means that LTSmin offers a function to retrieve transitions to following states given a current state. The next state function is partitioned because it is possible to retrieve following states by only providing a subset of the current state. These are the values that are read or written during the transition. This subset of the state is called a short state. The full state refers to the long state.

To be able to provide which variables are accessed during a transition LTSmin must also distinguish between transitions. Transitions are grouped together based on which variables of the state they influence. These groups are given an index. There are functions which produce the next state for a given group. There is also a function which simply iterates over all groups to find all transitions.

The relation between the transition group and the influenced variables is stored in a dependency matrix. It is possible to have different dependency matrices for variables that are read and ones that are written to. The language module determines whether this is the case.

## 3.1 Runtime

To give an understanding of the inner workings of LTSmin, a sequence diagram of some of the important calls is displayed in figure 4. It shows an example scenario of a reachability analysis of an mCRL2 model. In the LTSmin the Pins interface is implemented in the `greybox` module. It is called `greybox` because it provides additional information about the model, like the dependency matrix, making it more transparent than a blackbox interface.

LTSmin consists of an analysis algorithm and a language module. The analysis algorithm (Reachability in figure 4) will guide the state space exploration and the language module (`mCRL2 greybox`) will load a model and provide its states. LTSmin has a single Pins interface to the model that is called from every analysis module. It contains methods to request information about the model and methods to give the subsequent state vectors based on a given vector. A state vector represents the state of a model as an array of integers. A transition can change the values in the array to give the state vector of a subsequent state.

For each combination of an analysis algorithm and language module a different executable is created by linking the appropriate objects. The analysis algorithm contains the `main` method which is called to start the program. Now we will describe the calls in figure 4.

- `1, 1.1`: First the reachability algorithm makes the `GBloadFile` call to to the `greybox`. The file that is given is specified as a commandline parameter. The file is then passed to another function which is stored in the `greybox`. Which function this is, is determined by a compilation flag. This function causes the language module to parse and interpret the file. In this case this is the `MCRL2loadGreyboxModel` call.

- `1.1.1`: The `GBsetContext` call allows the language module to store a pointer to information it might need in the future. This can be a pointer to any structure or function. This allows the language module to have a state. The context can for example contain the interpreted model. This context is available to all the subsequent calls from the `greybox` to the language module.

- `1.1.2`: The Pins interface has a function to request subsequent states. To begin the exploration process an initial state is required. The language module stores this state with a `GBsetInitial-State` call in the `greybox`.

- `1.1.3`: The `greybox` has to know which method to call when subsequent transitions (next states) are requested. This method is stored in the `greybox` by a `GBsetNextState` call.

- `2`: The reachability tool begins exploration by requesting the initial state. This can then be used to request the subsequent states.

- `3, 3.1`: Now the reachability tool will repeatedly request successor states until no new states are found. The first time it requests states following the initial state, after this, states following newly discovered states will be requested. The `greybox` translates the generic `GBgetTransitions` call to the one that was set by call `1.1.3 GBsetNextState`. The language module will then

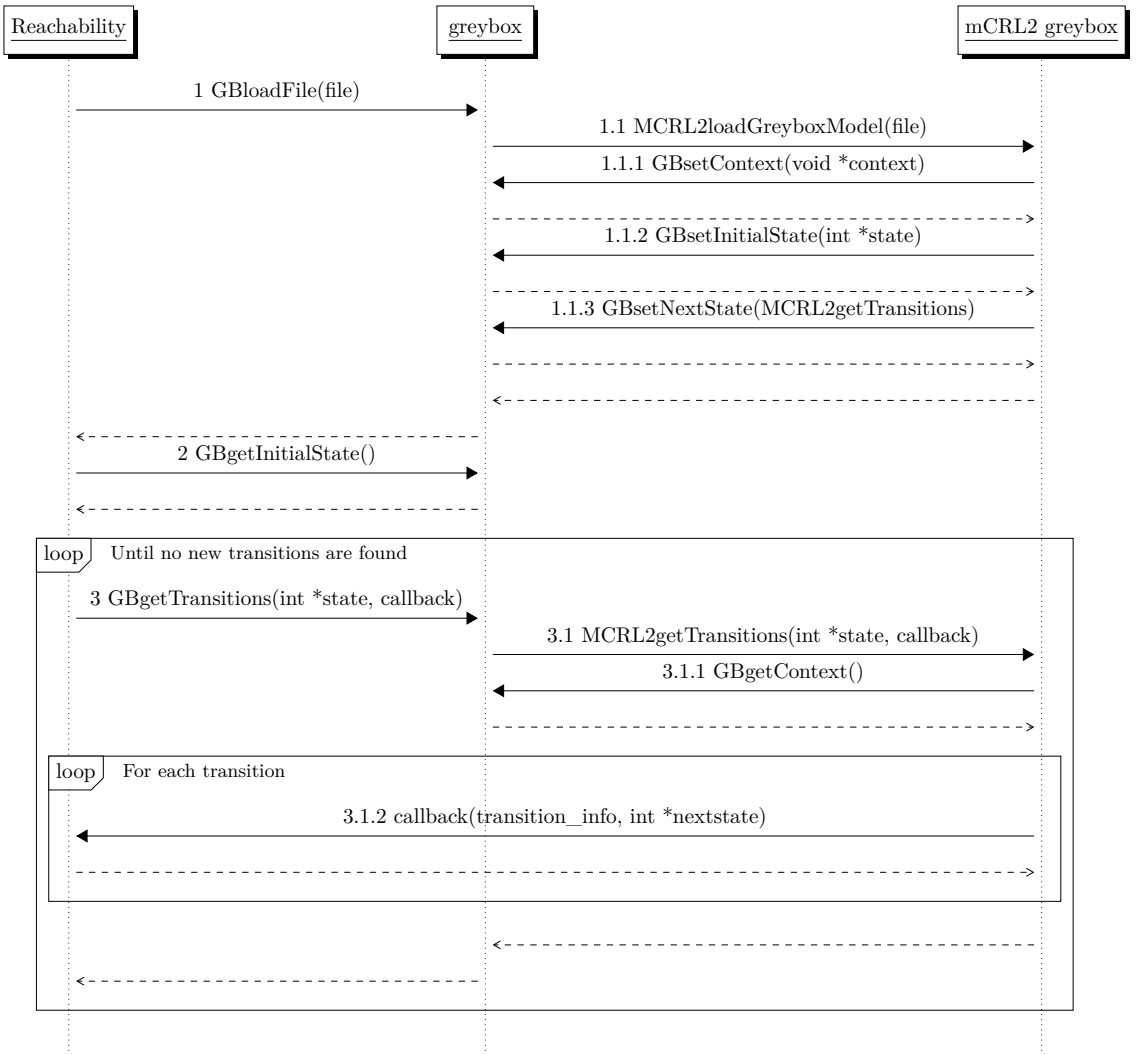



Figure 4: Sequence diagram showing LTSmin operation

determine the next states. For every state a call is made to the callback which was given along with the `getTransitions` call. This allows the states to be processed by the analysis algorithm. This callback mechanism is called internal iteration. This is because the iteration takes place in the implementation of the collection instead of in the method that initiated the iteration. Instead of returning an iterator it is required to provide a callback which will be used for iteration.

More on transitions in section 3.2.

- `3.1.1` The language module requests the context that was set by call `1.1.1 GBsetContext`. This contains the interpreted model which can be used to determine the subsequent states.
- `3.1.2` For each state that is found, the callback, which is a function pointer, will be called with a new state as a parameter. The reachability algorithm will then process the state by storing it in some data structure.

## 3.2 Transition types

As explained in section 1.2.3, it is possible to determine which variables are read or modified. States which only contain values of variables that are read or modified are called short states. Long states contain the values for all the variables. The Pins interface makes it possible to request transitions which contain either short or long states.

```
{guard done==0
 and a[25] == 2
 and a[35] == 2
 and a[42] == 2
 and a[37] == 2;

 effect done = 1; }
```

(a) A DVE transition

```
[ //array a
 1, 1, 1, 1, 1, 1, 1, 1,
 1, 0, 0, 0, 1, 1, 1, 1,
 1, 0, 1, 0, 1, 1, 1, 1,
 1, 2, 0, 0, 0, 0, 0, 1,
 1, 0, 0, 2, 0, 2, 0, 1,
 1, 0, 2, 0, 1, 0, 1, 1,
 1, 1, 1, 0, 0, 0, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1,
 //done
 0 ]
```

(b) A long sokoban state

```
[2, 2, 2, 2, 0]
```

(c) The short state for the transition in 5a

Figure 5: An example showing the different states of a model of a sokoban puzzle

We have an example in figure 5. It shows a transition and a long and short state of a simplified sokoban model. The array `a` contains a square playing field. A 0 represents a space, 1 represents a wall and 2 represents a box. The transition checks whether the value of `done` is 0 and checks whether four specific variables in the array `a` are equal to 2. If this is the case `done` is assigned 1. In sokoban terms this means that when the boxes are in the right place the puzzle is done. The short state only contains the values that are read and written (figure 5c). As can be seen the short state is a lot shorter than the long state, 65 versus 5 integers. The caching layer of LTSmin uses these short states. It will store transitions from one state to various other states. Because the short states are shorter it will use less memory. Another advantage is that variables which are not accessed are not used to find a cache hit. This means with short states there are a lot more cache hits. When a cache hit is found it is not necessary to make calls to language module. This can be a performance benefit.

The Pins interface has method calls to retrieve either short or long states. When an analysis algorithm benefits from short states it avoids having to translate a long state to a short state. Beside the fact that this makes the interface easier to use, it can also mean a performance optimization. It allows a language module to only allocate memory for the short state instead of having to request additional memory for a short state and freeing the memory for the long state.

In LTSmin the default implementation of `GBgetTransitionsShort` will call `GBgetTransitionsLong` and convert the long states to short states. The default `GBgetTransitionsLong` method will call `GBgetTransitionsShort` and expand the short states to long states. When a language module is implemented only one of the `getTransitions` methods has to be implemented. The other one will work automatically with the default implementation. Of course it is also possible and better performing to implement both `getTransitions` methods.

The third `getTransitions` method is called `getTransitionsAll`. The default implementation will call `getTransitionsLong` for every transition group. This will avoid avoid having to specify a transition group (represented by a row of the dependency matrix) for the requested transition.

## 3.3 Type system

LTSmin keeps type information about the transition system in the `lts-type` module. Each variable in the state, state label or edge label has a name and a type. Each type has a name and a value which is the actual type as used in LTSmin. This means there are variable names which are mapped to type names which are mapped to actual types.

There are four types in LTSmin: Direct, range, chunk and enum. Each of these types are converted to integers in LTSmin. This is done because of performance benefits. Integers can easily be compared

for equality and iterated over. Table 2 explains the different types. Note that the range type, like the direct type, refers to a type which can be mapped to a single integer. The difference between the range and direct type is that for the range type the maximum and minimum of the integer are known.

| Type | Description | Examples |
|---|---|---|
| Direct | Any type which can be mapped to an integer directly | 1, 2, 100, 99 |
| Range | A direct type with a known lower and upper bound | 10, 1, 100, 255 |
| Chunk | Any type which can be serialized | "label", `[0x20,0xF0,0x8A]` |
| Enum | A chunk type with a known number of different values | "label", `[0x09,0xFF,0xA0]` |

Table 2: LTSmin type descriptions

To convert the values of chunk and enum types to integers they can be stored in some sort of list. The resulting integer is the index at which the value is stored. In LTSmin these lists are called chunkmaps. The type of list that is used can be decided by the language module.

As an example we consider a model of a puzzle. The transition label has a String value for each possible move. There are four moves: "`up`", "`down`", "`left`" and "`right`". The chunkmap will map these values to integers like this: $\{up \to 0, down \to 1, left \to 2, right \to 3\}$. Now every time a transition label is encountered by the language module, it will request the integer the label is mapped to, from the chunkmap. This integer is returned as the value of the label to the analysis algorithm. The analysis algorithm could for example use the integer to see whether an identical transition has been encountered before.

The chunkmaps can also convert the integers back to the original chunks. This can be useful when the chunks are Strings because this makes it possible to print the String value of a type.

Chunkmaps are filled at runtime when chunktypes are encountered by the language module. Every time a chunk is encountered it is given to the chunkmap, which will return the integer that was assigned to it.

## 3.4 Code style

As explained in section 1.3 LTSmin is programmed in C which can be difficult for newcomers. LTSmin uses internal iteration by using callbacks. This means that collections will not provide an iterator. Instead they have a function which takes a callback function as a parameter. This function will call the callback with each element of the collection as a parameter.

We show an example in figure 6.

It shows a part of a breadth first search algorithm of LTSmin. What actually happens is that an iteration over a set is made and a method is called for each element. As can be seen the code is very difficult to read for people unfamiliar with C. We will attempt to explain the example and with this the code style of LTSmin. We assume a certain understanding of pointers and C.

A callback is a pointer to a function which is passed to another function. This function can then call the callback with certain arguments. When a callback is used to iterate over a set, these arguments include an element from the set. In the example `bfs_vset_foreach_open_enum_cb`, which is defined in line 7, is a callback that is called to iterate over a set. This is done by the `vset_enum` method which is called in line 22. It iterates over the `current_set` which is the queue of the bfs algorithm. The elements in the set are states, which are integer arrays. The implementation of `vset_enum` makes a call to the callback defined in line 7 for every element in the set. In this call the element will then be the `src` parameter of the callback. This is a pointer to an array of integers.

Often callbacks need more context than just the element from the set. In LTSmin this is solved by giving a callback a pointer to a structure containing the context the callback needs. We can see in line 7 that the `bfs_vset_foreach_open_enum_cb` method also has the parameter `args`. This parameter points to a structure that is declared in line 1. It contains another callback and a pointer to the context that callback needs. The `args` pointed is given to the `vset_enum` method in line 22. `vset_enum` passes it without modification to the callback in line 7 as the first parameter. There the `open_cb` function, from the structure that `args` points to, is called with the `src` state and the context meant for the `open_cb` function. So in summary as explained before, the example shows the code to call a function for every element in a set.

```
1  typedef struct bfs_vset_arg_store {
2      foreach_open_cb  open_cb;
3      void             *ctx;
4  } bfs_vset_arg_store_t;
5
6  static void
7  bfs_vset_foreach_open_enum_cb (bfs_vset_arg_store_t *args, int *src)
8  {
9      gsea_state_t s_open;
10     s_open.state = src;
11     args->open_cb(&s_open, args->ctx);
12 }
13
14 static void
15 bfs_vset_foreach_open(foreach_open_cb open_cb, void *arg)
16 {
17     bfs_vset_arg_store_t args = { open_cb, arg };
18     while(!vset_is_empty(gc.store.vset.next_set)) {
19         vset_copy(gc.store.vset.current_set, gc.store.vset.next_set);
20         vset_clear(gc.store.vset.next_set);
21         global.depth++;
22         vset_enum(gc.store.vset.current_set, (void(*)(void*,int*))
23                 bfs_vset_foreach_open_enum_cb, &args);
24         global.max_depth++;
25     }
26 }
```

Figure 6: LTSmin code to iterate over a set and call the open method

# 4 Java Bridge

## 4.1 Bridging technique

During the preliminary research Jace was picked as the tool to aid in bridging Java and C. The underlying technique to make this bridge is JNI. JNI allows methods of Java classes to be implemented in a compiled shared library. In Java methods can be declared with the `native` keyword. In C a method with a special name and JNI type parameters can be implemented. When this C code is compiled in a shared library it can be loaded into a running JVM. When a call is made to the method with the `native` keyword it will execute the compiled code from the shared library.

Jace helps to make such shared libraries by generated the JNI methods in C by interpreting a .class file. It will also generate a C++ class which represents the Java class. This C++ class is called a peer class. The only thing that has to be done is to provide an implementation of the methods that were declared `native` in Java.

Jace also allows calls to be made to Java objects. This is done by generating so called proxy classes. Proxy classes represent Java classes as C++ classes. The implementation of the methods however are JNI calls to a JVM which contains the represented Java objects. Primitive types are automatically converted to types which can be used by Java. Character arrays and C++ `Strings` are also automatically converted to Java `Strings`.

In figure 7 we show an example of calls from C++ to Java when Jace is used. It is part of the code that connects a C analysis algorithm to a Java language module. We can see that these calls are normal C++ calls. However the implementation of these calls will make a JNI call to a JVM. We can also see that the `const char *model_name` is converted to a Java `String` object automatically.

Jace automatically takes care of garbage collection of peer and proxy classes. Java classes which have a peer class in C++ are changed to add a methods which will ensure destruction of the objects in the shared library. This works by making a JNI call to a method in the shared library which will free the memory for that object in the shared library. For proxy classes the JVM is notified of references from the shared library which makes sure the objects are not garbage collected when there are still references to them in the shared library.

```
1  void JavaloadGreyboxModel (model_t m, const char *model_name)
2  {
3          Greybox g = GreyboxFactory::create(model_name);
4          int nvars = g.getDependencyMatrix().getNVars();
5          LTSTypeSignature sig = g.getLTSTypeSignature();
6          DependencyMatrix d = g.getDependencyMatrix ();
7          d.getRead(0,0);
8  }
```

Figure 7: C++ code that makes the connection between a C analysis algorithm and a Java language module

## 4.2 Design

This section will describe the runtime execution and design as described in the full class diagram in Appendix A. The design is of the Java implementation of the Java bridge. It will interface with the existing implementation of LTSmin.

### 4.2.1 Runtime

Here we will give an overview of the execution of the Java bridge using figure 8. This sequence diagram is similar to the one in figure 4. However some calls have been omitted to increase the readability. This diagram shows a Java analysis algorithm using a C language model. Operation of a C analysis algorithm using a Java language module look very similar. Only the names of the modules would be different.

Java has its own implementation of the Pins API which interacts with the Pins interface of LTSmin. This makes sure all the language modules are available at once. Since the interface is very similar it is easy to translate calls from one interface to another.

The Java analysis algorithm is a normal Java class. The Java `NativeGreybox` is a Jace peer class. This means that certain methods are implemented in C++. The calls to these methods are JNI calls. The C `greybox` is the same `greybox` module as in figure 4. The C language module can be any language module depending on how the shared library is linked. There is a shared library for each language module. Java will choose which one is loaded depending on the extension of the specified file. Because Jace only allows specifying one shared library per peer class, there is one `NativeGreybox` class for each LTSmin language. However, the only difference between them is their name and the shared library that is loaded.

- `1, 1.1, 1.1.1`: The Java analysis algorithm begins by parsing the commandline parameters and requesting the specified file to be loaded. This method call is passed via JNI to the `NativeGreybox` class and then to the C `greybox` and language model which loads the file.
- `2, 2.1, 2.1.1`: The `getTransitions` calls are made in the same way as the `loadFile` call. Call 2 is a JNI call, this method makes a call to the C `greybox`. The C `greybox` will make a call to a method that is specified by the language module in the `loadFile` method as explained in section 3.1.
- `2.1.1.1` Because LTSmin works with callbacks this also has to be supported in the Java bridge. In the diagram the callback is drawn from C to the Java `NativeGreybox` class. There the C transition will be converted to a Java transition. This transition will then be passed to a specified Java callback. More on how this works in section 4.2.4.

### 4.2.2 Greybox interface

As said before the `greybox` interface is the name of the Pins interface in LTSmin. In the Java bridge the same naming will be used. Therefore the Java bridge has a `Greybox` class.

Just as in LTSmin the language modules provide implementations of the `greybox`. In the Java bridge this means that language modules are subclasses of the `Greybox` class. In figure 8 we already saw the `NativeGreybox` subclass. This is the implementation of the `Greybox` class that makes JNI calls to the C implementation of the language modules. In the actual implementation there is a `NativeGreybox`

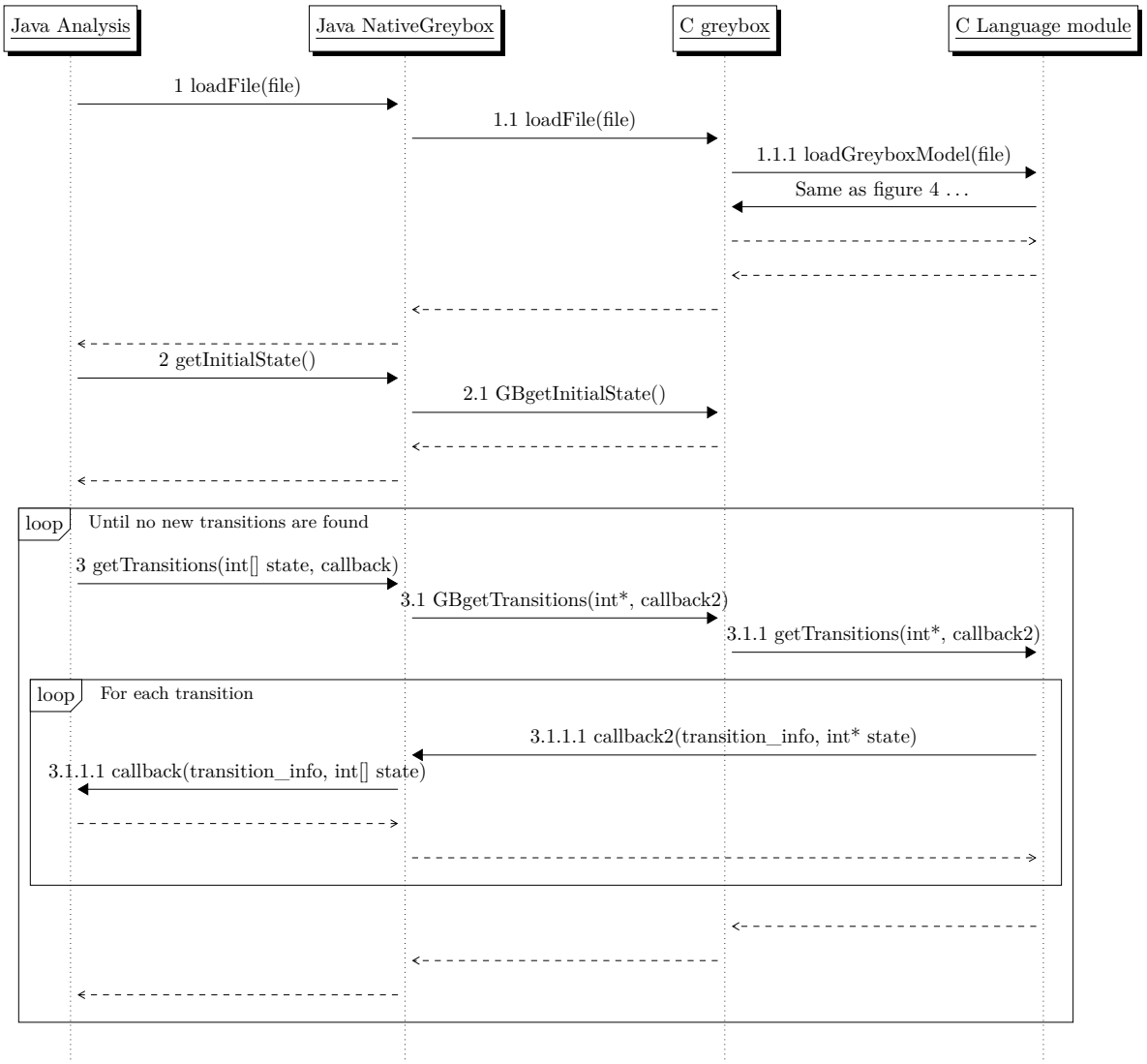



Figure 8: Sequence diagram showing bridge operation (some calls omitted for clarity)

class for every C language module. The same JNI calls are made but a different shared library is loaded to ensure a different language module is used.

Note that although `Greybox` acts as an interface to the model it is not an actual Java `interface`. This is because a Java `interface` refers to a declaration of methods that a class must implement. We will try to avoid confusion by specifically stating when we are referring to a Java `interface`.

The `Greybox` class provides access to the dependency matrix, the type information and the states of the model. This is provided via methods of the `Greybox` class. Later sections will describe the design of the dependency matrix and the type information. Now we will have a look first at the methods for retrieving the states.

`Greybox` is actually an `abstract` class. A class extending `Greybox` will provide the implementation. This can be a `greybox` wrapper or a language module. `GetInitialState` will do the same as it does in LTSmin. It provides the initial state from which subsequent states can be requested.

The Java bridge will provide the same default implementations of the `getTransitions` methods in the `Greybox` class. The language module, which is a subclass of `greybox`, will override `getTransitionsShort` or `getTransitionsLong`. The default implementation ensures the other methods work automatically.

A choice had to be made whether to copy the transitions to Java or to interface to the data in LTSmin. Because the Java analysis algorithms will always look at and often store a transition, they are

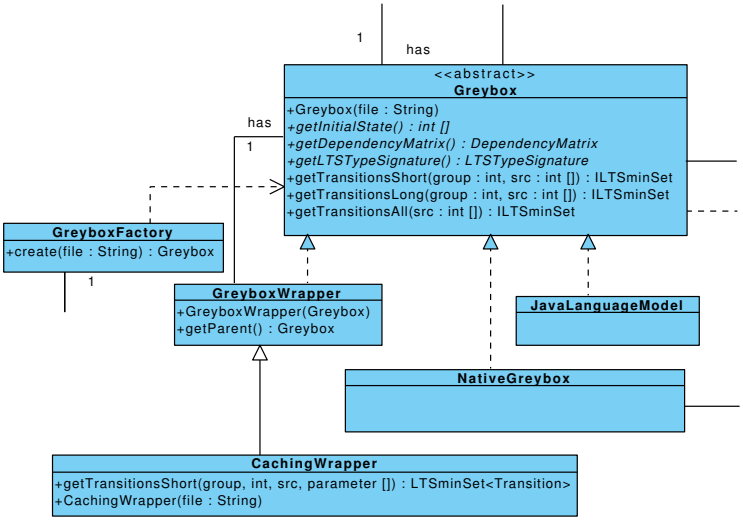



Figure 9: Class diagram of several `Greybox` classes

always copied to Java. This will avoid bridging calls every time a transition and its containing state is inspected by Java. This can occur when is transition is compared with another transition for equality. When certain storage structures are used a bridging call for every visited state will be needed to compare it to every newly discovered state. This will drastically decrease the performance. Therefore transitions are copied to Java. Note that after transitions are copied to Java they can always be removed if they are not needed. It depends on the analysis algorithm when this is the case. For BFS and DFS only the destination state of the transitions are stored and the transition label and transition group will be removed.

In LTSmin it is possible to make wrappers around the `greybox` interface. These are the Pins2Pins-Wrappers as described in section 1.2.4. The Java bridge will also allow such wrappers. An abstract class is defined called `GreyboxWrapper`. A `GreyboxWrapper` itself is a subclass of `Greybox`. It also takes a `Greybox` Object as a parameter in its constructor which is the `Greybox` that it is going to wrap. The caching layer that will be implemented as part of this project will extend the `GreyboxWrapper` class. User can can use the `GreyboxWrapper` class to make their own wrappers.

The creation of `Greybox` objects is performed by the `GreyboxFactory` class. This class creates a `Greybox` when a file with a supported extension is provided. It also takes care of possible `Greybox` wrappers. This class also registers the relation between file extensions and `Greybox` implementations. Therefore, when a new language module and thus a new `Greybox` implementation is added, its supported file extension should be added to this class.

### 4.2.3 Caching Layer

The caching layer is based on the fact that not every variable is used to determine a transition. We will demonstrate the caching layer using the example states in table 4. It shows a row in the dependency matrix for a transition group. Variable 1 and 5 are not read or written for this transition group. Variable 2 is read, variable 3 is written and variable 4 is both written and read. The caching wrapper implements the `getTransitionsShort` method. Therefore the long states are translated to short states and then given to the caching wrapper.

The example shows two different long states 1 and 2 which have the same short state. When the caching layer is asked the succeeding states for a certain short state for the first time will ask the language module. The resulting short states are stored. This will happen when the caching layer is given short state 1. The cache entry that is made in shown in the table. When the succeeding states for the same short state are requested again, the result is retrieved from cache. This occurs when short state 2 is given to the caching layer. It will find the cache entry and return the short result state. Using the original long state it is possible to convert the short result states to long states and then we have a normal transition.

This is the way in which the caching algorithm is implemented in LTSmin. This is also how one

| Description | Value |
|---|---|
| dependency matrix for transition group | `-rw+-` |
| long state 1 | `[0, 1, 0, 1, 2]` |
| long state 2 | `[255, 1, 0, 1, 0]` |
| short state 1 and 2 | `[1, 0, 1]` |
| short result state 1 and 2 | `[1, 1, 255]` |
| cache entry | `[1, 0, 1]` → `[1, 1, 255]` |
| long result state 1 | `[0, 1, 1, 255, 2]` |
| long result state 2 | `[255, 1, 1, 255, 0]` |

Table 3: Example states to demonstrate the working of the caching layer

caching algorithm in the Java bridge works. However, there is also an additional algorithm which has the possibility to have more cache hits.

| Description | Value |
|---|---|
| dependency matrix for transition group | `-rw+-` |
| long state 3 | `[2, 1, 3, 1, 255]` |
| short state 3 | `[1, 3, 1]` |
| read variables state 1, 2 and 3 | `[1, 1]` |
| written variable result state 1, 2 and 3 | `[1, 255]` |
| cache entry | `[1, 1]` → `[1, 255]` |
| short result state 1, 2 and 3 | `[1, 1, 255]` |
| long result state 1 | `[0, 1, 1, 255, 2]` |
| long result state 2 | `[255, 1, 1, 255, 0]` |
| long result state 3 | `[2, 1, 1, 255, 255]` |

Table 4: Example states to demonstrate the working of the improved Java bridge caching layer using the same transition as in the table above

This caching algorithm only stores the variables that are read to determine a cache hit. Because only the 1st and the 3rd variable of the short states are actually read, only those two variables are stored in the cache entry. Short state 3 has the same read variables as state 1 and 2, namely `[1, 1]`. This means that short state 3 which is different to short states 1 and 2 will also use the same cache entry that was stored for state 1.

The values of the cache entries in this algorithm are only the variables that are written in the transition. In the example this is the 2nd and 3rd value of the result short state from previous example. Those are the variables that are only written to. The caching layer will return the same short result state for state 3. Again using the original long states it is possible to convert this short state back to a long state.

This will result in more cache hits because variables that are not read and will be overwritten can be ignored when looking for cache hits. We have seen this for long state 3 where its 3rd variable will be overwritten without it being looked at. Therefore it can use the same cache entry as state 2 used.

### 4.2.4 LTSminSet

Because LTSmin uses internal iteration, as explained in section 3.4, it is required to provide callbacks to LTSmin to work with collections. In Java it is very uncommon to work with internal iteration. Some sort of `Collection` object is almost always used. Because one of the goals of this project is to make the interface easy to use for Java users an alternative to using the callbacks should be provided. However since callbacks do provide some performance advantages they should also be an option.

The solution to provide both the ease of use of Java Collections and the performance of callbacks is the interface `ILTSminSet<E>`. It is an extension of the Java Set interface and provides an additional method `callbackIterate` with a callback Object. A call to a `getTransitions` method of a native `greybox` will provide an `ILTSminSet<E>` Object. Only when `callbackIterate` is called, will the

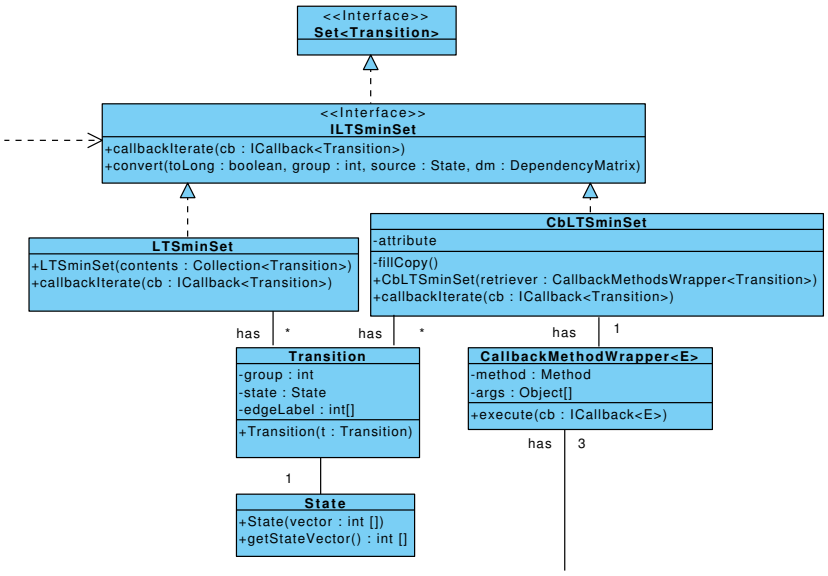



Figure 10: Class diagram of `LTSminSet` classes

actual bridging call to the native `greybox` take place. When another method of the Set interface is called, every transition will be copied to an internal `HashSet`. Then the corresponding method on this set will be called.

When a Java language module and thus a Java `greybox` is used, it might be cumbersome to implement a callback mechanism. To avoid having to do this it is also possible to convert a Java set to an `ILTSminSet<E>`. It will simply act as a wrapper around a provided set. The `callbackIterate` method is implemented by a standard iteration over the set and then performing the callback. This does not have the performance benefits of an actual callback, but it does add some additional ease of use. The `callbackIterate` method is always available and when the implementation uses a callback it has performance benefits.

### 4.2.5 LTSTypeSignature

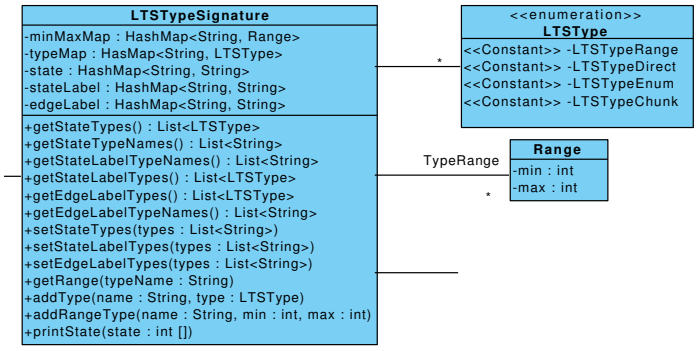



Figure 11: Class diagram of `LTSType` classes

To make the type information available in Java the class `LTSTypeSignature` is introduced. It stores the same information as the `lts-type` module does in LTSmin (see section 3.3). A language module is required to fill a `LTSTypeSignature` Object. This ensures the analysis algorithm knows how to interpret the integers from the state vector.

A choice had to be made to either make a bridging interface to the native `lts-type` module or to duplicate the type information in Java. Because most type information is initialised at once when a file

is loaded, we chose to duplicate the type information. This will avoid the need to make bridging calls when the type information is accessed either in Java or C. Admittedly this would only occur when state vectors are converted to String which typically occurs when the statespace exploration is completed. This means there will not be as much calls to the type signature as there would be when it was needed for statespace exploration. However only having one implementation of LTStypeSignature, which is the case when it is copied to Java, does simplify the design.

With the introduction of the `LTSTypeSignature` it was also possible to correct a design flaw (or missing feature) of LTSmin. In LTSmin each analysis algorithm needs to implement its own way of printing states and labels. The Java bridge will have methods available in `LTSTypeSignature` which can convert states, state labels and edge labels to Strings. This can be used to print them when necessary. More on what should be done by analysis algorithms in section 4.2.8.

### 4.2.6 Chunkmaps

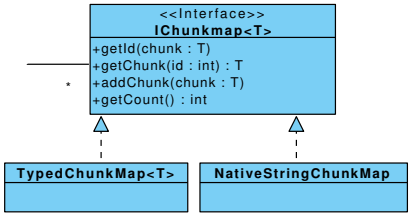



Figure 12: Class diagram of `Chunkmap` classes

As described in section 3.3 there are a lot of calls made to chunkmaps. Every time a chunk value is encountered in a state vector or transition label a call will be made to a chunkmap to convert the chunk to an integer. Therefore it would decrease performance a lot when using chunkmaps would require language bridging calls. The design takes this into account by having a Java implementation of the chunkmaps available when a Java language module is used and the native implementation for native language modules.

However when states or transitions are printed it might be necessary to convert the integers back to the chunks to make it possible to print them. As stated in the previous section it will be part of the Java code of `LTSTypeSignature` to convert LTSTypes to Strings. To allow access to native chunkmaps from Java the NativeStringChunkmap class is introduced. This will act as an interface to the native chunkmaps by wrapping the native methods. It converts chunks to Java strings using the methods provided by LTSmin. For Java language modules the class TypedChunkmap is provided. It is a generic class offering the possibility to store any Java type as chunks.

### 4.2.7 Dependency matrix

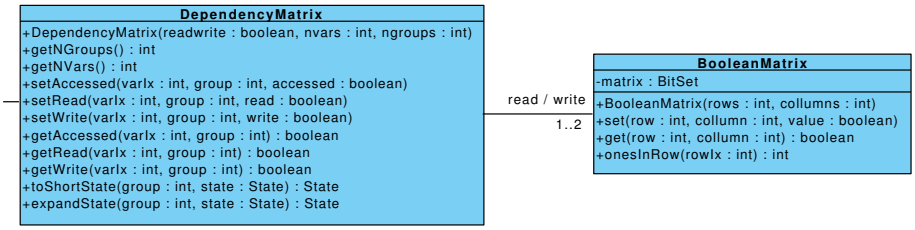



Figure 13: Class diagram of `Matrix` classes

An important part of LTSmin is the dependency matrix. It is what turns a model from a black box to a grey box. The dependency matrix does not change during runtime. As soon as the model is initialized it is available to the analysis algorithm. The analysis algorithm might use the dependency

matrix during statespace exploration which means that a lot of calls could be made to it. This mean that when it is copied to Java a lot of bridging calls can be avoided. Also because the dependency matrix does not change during runtime and the fact that it not a big data structure, it is copied to the Java Object after it is initialized in LTSmin.

The class `BooleanMatrix` provides a generic boolean matrix. The class `DependencyMatrix` has one or two `BooleanMatrices` depending on whether read and write has a separate matrix. As with LTSmin this can be determined by the language module.

### 4.2.8 Analysis algorithms

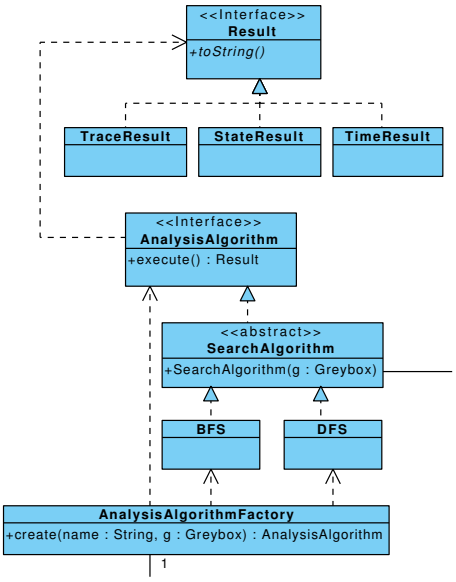



Figure 14: Class diagram of `AnalysisAlgorithm` classes

Analysis algorithms can be search algorithms like breadth first search and depth first search or any other algorithm that wants to interact with a language module. The design incorporates these algorithms as implementations of the `AnalysisAlgorithm` interface. This class requires a method `analyse` which can be called to begin execution of the algorithm. This method returns objects which are implementations of the `Result` interface. There are `Result` implementations which contain traces, states or times. Each of these objects can output a `String`. This allows results of analysis algorithms to be printed.

`AnalysisAlgorithm` objects are created by the `AnalysisAlgorithmFactory` class. Based on a name of an analysis algorithm, which is specified as a commandline parameter, and a `Greybox` it will return an `AnalysisAlgorithm` object.

### 4.2.9 Design choices summary

When a Java analysis algorithm is using a C language module there are multiple design choices that are concerned with whether certain information has to be copied to Java, whether it can be retrieved to from the existing LTSmin implementation or whether it should be moved to Java. When some information is used a lot by the Java analysis algorithm it should be copied or moved to Java. Information that is accessed a lot by the C language module should remain on the C side. Information that is not accessed often by either module could be on either side.

Table 5 shows what was chosen.

Because a full implementation has to be made in Java every part will already be available in Java. When a Java analysis algorithm is using a C language model we have to make design choices for every part of the native language module. The choice is whether to have Java use the native implementation, have Java use a copy implemented in Java, or make LTSmin use the Java implementation instead of the native one.

| | Use native implementation | Copy to java | Move to java |
|---|---|---|---|
| dependency matrix | | ✓ | |
| type signature | | ✓ | |
| chunkmap | ✓ | | |
| transitions | | ✓ | |
| statespace storage | | | ✓ |

Table 5: Show what was decided to do with various information

To have LTSmin use a Java implementation would not be too difficult to do for parts which are designed to have multiple implementations in LTSmin. Of the language module parts only the chunkmaps are designed this way. A problem with moving these to Java is that they are called by the C language module every time a chunk type is encountered. This can be multiple times for each state. Therefore having chunkmaps implemented in Java would be costly. Because the chunkmaps are not called often by the Java analysis algorithm they are not copied to Java.

The dependency matrix and the transition information are accessed often by Java. Therefore these are copied to Java to avoid language bridging calls. The type signature is not accessed as often as the dependency matrix and the transitions. Therefore the native implementation could be used. However the type information is not a big data structure and it was easier to implement to have it copied to Java. Therefore we chose to copy it to Java.

The statespace storage is exclusively used by the analysis algorithm. Therefore it is moved to Java. Requiring JNI calls to store every state would be costly. Of course when there is a highly efficient statespace storage technique implemented in C it could be considered to use this implementation. However for this project we will use a the built-in Java `HashSet` class to store states.

## 4.3 Implementation

### 4.3.1 `NativeGreybox` implementation

The `NativeGreybox` is a Jace peer class. In the Java implementation it has methods which are declared as `native`. These methods are not implemented in Java. Jace will generate a C++ peer class which declares these `methods` as well. This allows us to implement these methods in C++. Jace will ensure a call from Java to the `native` methods will execute the C++ implementation.

There are multiple methods that retrieve transitions. The C implementation works by using a callback which is called for each transition with the properties of the transition as the parameters of the callback. In LTSmin it is possible to provide context along with a callback function. A pointer to this context can be passed to a function which uses a callback which will pass this pointer unmodified to the callback function. The pointer can point to any data structure that contains data that is required by the callback function. This can, for example, be used to define to which collection the transition has to be added or even allows callbacks to call other callbacks. This makes it possible to convert the C transitions to Java objects and perform a Java callback.

Java callbacks are similar to C callbacks, but instead of a function pointer, which is a C callback, you give an object which has a method which takes the `Transition` object as an argument. In the Java bridge such an object implements the `ICallback<E>` interface where E is a `Transition`. The C++ implementation of `NativeGreybox` has a callback function which converts native transitions to Java objects. This function uses a pointer to the Java `ICallback` object in its context. When this function is given as a callback to a C function which retrieves the transitions, it will receive native transitions. Then this function will convert the native transition to Java types and perform the callback. In summary, the Java callback is wrapped by a C++ callback which performs the conversion to Java types.

### 4.3.2 LTSmin to Java

To connect LTSmin to Java language modules it is needed to connect the Pins interface of LTSmin to the Java Pins interface. Because this will operate very similar to the bridge from Java to LTSmin it is possible to use the same design choices. This means that the type information and dependency matrices and transitions are copied to LTSmin.

To retrieve transitions from Java it is needed to be able to convert a Java callback to a C callback. To do this the `NativeCallback` class is used. It implements the ICallback<Transition> interface which means that it can be used in Java as a callback to retrieve transitions. The `NativeCallback` class is a Jace peer class which means it has a method that is implemented in C++. From this C++ method the goal is to make a call to a C callback passing the transition to LTSmin. The ideal way to do this is to have the `NativeCallback` object store the C callback so it can call it from its C++ method. This means we have to store a C function pointer in a Java object. Pointers are essentially integral types. Depending on the architecture of the machine they can be 32 or 64 bits in size. This means that they can be stored as a sufficiently long integral type. Java has the `long` primitive type which is 64 bits long. We use this type to store the callback in the `NativeCallback` class. Now when a `NativeCallback` object gets a transition passed to it, it will call its C++ method providing the C callback as a `long` parameter. The C++ method converts the long back to a function pointers, converts the Java transition to a C transition and makes the C callback.

To create a Java `Greybox` a call to the `GreyboxFactory` is made using Jace. Jace will return a proxy object which can be used to make JNI calls to the Java `Greybox`. A pointer to the `Greybox` proxy is stored in LTSmin as the context of the C greybox. When a transition is requested the context is retrieved so JNI calls can be made to the Java `Greybox` object.

LTSmin requires a greybox to register the extensions it supports. These extensions are retrieved from the `GreyboxFactory` and then registered in LTSmin. This means that when a new Java language module, which supports new extensions, is added, it is only required to specify this extension in Java. It will automatically be available to LTSmin.

## 4.4 End user experience

In LTSmin the end user can choose varying configurations by executing different executables. There are two different implementation of the analysis algorithm. Both of these are linked with a specific language module. This means there are two executables per language module.

To use a Java language module a user must execute another specific executable. However, this one executable allows access to all the Java language modules. The correct one will be picked based on the extension of the model that is loaded. In theory it is even possible use a C language module while the Java bridge is used. However, this will cause an error because two language modules will be defined. One to bridge the C analysis to the Java bridge and one which loads the actual model.

To use a Java analysis algorithm the user only has to execute one command. Although Java can not provide executables, there is also only one class needed to use the Java bridge. The correct language module will be picked based on the extension of the model that is loaded. The analysis algorithm can be picked using commandline arguments.

Examples of the invocation of the Java tool and LTSmin with a Java language module are shown in appendix B.

# 5 Results

We ran performance tests on the initial implementation of the Java bridge. Section 5.1 describes how the measurements were made. Using these measurements and profiling we were able to locate parts of the implementation which could be improved. Which improvements were made are described in section 5.2. Section 5.3 describes the measurement results of the benchmarks.

## 5.1 Performance measuring setup

A problem with measuring performance when a language module implemented in Java is used is that there is no full Java language module available yet. However making a caching layer is part of this project, see section 1.4. When a Java analysis algorithm is used with a C language module transitions from the model are stored in the caching layer. This also makes it is possible to store the full statespace of the model. Using Java serialization it is possible to store this cache to a file which can be reused in performance tests as if it were a normal model which is loaded by a language module. Note that using the cache can not be compared to the original language module which filled the cache because the original model will not be interpreted, this means a lot of operations will not take place.

Figure 15 shows how to use the caching layer to provide a file with the serialized statespace. It shows two different executions. In the first run the the model is serialized by the caching layer to the model.dat file. The second run the caching layer uses this file to imitate a language module implemented in Java. The curly arrows indicate the locations where the Java bridge is used to bridge between languages.

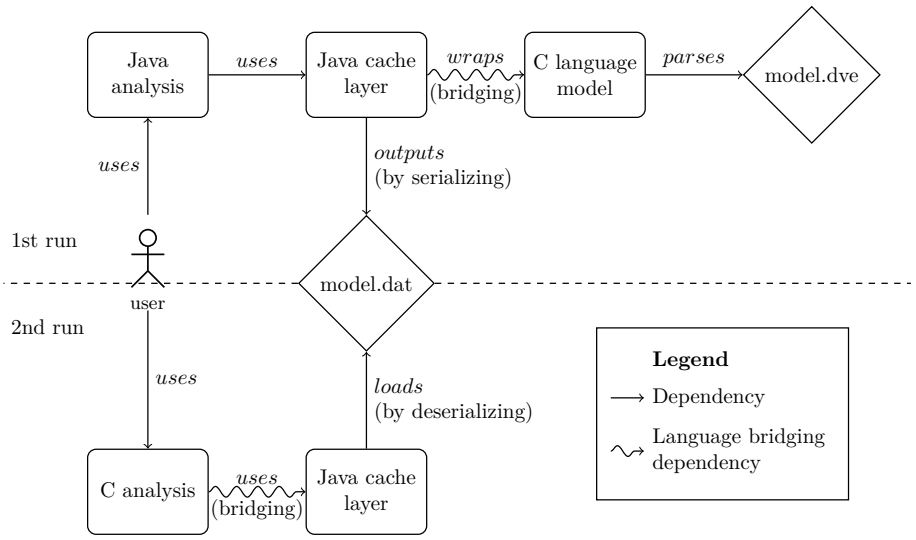



Figure 15: Diagram showing how to imitate a Java language model with 2 runs

Something else that has to be evaluated is how the Java caching layer will improve the performance. When transitions are requested which are already cached on the Java side it will avoid having to make a language bridging call. This could improve performance. To measure this a Java analysis algorithm using a C language model will be run with and without cache. The existing LTSmin implementation also has a caching layer. Its performance benefit will be evaluated when the Java cached statespace is used.

When we calculate all possible configurations the experiments can run we have analysis algorithms and language modules in 2 languages. These can have no caching, C caching or Java caching. This gives us 12 $(2 * 2 * 3)$ possible experiments. Another run has to be made to convert the model to a serialized statespace for the Java language module. The experiments are shown in table 6 and table 7.

First we will motivate why the experiments in table 7 will not be performed. Experiments 1 and 5 have caching in a different language than the analysis algorithm and the language module. These experiments would require bridging languages twice to add caching which will not be researched because the caching adds bridging calls instead of avoiding them. Experiments 2 and 3 have caching in the same language as the language module. This means that a bridging call is already made before the cache is

| # | Name | Analysis | Cache | Language module |
|---|---|---|---|---|
| 1 | LTSmin | C | | C |
| 2 | LTSmin + C Cache | C | C | C |
| 3 | C → Java | C | | Java |
| 4 | C → Java + C cache | C | C | Java |
| 5 | Java → C | Java | | C |
| 6 | Java → C + Java cache | Java | Java | C |
| 7 | Java → Java | Java | | Java |

Table 6: Table describing the experiments that will be performed

| # | Name | Analysis | Cache | Language module |
|---|---|---|---|---|
| 1 | LTSmin + Java cache | C | Java | C |
| 2 | C → Java + Java cache | C | Java | Java |
| 3 | Java → C + C cache | Java | C | C |
| 4 | Java → Java + Java cache | Java | Java | Java |
| 5 | Java → Java + C cache | Java | C | Java |

Table 7: Table describing the experiments that will NOT be performed

reached. This will not avoid bridging calls but only avoids interpretation by the language module. This will not be part of the research. Experiment 4 adds Java caching when a Java language module and analysis algorithm is used. Because the Java language module will load the entire statespace in a Java Object the caching will not add any benefit.

We also considered using different Java Virtual Machine implementations to figure out which one was the best performing. We considered JRockit and IBM J9. However because this project is more concerned with the performance of the implementation instead of the performance of different Virtual Machines, this will be outside the scope of the research.

Now we will describe the experiments from table 6 that will be performed. Before the experiments are started we will first generate the serialized statespace. This will be loaded when a Java language model is required. Experiment 1 is a normal LTSmin run. This uses C analysis and a C language module. This experiment is repeated with caching on. This will act as a baseline for the effectiveness of the caching algorithm. Then we have experiment 3 which uses C analysis and the Java language module. This will load the serialized statespace that was generated before. This experiment is repeated with C caching to see how effective this cache is at avoiding bridging calls. Experiment 5 uses a Java analysis algorithm with a C language module. This is repeated with the normal Java cache (normal meaning the cache layer that does not serialize the statespace). We also have a run using Java analysis and a Java language module. This will give an idea of the overhead of the statespace exploration.

The runtime will be measured by using the linux `time` utility. This measures the runtime of a command in seconds, with a hundreds of seconds precision. It can also give the maximum amount of memory used during the execution. More precisely it gives the resident set size in kilobytes.

Models will be taken from the BEEM database [14]. This database offers models of varying application areas with statespaces of varying size. Most of the models are well-known examples and case-studies. The database contains more than 50 models with 300 concrete instances. Every test will be performed on all of the instances of the models of the database. The accompanying website contains detailed information about the models which allows us to confirm that the statespace exploration was executed correctly.

## 5.2 Performance improvements made

Using the design as mentioned in the previous section we implemented the Java bridge. When an initial working implementation was made, we evaluated its performance. Using profiling tools we determined performance bottlenecks and eliminated these to increase the performance. We also ran preliminary benchmarks to find out the results of our improvements. This also provided some outliers which ran

slower than the rest of the performance tests. We profiled these executions and determined what was causing them to be outliers. This allowed more performance improvements. This section will describe what performance limiting factors were encountered and how these issues were resolved.

We ran the preliminary tests on around 200 of the smallest models of the BEEM database. We were testing the Java bridge by using the Java analysis algorithms with the C language module. The runtime of a full reachability check was compared to that of an LTSmin reachability check. The average times the Java bridge takes longer is shown in table 8. Note that these are preliminary results and are not as accurate as the final results which will follow.

| Implementation | Times slower than LTSmin |
|---|---|
| LinkedList for BFS queue | 284 |
| HashQueue for BFS queue | 27 |
| After JNI optimization | 22 |
| After array optimization | 13 |

Table 8: Overview of preliminary results of optimizations

### 5.2.1 Repeating the same `getTransitions` call

During preliminary performance evaluations we noticed that the breadth first search algorithm was making the same `getTransitions` calls multiple times. Because of the way breadth first search works this should not be the case. The results of every `getTransitions` call should be stored which would mean that it should not have to be made again.

It is important that `getTransitions` calls are made as few times as possible because these calls can be expensive. For example when a C language module is used a `getTransitions` call will be a language bridging call which is slower than normal calls. `GetTransitions` calls typically already have to be made for every state in a model when a full reachability check is made. When, for example, every call is made twice it is likely that the reachability check will also take twice as long to complete.

The problem turned out to be that newly discovered states were added to the queue without checking whether they already were on the queue. This meant that states were on the queue multiple times which caused the same `getTransitions` call to be made. Checking whether the queue contained the state solved the problem.

Depth first search also turned out to be making the same call multiple times. The solution to this however was somewhat different. In DFS a node will be visited multiple times because it will backtrack and look for new transition on a node it has visited before. In the depth first search implementation of LTSmin, the algorithm will only request one transition at a time. This is done by making a `getTransitions` call for every transition group until a transition is retrieved. In some models this can result in a lot of `getTransitions` calls which do not result in a transition. Because `getTransitions` calls in the Java bridge take longer to complete, it will decrease performance to make such a call for every transition group for every state. The solution we implemented is to make `getTransitionsAll` calls to retrieve all transitions at once. The resulting transitions are stored in a cache. This way, when a node is visited for a second time the new transitions can be retrieved from storage instead of from making a `getTransitions` call. This way `getTransitions` calls are not made more than required.

Note that this cache is a different cache from the caching layer. It it part of the DFS algorithm and it stores long states. When all transitions for a state are visited, they are removed from the cache. This means that this cache only contains transitions for the states on the DFS stack.

### 5.2.2 Slow performing data structures

In the previous section we had to check whether the BFS queue contains a state for every newly discovered state. This did give some performance improvement but not the improvement we expected it to give. We looked at models which took a particularly long time to complete. When profiling the Java implementation using one of these models we noticed that more than 90% of the CPU time was spent in the `equals` method of the `State` class. When we looked at the stack trace for this method we found out it was because of the `contains` method of the `LinkedList` class. We were using linked lists at

an implementation of the `Queue` interface for the BFS queue. For the `contains` call every element of the `LinkedList` has to be compared to the parameter of the `contains` call. Because there are many states that are discovered and many states on the queue this was taking a long time.

The solution to this problem was to use a `HashSet` as the BFS queue. When the `contains` call is made now, a hash of the state is calculated and it is compared to the ones in the set. This is so much faster that the 90% CPU time it took before is reduced to less than 1%. See table 9 for the methods which were consuming the most CPU time before and after the solution. The method which does the actual retrieving of the transitions is the `CallbackMethodWrapper.execute` method. Therefore this is the method which should be consuming the most CPU time.

| LinkedList for Queue | |
|---|---|
| Method | Time spent |
| State.equals(Object) | 92.7% |
| CallbackMethodWrapper.execute(callback.ICallback) | 6.2% |
| DVE2Greybox.getTransitionsAll(State) | 0.6% |
| Transition.<init>(int, int[], int[]) | 0.2% |

| HashSet for Queue | |
|---|---|
| Method | Time spent |
| CallbackMethodWrapper.execute(callback.ICallback) | 95.4% |
| BreadthFirstSearch.search() | 2.4% |
| Transition.<init>(int, int[], int[]) | 0.7% |
| State.equals(Object) | 0.5% |

Table 9: Profiling results with different data structures for the BFS queue

We used the schedule_world.2 model to produce these tables. This model was taking 13.5 hours to complete before. After the optimization it was only taking 9 minutes. This model was taking a particular long time to complete because it has 1.6 million states while it only has 17 BFS levels. This means that the BFS queue was getting very long. On some levels it was longer than 250 thousand states. This was causing the many state comparisons.

### 5.2.3 Limiting JNI calls

Because language bridging JNI calls have some overhead which causes them to be slower than normal calls they should be limited as much as possible. Especially in the `getTransitions` methods, which are often called for every state in a model, it is important to have as few JNI calls as possible. The initialization of the type information and dependency matrix also requires JNI calls. But because this stage only occurs once and is typically less then a few percent of the total runtime, limiting JNI calls here will not give big performance enhancements.

We looked where JNI calls were made and where they can be avoided. In the initial implementation the `getTransitions` methods of the C++ peer `Greybox` classes were receiving `State` objects. These classes need to convert the `State` objects to integer arrays that are used by LTSmin. However in order to retrieve the Java integer array from the `State` object a JNI call had to be made. This can be avoided by giving the java integer arrays directly to the peer class. This moves the call to extract the integer array from the `State` object to the Java side where it is a normal Java call.

When the transitions are calculated they initially were converted to `Transition` objects and then given to Java. This required additional JNI calls to make the `Transition` object. Fortunately this could be avoided by giving the `NativeGreybox` class a Java callback which takes Java primitives and will make the `Transition` object itself. This means only one JNI call has to be made to hand a transition over to the Java side.

When looking for JNI calls to limit we discovered an incomplete implementation in Jace. Jace uses the JArray class as a proxy to Java arrays. In the documentation of Jace we were recommended to use the iterator of the JArray class to copy arrays from and to Java. This was because it supposedly used a buffer to store changes before they were sent to Java with a JNI call. This is faster than making a JNI

call whenever any element of the array is changed. However, upon inspecting the code we discovered this buffer was not yet implemented. This meant using the iterator was also making a JNI call for every element that was copied. This is really inefficient because it means making a JNI call for every element in every state, every time this state crosses languages.

Fortunately the solution was easy. Jace allows to retrieve the JNI objects from proxies. Using this it is not difficult to make a JNI call manually. This way it is possible to copy an array from and to Java using a single JNI call. The result of doing this is that the implementation's average runtime was halved.

After making these improvements we can make the following summary: When a Java analysis algorithm is using a C language module one JNI call will be made to retrieve transitions. The Java state vector has to be converted to a C integer array with an additional JNI call. For every transition that is found another JNI call will be made to hand that transition over to Java. To convert the C integer array from this transition to a Java array requires another JNI call. This means 2 JNI calls to request the following transitions and 2 JNI calls to give a C transition to Java. The first JNI call is one from Java to C and the other ones are from C to Java.

For the execution from C to Java there were also opportunities to limit JNI calls. In the initial implementation a `NativeCallback` object was constructed for every `getTransitions` call. This object converts a Java callback to a C callback. Even worse another JNI call was used to set the variables of the `NativeCallback`. During profiling it became clear a lot of time was spent constructing the `NativeCallback` objects. The solution was to use a single `NativeCallback` object for all the `getTransitions` call. A set method was used to set the variables this class needs. This way a lot of object constructions were avoided along with a JNI call for every `getTansitions` call.

`GetTransitions` calls on the C side return the number of transitions that were returned. At first the `NativeCallback` object was counting the amount of transitions that passed through it. To retrieve the count from the object a JNI call was made. The implementation could be changed to avoid needing a JNI call to retrieve the count. The solution was to make a callback on the C side which counted the transitions there. This callback increased the count for every transition that passed through it and then handed the transition over to the original C callback. It acted like a callback wrapper. This way the counting takes place on the C side which means no JNI calls are necessary.

### 5.2.4 Caching layer

When we first looked at the results of performance tests of a Java to C run with caching we noticed that it was not very fast. Even when a lot of JNI calls were avoided it was running slower than a run without caching. Using a model where the caching layer avoided a lot of JNI calls we determined what was limiting the performance. When looking at the profile results as shown in the first table of table 10, we notice that a lot of time is spent in the `inState` method of the `DependencyMatrix`. This method is used to determine whether a variable should be in a long or a short state. This is needed to convert between long and short states. It makes sense that it is used a lot when a model that has a lot of cache hits is used. This is because there are not many JNI calls and most of the time will be spent converting states to store them in cache or retrieve them from the cache.

However, the `stateSize` method is also using 7% of the CPU time. This method is used to determine how many variables short or long states have and it also calls the `inState` method. The `stateSize` method could be a lot faster when its result would be stored instead of calculated using the dependency matrix. When a result is retrieved from storage no `inState` calls have to be made. When we did this we got the profiling results as shown in the second table in table 10. A lot less time was spent in the `stateSize` and `inState` methods. The `stateSize` method was only using 0.6% of the CPU time. The `inState` method was reduced to using 28% of the CPU time. The reduction of time spent in these methods meant that relatively more time went to the `convertState` method. This method also calls the `inState` method a lot, which explains why the `inState` method is still the method where the most CPU time is spent. The result of storing the results of the `stateSize` method instead of calculating it is that the overall runtime decreased on average by 39% for Java $\rightarrow$ C runs with caching turned on.

Upon examining the number of transitions for every test we discovered that some tests that were using the cache produced a different number of states and transitions to the other tests. This meant that the statespace exploration was not performed correctly. Upon examining the caching algorithm in great detail we found nothing wrong. However we did discover what was causing the problem.

We were using the improved caching layer as explained in section 5.2.4. It was not functioning as

| Without storing state size | |
|---|---|
| Method | Time spent |
| DependencyMatrix.inState(StateType, int, int) | 35% |
| BooleanMatrix.get(int, int) | 19% |
| DependencyMatrix.getAccessed(int, int) | 17% |
| DependencyMatrix.convertState(State) | 12% |
| DependencyMatrix.stateSize(StateType, int) | 7% |
| DependencyMatrix.getRead(int, int) | 3% |

| With storing state size | |
|---|---|
| Method | Time spent |
| DependencyMatrix.inState(StateType, int, int) | 28% |
| DependencyMatrix.convertState(State) | 21% |
| BooleanMatrix.get(int, int) | 18% |
| DependencyMatrix.getAccessed(int, int) | 16% |
| DependencyMatrix.getRead(int, int) | 2% |
| CachingWrapper.getTransitionsShort(int, State) | 2% |
| ... | |
| DependencyMatrix.stateSize(StateType, int) | 0.6% |

Table 10: Profile results before and after storing state size

expected. We will demonstrate this by an example. Suppose the language module was given the short states `[0,0]` and `[0,1]`. The dependency matrix for a transition says the first variable is read and the second variable is written to. Because they both have the same first variable `0` the language module should not see a difference between the two states. Therefore the second variable should be overwritten with the same variable. The result state could for example be `[0,255]`. However the language module was giving two different result states for the same transition. This should not be possible because if the language module was looking only at the variables that it says it is looking at, it would see no difference between the two states. The only way to see the difference is to look at the seconds variable also. This would mean the dependency matrix was wrong. Therefore we concluded that there is a bug in the DVE language module of LTSmin which cause the dependency matrix to be wrong. The language module is looking at more states than it says it does.

A solution to this problem would be to find the bug in the language module or to use a caching algorithm which works the same as the one in LTSmin. This is possible because LTSmin was not having problems when its caching algorithm was used. Because we have not looked at the implementation of the language module yet and finding the bug might be a time consuming activity we chose to use a caching algorithm which stores short states. We reported the bug and its solution shall remain out of the scope of this project. Running the tests again indicated that the problem was resolved. Unfortunately we were having fewer cache hits, like we expected. The short storing caching layer was making on average 3,6 times more `getTransitions` calls to the language module. However this is somewhat unreliable because the statespace exploration did not execute correctly for some models.

### 5.2.5 Other bugs

When using the caching layer a lot of CPU time was spent in creating a `Vector`. This `Vector` was used to store the cache hits in. The bug was that the `Vector` that was created was not as big as the transition that it was supposed to hold but as big as the entire cache. This meant that really big amounts of memory were requested unnecessarily. The problem was easily resolved by requesting the right `Vector` size.

When running performance tests using a C analysis algorithm and C caching with a Java language module we received an error message. It indicated that we were using a short state which had an invalid size. We discovered that the C implementation of the `getTransitionsShort` method was copying too much integers from the C state to Java. It was using the length of long states instead of that of short

states. This meant that the short states were the length of long states and that some random integers were copied to Java. This was resolved by using the correct state size.

## 5.3 Benchmarks

Because the performance tests require a lot of executions we ran the tests on the university's cluster. This allowed parallel execution of the tests on multiple machines. During the preliminary performance tests we ran multiple tests on a single machine. For the final results we do not want the tests to influence each other and therefore we used a single machine for every test. The machines are equipped with 2 Intel Xeon E5520 CPU's and 24GB of ram. Both CPU's have 4 cores and can execute 8 threads because of Hyper-Threading, this means a total of 16 threads can be executed in parallel. However this is of limited use because the Java bridge only supports single threaded statespace analysis. The JVM will use additional threads to perform tasks such as garbage collection.

We ran the tests on the latest JVM that was available on the cluster. This was the Java HotSpot(TM) 64-Bit Server VM build 21.0-b17 of JDK build 1.7.0-b147. We want to avoid garbage collection as much as possible to ensure the Java bridge performs at its best. To do this, the JVM was given as much heap space as possible. Using trial and error we determined we could give at most 20GB of heap space. Giving a 1GB more resulted in an error. 20GB is both the maximum and the initial amount of heap space given. This is because just setting the maximum would not result in less garbage collection.

The BEEM database has 300 model instances. Unfortunately the models `plc` and `train-gate` with their 11 instances would not compile for use with LTSmin. This is probably due to a bug in the translator of divine models for LTSmin. We could not solve the bug, but we did report it. This leaves 289 instances to be tested, which is sufficient.

To ensure we get reliable results we ran each test 3 times. This allows detection of possible outliers. We also set a timeout to ensure the benchmarks would complete in a reasonable amount of time. We set the timeout for 1 hour. If this would not be enough because too many tests would timeout we would rerun the timed out tests with a longer timeout. But after the measurements completed we saw that LTSmin would not time out at all because the execution would run out of memory before it ran out of time. The execution from Java to C ran out of time for only 9% of the tests. Therefore we decided to keep the time out at 1 hour.

For each test there are multiple possible outcomes. The normal outcomes are: A runtime measurement, not enough time or not enough memory. Another possible outcome is that the test could not run because the statespace was not serialized. This can occur for tests which make use of the Java language module and a Java $\rightarrow$ C run with caching could not complete. This means that the statespace can not be serialized and not loaded into the Java language module.

Unfortunately not every tests gave the same type of result 3 times. When this occurs we used the result which was found 2 out of the 3 runs. This means that when a result does not have enough memory for 2 runs and not enough time for another run the final result would be not enough memory. Fortunately 3 different types of outcomes for one model did not occur.

Because we run each test 3 times we can detect measurement errors. We define a measurement error as a results which is more than 10% higher or lower than the average and this difference is more than 1 second. When this occurs the outlier will be ignored and the average of the 2 remaining measurements will be the final measurement. We found that this detected actual measurement errors and not normal variance because the 2 remaining measurements were close to each other while the outlier was not. The outliers are probably caused by some rare circumstances like running system processes that did not occur with the other 2 measurements.

As explained in section 2.1 it is required to use the same analysis algorithm and statespace storage in both Java and LTSmin. Both the Java bridge and LTSmin have breadth first search and depth first search implemented. Therefore either of these can be used. Initially we preferred breadth first search because it gives more useful output for every depth level that is visited. However for the statespace storage the Java implementation is using hash tables. LTSmin also has this storage technique, but we discovered that only depth first search in combination with hash tables is implemented in LTSmin. An option would be to use another storage technique, but LTSmin only has storage techniques which are not in Java's default API. Since implementing other storage techniques is not part of this project we chose to use depth first search in combination with hash tables.

The results of the benchmarks are shown in appendix C. We will evaluate the results of the various benchmark by comparing them to a normal LTSmin run. We do this by dividing the runtime of a benchmark run by the runtime of an LTSmin run. We will call this the slowdown of the run. The average slowdown and the confidence interval of the average slowdown are shown in table 11.

| Benchmark | Average slowdown | 95% confidence of slowdown |
|---|---|---|
| LTSmin with caching | 15,4 | 12,3 – 18,5 |
| Java → C | 11,3 | 10,7 – 11,9 |
| Java → C with caching | 38,6 | 31,8 – 45,5 |
| C → Java | 45,9 | 38,6 – 53,2 |
| C → Java with caching | 26,4 | 20,9 – 31,9 |
| Java → Java | 39,0 | 32,2 – 45,8 |

Table 11: Table showing the 95% confidence interval of the mean of the times slower than LTSmin of various benchmarks

.

### 5.3.1 LTSmin caching

The goal of the LTSmin cache is to limit calls to the language module. When a slow performing language module is used this will cause a performance improvement. With an average slowdown of 15,4 we determined that the caching layer will most likely make the execution slower. This means that the divine language module is not slow performing. In that case the caching layer will add additional overhead which causes the slowdown. The caching layer works by using short states which have to be converted to long states. There is also additional overhead in storing and retrieving states from the cache.

### 5.3.2 Comparing Java → C to LTSmin

A Java → C run refers to a run using a Java analysis algorithm using a C language module without caching. We show the slowdown of a Java → C run plotted against the number of transitions of a model in figure 16.

We notice that an interesting pattern is shown. Up to 100 thousand transition there is a trend of an increased slowdown. From 100 thousand until a million transitions the slowdown decreases. From there on there is an increased slowdown. Note that the plot is made on a logarithmic scale which causes the increase to look more dramatic than it is. We applied a linear regression analysis and plotted the resulting trend line in the figure. This calculates a linear trend line using the number of transitions as an x variable. It starts at a slowdown of 9,7 and has an increase of 0,23 for every million transitions.

The increase of slowdown that occurs with tests before 100 thousand states and from 1 million and up can be explained by garbage collection. When the memory consumption of the JVM increases it will spend time freeing memory. Unfortunately this will occur even when the maximal heap space is available. Because garbage collection does not occur in LTSmin it will increase the runtime compared the LTSmin.

It is difficult to determine what is causing the decrease in the middle of the plot, but we can make make a suggestion. As shown in [7] it is possible that a JVM can cause shifts in the execution speeds of functions. This means it is possible that certain functions can suddenly become faster during runtime. If this were to occur to a function that is used very often during execution and it can result in a decreased slowdown. When this function remains in this fast execution state and it is called more often, because more transition are passing through it, it will cause less slowdown when more transition pass through it. This would explain the decrease in slowdown of tests with between 100 thousand and 1 million transitions.

The goal of this performance test is to determine the slowdown that can be expected when the Java bridge is used. We can use the linear regression line to get an idea of what can be expected, but we will also look at the mean of the performance tests. A Java → C run is on average 11,3 times slower than LTSmin. Later we will compare this number to other configurations. To do this fairly we will use the 95% confidence interval of the mean. This is shown in table 11.

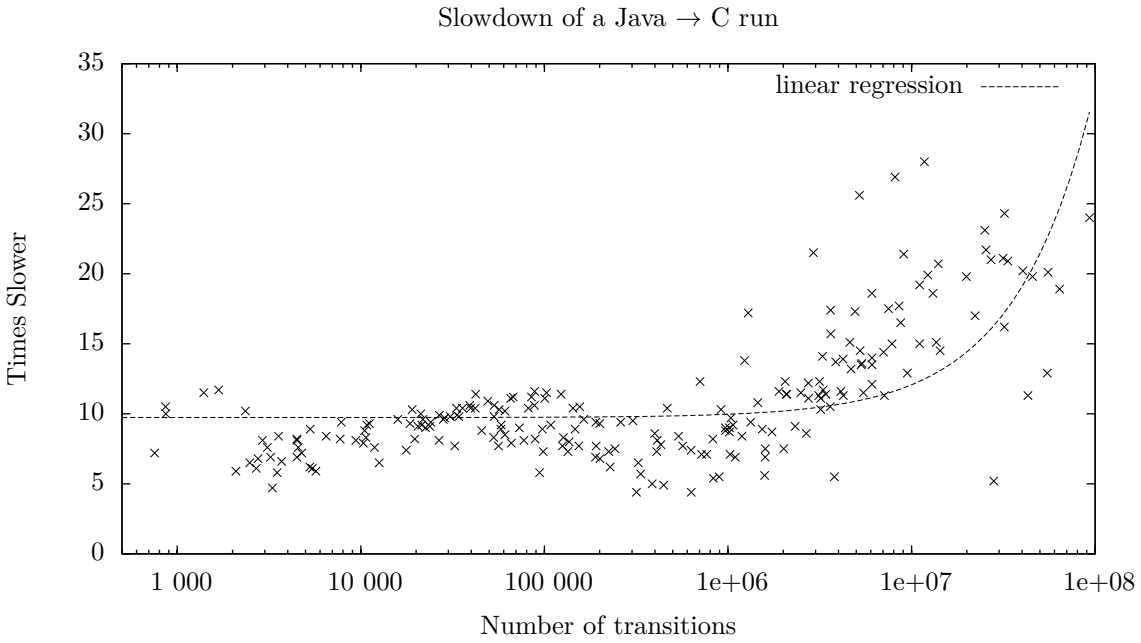



Figure 16: Plot of the of slowdown of a Java → C run on a logarithmic scale

### 5.3.3 Explanation of the slowdown

When starting this project we already took a performance loss into account when languages were being bridged. This was because JNI calls have some overhead which will make them slower than regular calls. There is also time spent converting between C and Java arrays. This is also considered as part of the cause of the slowdown.

To make sure the slowdown was not caused by something other than the JNI overhead we measured the time lost by JNI calls. We did this by measuring the time a JNI call takes in one language. We also measured the time the same call takes in the other language. When subtracting the latter from the former we determined the overhead of the JNI call. We also measured the time that was spent converting arrays. For this we are measuring the total time spent converting arrays, not only the JNI overhead for the conversion. This is because converting arrays is not performed in LTSmin. It is considered a slowdown fundamental to bridging languages.

For the JNI overhead we will measure the overhead of the `getTransitions` calls and the overhead of the callback that transfers the transitions from C to Java. We will also look at the time that is spent in the language module. This is done by measuring the time of the `getTransitionsAll` to the LTSmin `greybox` and subtracting the time spent by the callback. Note that this refers to the C callback instead of the Java callback. Therefore the time spent by the language module can be compared between the Java bridge and the original LTSmin implementation.

We used the `lann.5` model to run the test. It has around a million states and 3.6 million transitions. The measurements are show in table 12. Because both runs use the same exploration strategy they will make the same amount of `getTransitionsAll` calls. This is one `getTransitionsAll` call for each state. In the Java bridge every `getTransitionsAll` call is a JNI calls. The callback and the array conversion are also JNI calls.

The overall runtime using the Java bridge is 5 times slower than the runtime of LTSmin. When we look at the JNI overhead we see that the JNI overhead of the callback takes the most time. Almost half of the runtime is spent on overhead of this JNI call. When we divide the total amount of overhead by the number of transitions we can tell that an average of 5,3 $\mu$s per JNI call is spent. For the `getTransitionAll` call this is 3,3 $\mu$s. When the exploration time is subtracted by the JNI overhead and the time spent converting arrays we get 9,5 seconds (this calculation was performed before rounding). This is much closer to the exploration time of LTSmin which is 6,4 seconds.

| | Java Bridge | LTSmin |
|---|---|---|
| total runtime | 40,4 s | 8,1 s |
| exploration time | 37,9 s | 6,4 s |
| runtime - exploration time | 2,4 s | 1,7 s |
| | | |
| JNI overhead getTransitionAll | 3,3 s | |
| JNI overhead callback | 19,2 s | |
| array conversion overhead | 6,0 s | |
| total JNI overhead | 28,5 s | |
| | | |
| exploration - JNI overhead | 9,5 s | |
| | | |
| time in language module | 1,3 s | 0,3 s |
| | | |
| exploration not in language module | 8,2 s | 6,1 s |

Table 12: Measurements to explain the slowdown of the Java bridge

When we compare the time spent in the language module we notice that the Java bridge spends 1 second more than LTSmin. This is interesting because its language module uses the same code in both the Java bridge and LTSmin. However the difference is that in the Java bridge the language module is inside a shared object and in LTSmin everything is in the same executable. Calls within shared objects are slower performing than calls within the same executable. This is because the machine code for a shared object is different because it can be shared by different processes.

When we also subtract the time spent in the language module from the exploration time, we get a time of 8,2 seconds for the Java bridge and 6,1 seconds for LTSmin. There is not as much difference between these numbers as there was for the original runtimes. This means that the slowdown has almost fully been explained. The remaining difference can be attributed to the difference in performance of Java and C. A big part of this might be that Java has a garbage collector that does automatic memory management and in C the developer programs the memory management. We did give the JVM a sufficient heap space, but when we profiled a run we noticed that garbage collection was still taking place.

### 5.3.4 Comparing Java → C with caching to Java → C without cache

Part of the research is to determine whether the caching algorithm lowers the runtime by limiting the required JNI calls. However a problem is that the caching layer has to use `getTransitionsShort` calls instead of `getTransitionsAll` calls. `getTransitionsShort` have to be made one transition group at a time. Therefore the only way that the caching layer could reduce JNI call is when there are so many cache hits that less `getTransitionsShort` calls have to be made than `getTransitionsAll` calls would be made without caching. This is because when there is a cache hit it is not needed to request transitions from the C language module because they can be retrieved from the cache instead.

| Result for Java → C with caching | |
|---|---|
| Test Java → C completed | 242 |
| Java → C with caching completed | 237 |
| Java → C with and without caching completed | 233 |
| tests with less JNI calls | 174 |
| Java → C less getTransitions with caching | 75% |
| tests faster | 12 |
| average amount slower than Java → C without cache | 48% |
| 95% confidence of average slower | 52% – 45% |

Table 13: Results of Java → C with cache in numbers

The numbers of the results for Java → C with caching benchmarks are shown in table 13. 75% of the tests have less `getTransitions` calls when caching was enabled. However only 12 of the tests are actually faster than the tests without caching. On average a run with caching enabled is 48% slower.

This means that although caching is limiting JNI calls it is rarely beneficial to enable it. To explain why this is happening we will profile a Java → C run with caching which has few JNI calls but still a higher runtime than a run without caching. We will use the `lann.5` model. It only makes 1% of the `getTransitions` calls of a run without caching yet it is runs twice as long with caching.

When we profile this test we notice that around 60% of the CPU time is spent in the `convertState` method. This explains the increased runtime when caching is used. Although there are fewer JNI calls made there is a lot more time spent converting between long and short states. This is not something that can be avoided because it is fundamental to the caching algorithm that it stores short states. We have already seen that this also occurs in LTSmin runs with caching.

### 5.3.5 C → Java

We will compare a C → Java to LTSmin and a Java → C run. We will also calculate the number of times that C → Java is slower than LTSmin. From table 11 we can see that the confidence interval of the average slowdown is 38,6 – 53,2. This means that it is slower than a Java → C run and it might be slower than a Java → C run with caching. The 95% convidence interval of Java → C with caching and C → Java overlap which means that we can not be certain that it will be slower on average.

It is expected that a C → Java is slower than Java → C because it is both using JNI and the caching layer as the Java language module. From the previous tests we discussed we noticed that JNI calls and the caching layer are both causing slowdown.

### 5.3.6 C → Java with caching

This time the caching layer is on the C side. It has the same goal of eliminating JNI calls. With the confidence interval of the slowdown being 19,0 – 28,8 we can say that the C caching is making the tests run faster than without caching. The average slowdown is about half that of a C → Java run without caching. This means that even though `getTransitionsShort` calls are used instead of `getTransitionsAll`, a lot of JNI calls are being avoided. This is to be expected because the Java caching layer, which uses the same algorithm, also successfully reduced the number of JNI calls.

### 5.3.7 Java → Java

Because a Java → Java run is not using any JNI calls it can be used a measurement of what would happen if every transition is retrieved from cache. The confidence interval of the average slowdown is 32,2 – 45,8 times. This is similar to the interval for a Java → C run with caching, but a bit higher. Because we already determined that retrieving transitions from cache is actually slower than retrieving them with a JNI call it is expected that the Java → Java run is slower than a Java → C run with caching. That is because all transitions are retrieved from cache in a Java → Java run.

### 5.3.8 Memory usage

An important part of model checking is that it can be limited by memory usage. We measured the maximum memory usage of the benchmarks. The average ratio of the maximum memory usage between LTSmin and the benchmarks is shown in table 14. It can be seen that using Java will increase the maximum amount of memory used. An important note is that during the benchmarks the JVM was given the maximum amount of heap space, therefore it will not perform garbage collection as much. The result of this is that there is more memory usage.

The converting of states will generate garbage which is unlikely to be collected. We see this in the memory usage. When the Java caching layer is used in the Java → C run and when it is used as a a language module in the Java → Java run it is needed to convert the short states from the cache to long states. The memory usage is the highest for these tests. When the Java caching layer is used as a language module in the C → Java run without caching it is not needed to convert the short states to long states on the Java side. We see that this means that less garbage is generated and therefore the memory usage is lower. When less calls are made to Java cache when C caching is made there is also less

memory usage. This is because the caching layer will make a copy of the transition that is in the cache to retrieve it. This is done to ensure the cache in not edited when the provided transition is edited. This will also generate garbage. Another cause of high memory usage would be a memory leak, but we did not find this therefore we will assume that there is no leak.

In summary we can say that the usage of the Java caching layer will generate a lot of garbage which results in a lot of memory usage. This is due to the copying of the states from the cache and the converting to and from short states which is required to use the caching layer. The less it is used the less memory is used.

| Benchmark | Average times more memory | 95% confidence |
|---|---|---|
| LTSmin with caching | 1,3 | 1,1 – 1,5 |
| Java → C | 1,6 | 1,4 – 1,9 |
| Java → C with caching | 4,6 | 4,1 – 5,2 |
| C → Java | 2,5 | 2,2 – 2,7 |
| C → Java with caching | 1,6 | 1,4 – 1,9 |
| Java → Java | 4,6 | 4,0 – 5,2 |

Table 14: Table showing the 95% confidence interval of the mean of the ratio of the memory usage between LTSmin and various benchmarks

.

## 5.4 Ease of use

To evaluate the ease of use we are looking at the lines of code and number of steps to add certain modules. We exclude comments and blank lines from the lines of codes we mention. Table 15 shows the various modules and the lines of code it takes in LTSmin and in the Java bridge.

| | LTSmin | Java Bridge |
|---|---|---|
| language module | 200-400 | 93 |
| caching wrapper | 117 | 78 |
| search algorithms | 800 - 1100 | 258 |

Table 15: Table showing the lines of code of various modules

### 5.4.1 Language modules

For the Java implementation of a language module we are looking at the Java cache layer that retrieves the statespace from a file. This takes 166 lines of code. Because this caching layer also contains code to write the statespace to cache, we also made an implementation which can only read the serialized statespace. This implementation has 93 lines of code. This can be considered as the minimum amount of code needed to make a language module in Java. This is considerably less than the 200 lines minimum of a C language module. In practice a Java language module will probably require more than 93 lines of code because loading a serialized model does not require any translation from one type of model to the one presented by the Java bridge. However even when it takes more than three times as many lines it will still be considerably less than the 500 lines LTSmin sometimes requires.

Now we will examine the required steps to add a language module. To begin implementing a language module a class extending the `Greybox` class has to be made. The first object which it must initialize is the `LTSTypeSignature` object. This object contains the type information as discussed in section 4.2.5. The next object that a language module must provide is a `DependencyMatrix` as described in section 4.2.7.

Next the methods for retrieving states must be implemented. As a minimum the `getInitalState` and either `getTransitionsLong` or `getTransitionsShort` has to be implemented. As explained in section 3.3 when one of those `getTransitions` are implemented the other ones will work automatically.

When these methods also have been implemented the Java language module has to be used by the rest of the implementation. To do this it has to be added to the `GreyboxFactory` class. There the

relation between a file extension and the language module is made. This involves only adding a few lines of code to make this relation and the language module will be used automatically.

The steps required to add a Java language module are essentially the same as the ones that are needed in LTSmin. However in LTSmin a lot more lines of code are required to achieve the same result. When we inspect the code we see that a lot of lines are simulating a polymorphism by setting function pointers in structures. Because Java is object oriented, polymorphism is built-in and thus this code is not needed.

A big advantage to the Java bridge is that it is much more clear what needs to be done. In LTSmin there are various values which must be initialized. For some values this also must occur in a particular order. When values are not initialized or in the wronger order there will be no compilation errors. However there will be vague errors at runtime. In the Java bridge when values are not initialized it will not compile. Also the order of initialization is much less important. If an error would occur at runtime because of a wrong implementation it is much more clear what went wrong and where the error occurred.

Also, to add a new language module in LTSmin it is required to link it with the analysis algorithms to make it available as an additional executable. In Java language modules are used dynamically based on the file extension of the loaded model.

### 5.4.2 `Pins2Pins` wrappers

We have implemented a Java `Pins2Pins` wrapper in the form of a caching layer. This caching layer is implemented in 78 lines of code while the C caching in LTSmin takes 117 lines. Another wrapper that was implemented was wrapper which counted the amount of `getTransitions` calls. This wrapper used 35 lines of code. With the use of Java objects instead of C structures the code is also much easier to read.

To make a `Pins2Pins` wrapper the `GreyboxWrapper` class must be extended. Then any method which the wrapper changes can be overridden. Other methods will automatically use the implementation of the wrapped `Greybox`. The wrapped `Greybox` is also available to the implementation of the `Pins2Pins` wrapper. To make sure the `Pins2Pins` wrapper is used it also has to be added to the `GreyboxFactory`. Because a `Pins2Pins` wrapper is turned on based on a commandline parameter, this should also be added.

For adding new `Pins2Pins` wrappers it is needed to add new commandline parameters. This is done by adding a value to the `Parameter` enum. This can then be used as a key to add the parameter to the `CommandlineParameters` class. On a single line the parameters character, long name, description and possible values is added.

To add a commandline parameter to LTSmin a similar line has to be added describing it.

### 5.4.3 Analysis algorithms

For the analysis algorithm we implemented breadth first search and depth first search. LTSmin also has these algorithms, however LTSmin also has two other analysis algorithms specifically for LTL model checking. All of these algorithms are implemented twice. Each of these implementations is specified in one file. This means that BFS and DFS are implemented in the same file together with other analysis algorithms. This makes it difficult to compare the lines of code of the two algorithms fairly. One file has BFS and DFS and two other algorithms and it contains 1300 lines of code. The methods for the LTL analysis algorithms contain around 500 lines of code. The other file has BFS and DFS together with one other algorithm and it contains 1208 lines of code. Its other analysis algorithm takes around 60 lines of code. We will use the total lines of code with the code subtracted with the lines of code of the algorithms which the Java bridge does not have. The actual lines of code BFS and DFS in LTSmin require will be somewhat less because there is still some code of the other algorithms interleaved.

In the Java bridge the code for BFS and DFS has 258 lines of code. This is considerably less than the 800 or 1100 for the LTSmin implementation when the other algorithms are subtracted. When we inspect the code we see that a lot of lines of code deal with setting function pointers in structures. Again this is used to simulate polymorphism. Another big part of the code deals with callback mechanisms as the one explained in section 3.4. It is also needed to have different code for every statespace storage technique that is used. In the Java implementation the storage can just be performed by any `Set` implementation. This means that there is no need for changes in the algorithm when the storage technique is changed.

Since Java sets also do not use internal iteration, the code for iterating over a set is also much smaller and easier.

In LTSmin all analysis algorithms are added to a single file. Because they are all search algorithms it is possible to reuse some of the code. To add an entirely different implementation of an analysis algorithm it is needed to compile LTSmin differently. Every language module has to be compiled with the new analysis algorithm to make new executables.

In the Java bridge there is the `SearchAlgorithm` abstract class which can be extended for different search algorithms. There are 4 methods which have to be implemented by a search algorithm. When this is done search algorithm should be added to `AnalysisAlgorithmFactory`. This way it is possible to select the new search algorithm by setting the `-analysis` commandline parameter.

To implement an entirely different analysis algorithm a class implementing the `AnalysisAlgorithm` interface must me made. This only requires implementing one method which returns a `Result`. This `AnalysisAlgorithm` should also be added to the `AnalysisAlgorithmFactory` to allow it to be used.

## 5.5 Maintainability

To evaluate the maintainability of the Java bridge we look at various scenarios that can occur and require maintenance of the Java bridge. For each of these scenarios we will look at the number of steps required to complete the scenario.

### 5.5.1 Extending the Pins interface

When a method is added to the Pins interface of LTSmin it should also be added to the Pins interface of the Java bridge. In LTSmin the method declaration is added to the `greybox` header. In the Java bridge the method declaration should be added to the `Greybox` abstract class. This means that every class extending `Greybox` should implement the new method. The Java language modules have to provide a Java implementation of the method.

The `NativeGreybox` class should use the implementation of LTSmin. It does this by declaring the method `native`. Now Jace should also add this method to the `NativeGreybox` peer class. This way a C++ implemention of the method can be made which can call the original LTSmin method. To add the method to the peer class an Apache ANT script has to be executed [5]. This will automatically generate the appropriate headers and sources for the peer class. Now the C++ implementation of the method can be added to the peer class. There it is might be needed to perform type conversions. Java primitives are automatically converted and arrays can be converted using the method that are included in the Java bridge. Using the converted types the original method of LTSmin can be called.

In summary the following steps need to be taken:

1. Add the method to the `Greybox` class
2. Implement the method in Java language modules
3. Declare the method as `native` in `NativeGreybox`
4. Run Jace using the ANT script
5. Implement C++ method wrapper

The implementation of the method in Java and the C++ method wrapper in the `NativeGreybox` peer class are the steps that take the most work. The other ones only require adding a declaration or running a script. How much work adding the method to Java takes precisely depends on how complicated this method is. When it adds a fundamental feature to LTSmin it might be needed to add new classes and methods to various parts of the Java implementation. When it is less complicated only a few lines of code might be needed to implement the method in Java.

### 5.5.2 Adding a parameter to a Pins interface method

Adding a parameter requires similar steps to adding a method. However instead of adding declaration it is needed to change existing declarations. Since the method already exists in the `NativeGreybox` peer class it is only needed to modify the method by adding the parameter.

The following steps need to be taken:

1. Add the parameter in the `Greybox` class
2. Add the parameter in Java language modules including `NativeGreybox`
3. Run Jace using the ANT script
4. Update the implementation of the C++ method wrapper

When the parameter is a primitive type it is easy to add to the different methods. Primitive C types can be mapped to some Java primitive. When it is a pointer to some structure, there is more work that needs to be done. It could be needed that a class representing the structure needs to be created. The C++ implementation of the wrapping peer method will need to convert this class back to a C structure.

### 5.5.3 Removing a method or parameter

Changing a method from the Pins interface causes the analysis algorithms which use it to not work anymore. Therefore it is unlikely that methods that are used often are changed. However it could still occur for example during the development of new features. This would require maintenance of the Java bridge.

To remove a method or parameter is easier than adding them. It requires the removal from the `Greybox`, Java language modules and the `NativeGreybox`. When the `NativeGreybox` is changed it is needed to run Jace. This will update the method declaration for the peer class. Now the implementation can be updated. The method or parameter can be removed. When a parameter is removed its accompanying type conversion can also be removed.

1. Remove the parameter/method from the `Greybox` class
2. Remove the parameter/method from the Java language modules including `NativeGreybox`
3. Run Jace using the ANT script
4. Update the implementation of the C++ peer class

### 5.5.4 Comparison to other possible implementations

There are other techniques to make a bridge between Java and C. Because we use Jace we do not need to write language bridging code manually. Code to make the bridge between a Java method and a C++ implementation in a peer class is automatically generated. When we would be using a technique that requires writing JNI code manually there is a lot more work involved with maintaining the bridge. Implementing Java methods in C manually requires a very specific declaration. When the declaration is wrong it is hard to tell what is wrong. Therefore it is much easier to use a tool like Jace which can generate these declarations.

Making JNI calls from C manually is also difficult. Making JNI calls might be needed to perform type conversions when a parameter is an object. Unfortunately it requires multiple calls to be able to perform one single JNI call, for example: a call to retrieve the class, a call to retrieve the method, a call to retrieve an instance and then performing the actual call. When the method call requires arguments, they too might need multiple calls to retrieve them. Unfortunately these calls are not type safe, which means errors will display as segmentation violations at runtime [13]. This means Jace will take away a lot of work by making it possible to make JNI calls easier. It is possible to make a call to a C++ proxy class which will result in a JNI call.

Besides using a different language bridging technique there also was the possibility to implement every language module in Java. However, the problem with this is that there is no generic interface to make calls to. When a change needs to be implemented in the `Pins` interface it is needed to make a

different implementation of that change for every LTSmin language module. Bridging from one `Pins` interface to another allows every language module behind this interface to work automatically.

Another possible solution would be not to make a bridge at all. We already decided that a bridge needs to be made to allow support of the existing language modules. If there was no bridge then the Java version would need new language modules in order to work. But this would make maintenance much easier. Of course there would be no bridge maintenance at all. However, some maintenance might be needed to ensure both the Java and C implementation have the same features. When a feature is added to LTSmin it only has to be implemented in Java. There is no need to run Jace or to update C++ method wrappers.

# 6 Conclusion

## 6.1 Performance

The goal of the performance tests was to determine whether the Java bridge is fast enough to be useful for research in model checking. There are many models which complete in under 10 seconds which allows fast development of new algorithms. Only 47 out of the 300 models from the BEEM database did not complete in under one hour. This includes models which ran out of memory before an hour. With the possibility of checking so many models we can conclude that the Java bridge is fast enough to be useful. Analysis algorithms can be implemented in Java and the models are checked with an average slowdown of 11,3.

The major cause of the slowdown is the fact that language are bridged using JNI. When new Java only language modules are added there will be no such slowdown. Then the Java bridge will function as a full Java implementation of LTSmin.

The other runs that made use of the Java caching layer were not that quick. This includes the runs that used the Java caching layer as a language module. We see the same when we look at the slowdown that is caused in an LTSmin run with caching. With this information we can conclude that the caching layer is not effective in increasing performance when the DVE language module is used. It does not always reduce the amount of JNI calls and when it does it adds so much overhead that it is not beneficial.

However, when the calls that the caching layer avoids are slow enough, it does have a benefit. The caching layer for the C $\rightarrow$ Java run was avoiding both the JNI call and the use of the Java caching layer as the language module. From the results we can see that these calls are slow enough to make the caching layer effective. It made the C $\rightarrow$ Java runs twice as fast.

The memory usage of Java is more than that of LTSmin. Because we gave the JVM the maximum amount of heap space we can not make definitive conclusions about its consequences. However, we can say that this is something that has to be taken into account when working with the Java bridge.

## 6.2 Ease of use

In the previous section we discussed the differences in the ease of use of LTSmin and the Java bridge. Additional modules take less lines of code to program in the Java bridge. This is for a big part because of the object oriented nature of Java. There is no code needed to simulate polymorphism. Also iteration using an external iterator also takes less lines of code compared to internal iteration with callback functions.

In the Java bridge, the language modules need to provide the same types of information as they do in LTSmin. This means that the number of steps is not lower. The same goes for analysis algorithms. In the most basic case only one method has to be implemented in both LTSmin and the Java bridge. However, in the Java bridge there is an abstract class or interface which must be implemented before the it can be compiled. This makes it easy to see what must be implemented. In LTSmin, there are methods calls which must be made, but whether this has been done correctly will only become visible at runtime. Therefore the steps that need to be made are more obvious in the Java bridge compared to LTSmin.

Another thing affecting the ease of use are the difference between Java and C. Java will often generate more and more detailed error messages at compile time. This allows developers to correct errors early on. Java also takes care of memory management, whereas in C the developer has to take care of this manually. There is also more functionality built-in Java's API then there is the standard C library. This together makes Java easier to use than C for a lot of developers.

In conclusion we can say that the Java bridge will make developing new modules for LTSmin much easier. The Java bridge provides an easy to use API to allow rapid development of new features.

## 6.3 Maintainability

The maintainability of the Java bridge depends greatly on the bridging technique used. Of course not having a bridge is the easiest to maintain, but then the features of the C implementation of LTSmin will be lost. Jace makes maintaining a bridge between C and Java much easier. Changes to the Java side are

automatically translated to C++ peer and proxy classes. There is no need to work with difficult JNI calls. With Jace Java methods can be implemented in C++ and it makes Java calls from C++ look like normal C++ calls.

We can conclude that although maintaining the Java bridge requires some work when the `Pins` interface changes, it is not overly work intensive. This will make it less likely that the Java bridge will remain broken after a `Pins` interface change.

## 6.4 Summary

The Java bridge makes it possible to add Java analysis algorithms and Java language modules to the LTSmin toolset. The Java architecture provides a more structured design of LTSmin. Developing new features is a lot easier when the Java bridge is used. Maintenance of the Java bridge is required, but is not very difficult.

The caching layer is only effective when the calls it avoid are really slow. It will add significant overhead which will decrease performance. JNI calls are not slow enough to benefit from caching.

# 7 Future work

Some features of LTSmin still have to be implemented in Java. Currently state labels are not implemented and the the trace to a deadlock state is not recorded. These features should not be that difficult to implement.

Other future work could look at improving the performance of the Java bridge. Currently a JNI call has to be made to copy an array from C to Java. It might be possible to implement some way of sharing memory between Java and C to avoid this JNI call. It could even be possible to avoid the need to copy every state to Java if it is possible for Java to use the native state.

Another area where performance improvement are needed is the caching layer. Currently it is slower to retrieve a state from cache than to make a JNI call to a C language module. It is likely that this is just because of the way the caching algorithm works. We also see that the C cache is slower than the DVE language module. However there is room for improvements. Some performance loss might be caused by the fact that the `BitSet` implementation that is used in the `DependencyMatrix` is not that fast. Another cause might be that many method calls are made to convert a state. The amount of garbage generated by the caching layer is also substantial. Either way it is likely that the performance of the Java bridge can be improved on various areas.

The other way to make caching faster is to make sure that the caching algorithm which only looks at variables that are read works. It showed that it is capable of having a lot more cache hits than the algorithm which stores the full short states. To make sure that it works, the possible bug in the DVE language module has to be resolved.

Other future work can focus on adding additional features. Currently it is not possible to pass commandline options from Java to the C `greybox` and the other way around. This would allow the Java bridge to take advantage of the `Pins2Pins` wrappers of LTSmin like regrouping and partial order reduction. The Java bridge can also implement these `Pins2Pins` wrappers. Other features than can be implemented are multi-core and distributed reachability in the Java bridge.

All these features are already implemented in LTSmin, but there is also room for entirely new features to be added to the Java bridge. Additional language modules can be added. Especially languages from Java based model checking tool. New model checking techniques can also be implemented.

In conclusion we can say that the Java bridge is an expandable model checking framework on which new features can easily be added.

# References

[1] Stephan Blom, Jaco van de Pol, and Michael Weber. Bridging the gap between enumerative and symbolic model checkers. *CTIT, University of Twente, Enschede, Technical Report TRCTIT-09-30*, 2009.

[2] Edmund Clarke. Model checking. In S. Ramesh and G Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 54–56. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0058022.

[3] Jace Community. jace - Code-generation framework that makes it incredibly easy to integrate C++ and Java code - Google Project Hosting. `http://code.google.com/p/jace/`.

[4] Jean Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. CADP a protocol validation and verification toolbox. In Rajeev Alur and Thomas Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Berlin / Heidelberg, 1996.

[5] The Appache Software Foundation. Apache ant. `http://ant.apache.org/`.

[6] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[7] Joseph Yossi Gil, Keren Lenz, and Yuval Shimron. A microbenchmark case study and lessons learned. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 297–308, New York, NY, USA, 2011. ACM.

[8] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. The Formal Specification Language mCRL2. In *In Proceedings of the Dagstuhl Seminar*. MIT Press, 2007.

[9] Jan Friso Groote and Alban Ponse. Proof Theory for muCRL: A Language for Processes with Data. In *Proceedings of the International Workshop on Semantics of Specification Languages (SoSL)*, pages 232–251, London, UK, UK, 1994. Springer-Verlag.

[10] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1 edition, September 2003.

[11] Sheng Liang. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.

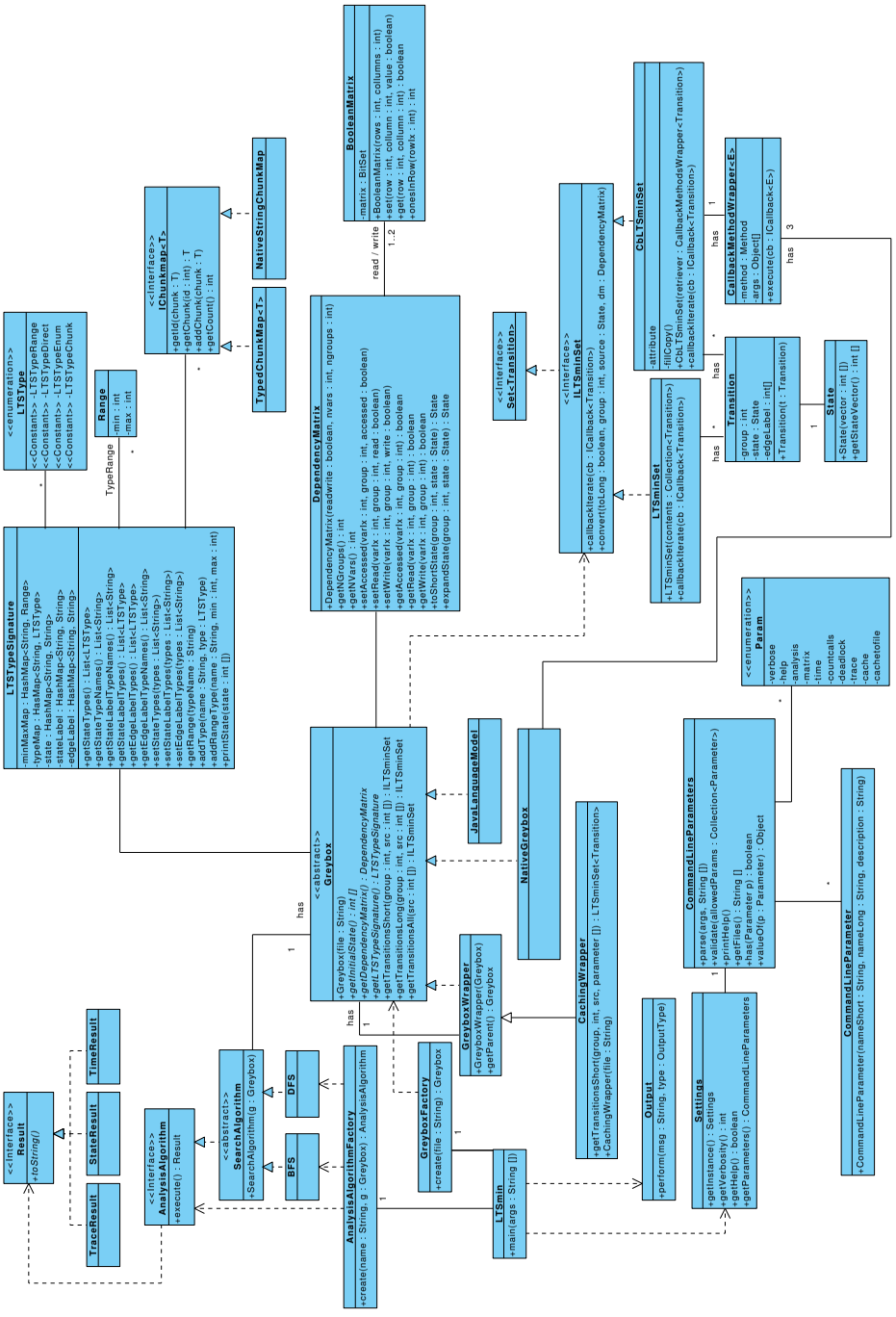
[12] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[13] Ruben Oostinga. Research topics: A java bridge for LTSmin. January 2012.

[14] Radek Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In Dragan Bošnacki and Stefan Edelkamp, editors, *Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer Berlin / Heidelberg, 2007.

[15] Michael Weber. An embeddable virtual machine for state space generation. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 168–186, Berlin, Heidelberg, 2007. Springer-Verlag.

# A Class Diagram



# B Invocation examples

Show all the commandline options

```
$ ./run.sh --help
Option                                  Description
------                                  -----------
--analysis                              Analysis algorithm = dfs | bfs
-c, --cache                             Smart cache: Store read variables,
                                          retrieve write variables
--cachetofile                           Cache all transitions to a specified
                                          file (--cachetofile file.dat)
--countcalls                            Count number of getTransition calls to
                                          greybox
-d, --deadlock                          Find a deadlock
-h, --help                              Print this message
-m, --matrix                            Prints the Dependency Matrix
--shortcache                            Cache short transitions
-t, --trace                             Find a trace to deadlock
--time                                  Measure the analysis time
-v, --verbosity [Integer]               Set the verbosity 0 - 2
```

View the contents of run.sh

```
$ cat run.sh
#!/bin/bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

. $DIR/config_local.sh

export LD_LIBRARY_PATH=$DIR/cpp_src/.libs
java -Xmx2G \
-cp $DIR/bin/enhanced:$DIR/lib/jopt-simple-4.3.jar:\
$JACE_HOME/core/runtime/target/jace-runtime-1.2.14.jar:\
$DIR/lib/guava-11.0.2.jar ltsmin.LTSmin $@
```

Run a full statespace exploration in BFS order on phils.1.dve

```
$ ./run.sh phils.1.dve
.: Precompiled divine module initialized
INFO: Level 1 has 4 states, explored 1 states 20 transitions
INFO: Level 2 has 10 states, explored 5 states 52 transitions
INFO: Level 3 has 16 states, explored 15 states 92 transitions
INFO: Level 4 has 15 states, explored 31 states 124 transitions
INFO: Level 5 has 12 states, explored 46 states 152 transitions
INFO: Level 6 has 10 states, explored 58 states 176 transitions
INFO: Level 7 has 4 states, explored 68 states 188 transitions
INFO: Level 8 has 4 states, explored 72 states 200 transitions
INFO: Level 9 has 4 states, explored 76 states 212 transitions
INFO: state space has 10 levels 80 states 212 transitions
INFO: Analysis finished
```

Run a full statespace exploration in DFS order on phils.1.dve

```
$ ./run.sh phils.1.dve --analysis dfs
.: Precompiled divine module initialized
INFO: state space has depth 43, 80 states and 212 transitions
INFO: Analysis finished
```

Find a deadlock in phils.1.dve

```
$ ./run.sh phils.1.dve -d
.: Precompiled divine module initialized
INFO: Level 1 has 4 states, explored 1 states 20 transitions
INFO: Level 2 has 10 states, explored 5 states 52 transitions
INFO: Level 3 has 16 states, explored 15 states 92 transitions
INFO: Level 4 has 15 states, explored 31 states 124 transitions
INFO: state space has 4 levels 46 states 124 transitions
INFO: State found
INFO: [fork[0]={1}, fork[1]={1}, fork[2]={1}, fork[3]={1},
phil_0={one}, phil_1={one}, phil_2={one}, phil_3={one}]
```

Print the dependency matrix of a model

```
$ ./run.sh loyd.1.dve --matrix
.: Precompiled divine module initialized
Dependency matrix:
+++++++r+-
+++++++r+-
++++++r++-
++++++r++-
rrrrrr---+
```

# C Performance tests

| Did not compile |
|---|
| plc.1 |
| plc.2 |
| plc.3 |
| plc.4 |
| train-gate.1 |
| train-gate.2 |
| train-gate.3 |
| train-gate.4 |
| train-gate.5 |
| train-gate.6 |
| train-gate.7 |

| Model | States | Transitions | C → C | C → C + Cache | Java → C | Java → C + Cache | C → Java | Java → Java | C → Java + Cache |
|---|---|---|---|---|---|---|---|---|---|
| adding.1 | 7372 | 11144 | 0,08 | 0,095 | 0,756667 | 0,986667 | 1,10333 | 1,155 | 1,15333 |
| adding.2 | 836838 | 1289748 | 0,77 | 2,92833 | 13,2633 | 18,3967 | 38,5567 | 18,83 | 36,3667 |
| adding.3 | 1894376 | 2921634 | 1,29333 | 6,15833 | 27,745 | 38,5567 | 88,83 | 43,0533 | 77,85 |
| adding.4 | 3370680 | 5201282 | 1,93667 | 11,205 | 48,0745 | 68,1033 | 159,073 | 74,8767 | 148,083 |
| adding.5 | 5271456 | 8135364 | 2,82667 | 19,25 | 76,08 | 103,717 | 260,82 | 119,243 | 237,333 |
| adding.6 | 7609684 | 11746148 | 3,91 | 25,43 | 111,56 | 154,387 | 389,84 | 177,493 | 352,627 |
| anderson.1 | 347039 | 704302 | 0,566667 | 1,79 | 6,97 | 6,59667 | 14,4167 | 6,90667 | 5,06 |
| anderson.2 | 1459 | 3705 | 0,0733333 | 0,0733333 | 0,486667 | 0,703333 | 0,633333 | 0,706667 | 0,483333 |
| anderson.3 | 75573925 | 388237977 | 143,897 | 824,03 | 3434,03 | 3354,05 | nomem | 3360,67 | 798,097 |
| anderson.4 | 29641 | 97516 | 0,176667 | 0,481667 | 1,57 | 1,72 | 2,44 | 1,69 | 0,96 |
| anderson.5 | 28143673 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| anderson.6 | 18206917 | 86996322 | 41,2433 | 276,19 | 643,167 | 785,507 | nomem | 745,753 | 273,17 |
| anderson.7 | 28558175 | | nomem | nomem | nomem | timeout | notserialized | notserialized | notserialized |
| anderson.8 | 27858852 | | nomem | nomem | timeout | nomem | notserialized | notserialized | notserialized |
| at.1 | 39354 | 108438 | 0,19 | 0,881667 | 1,74 | 2,47 | 3,46 | 2,54333 | 1,40333 |
| at.2 | 49443 | 146940 | 0,226667 | 1,05833 | 2,02 | 2,80667 | 4,01667 | 2,80667 | 1,52667 |
| at.3 | 1711620 | 6075360 | 2,49333 | 27,2833 | 46,35 | 76,8433 | 132,15 | 74,89 | 27,41 |
| at.4 | 6597245 | 25470140 | 8,86 | 132,327 | 191,89 | 373,627 | 587,3 | 365,503 | 133,63 |
| at.5 | 31999440 | 125231180 | 45,6867 | 662,263 | 930,727 | 1828,97 | nomem | 1789,02 | 645,913 |
| at.6 | 37412761 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| at.7 | 28881350 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| bakery.1 | 1506 | 2697 | 0,0733333 | 0,11 | 0,45 | 0,693333 | 0,726667 | 0,75 | 0,563333 |
| bakery.2 | 1146 | 2085 | 0,07 | 0,07 | 0,41 | 0,696667 | 0,66 | 0,72 | 0,496667 |
| bakery.3 | 32919 | 85061 | 0,17 | 0,56 | 1,90333 | 1,89667 | 2,73333 | 1,84333 | 1,25333 |
| bakery.4 | 157003 | 411843 | 0,496667 | 1,89333 | 4,14 | 5,31667 | 9,22667 | 5,43667 | 2,96667 |
| bakery.5 | 7866401 | 27018304 | 9,85 | 102,05 | 206,935 | 269,597 | 500,847 | 263,63 | 107,257 |
| bakery.6 | 11845035 | 40400559 | 15,05 | 154,4 | 306,47 | 401,523 | 749,487 | 402,477 | 155,83 |
| bakery.7 | 29047471 | 100691444 | 40,8367 | 395,957 | 778,563 | 1002,1 | nomem | 967,7 | 396,07 |
| bakery.8 | 30442698 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| blocks.2 | 7057 | 18552 | 0,0866667 | 0,205 | 0,806667 | 1,44667 | 1,67 | 1,51667 | 1,94667 |
| blocks.3 | 695418 | 2094753 | 1,56333 | 16,95 | 17,83 | 61,1067 | 101,22 | 60,1333 | 145,06 |
| blocks.4 | 23598882 | | nomem | nomem | timeout | nomem | notserialized | notserialized | notserialized |
| bopdp.1 | 12642 | 24039 | 0,103333 | 0,671667 | 0,97 | 2,16333 | 2,33333 | 2,21333 | 1,35 |
| bopdp.2 | 25685 | 72968 | 0,166667 | 1,16 | 1,81333 | 3,30333 | 3,77 | 3,36 | 1,64 |
| bopdp.3 | 1040953 | 2747408 | 2,10333 | 31,735 | 23,4767 | 85,71 | 110,693 | 86,2767 | 31,1567 |
| bridge.1 | 3186 | 4565 | 0,0733333 | 0,135 | 0,56 | 1,01667 | 1,27667 | 1,07333 | 1,03333 |
| bridge.2 | 96923 | 191360 | 0,326667 | 2,57667 | 2,50333 | 7,51667 | 9,18333 | 7,14667 | 4,47667 |
| bridge.3 | 838864 | 1896973 | 1,44333 | 27,0583 | 16,7433 | 63,63 | 85,0133 | 61,34 | 31,7167 |
| brp.1 | 18928 | 35772 | 0,123333 | 0,516667 | 1,28 | 1,68667 | 2,01333 | 1,655 | 1,24 |
| brp.2 | 29188 | 58151 | 0,15 | 0,671667 | 1,38 | 2,02667 | 2,58333 | 1,99667 | 1,31667 |
| brp2.1 | 42285 | 60962 | 0,186667 | 1,08 | 1,59 | 2,88667 | 3,83 | 2,93667 | 2,45333 |
| brp2.2 | 61464 | 89007 | 0,23 | 1,23667 | 1,87667 | 3,64333 | 4,83333 | 3,72 | 2,90667 |
| brp2.3 | 40184 | 57966 | 0,176667 | 1,03167 | 1,58 | 2,92 | 3,63333 | 2,78667 | 2,3 |
| brp2.4 | 679993 | 1065222 | 1,24 | 10,16 | 11,4133 | 29,4667 | 42,9167 | 29,2633 | 20,6 |
| brp2.5 | 298111 | 430858 | 0,71 | 4,62167 | 5,54333 | 14,1267 | 18,7733 | 14,1367 | 9,08667 |
| brp2.6 | 5742313 | 9058624 | 4,08667 | 77,0533 | 87,35 | 220,157 | 348,367 | 222,187 | 139,82 |
| brp.3 | 996627 | 2047490 | 1,55333 | 14,585 | 19,13 | 35,7033 | 59,5267 | 34,95 | 15,59 |
| brp.4 | 12068447 | 25085950 | 9,58 | 193,437 | 223,42 | 450,85 | 713,457 | 433,767 | 201,277 |
| brp.5 | 17740267 | 36903290 | 17,63 | 289,92 | 318,783 | 643,723 | 1058,37 | 651,633 | 290,793 |
| brp.6 | 42728113 | 89187437 | 36,2133 | 789,757 | 773,787 | 1780,53 | nomem | 1742,39 | 767,043 |
| cambridge.1 | 11339 | 26768 | 0,106667 | 0,461667 | 1,05333 | 1,79667 | 2,13667 | 1,84 | 1,57667 |
| cambridge.2 | 15940 | 60907 | 0,13 | 0,671667 | 1,33 | 2,08 | 2,86 | 2,2 | 1,72 |
| cambridge.3 | 18138 | 45536 | 0,136667 | 0,745 | 1,2 | 2,27667 | 2,80333 | 2,41333 | 2,16667 |
| cambridge.4 | 60463 | 153956 | 0,286667 | 1,985 | 2,16667 | 5,13333 | 6,98667 | 5,63 | 4,15667 |
| cambridge.5 | 698912 | 3199507 | 2,32667 | 24,88 | 23,8633 | 58,62 | 87,35 | 58,7867 | 39,8633 |
| cambridge.6 | 3354295 | 9483191 | 6,11667 | 126,903 | 80,5733 | 286,137 | 382,017 | 289,387 | 187,277 |
| cambridge.7 | 11465015 | 54850496 | 31,58 | 564,747 | 400,563 | 1255,71 | nomem | 1324,82 | 795,27 |
| collision.1 | 5593 | 10792 | 0,0766667 | 0,148333 | 0,706667 | 0,886667 | 1,03667 | 0,915 | 0,516667 |
| collision.2 | 12661 | 28144 | 0,106667 | 0,265 | 1,02 | 1,20333 | 1,51667 | 1,22 | 0,65 |
| collision.3 | 434530 | 1018734 | 1,07667 | 6,47 | 9,38333 | 16,29 | 25,3267 | 16,3733 | 6,79667 |
| collision.4 | 41465543 | 113148818 | 53,5633 | 756,62 | 931,57 | 1787,42 | nomem | 1772,3 | 751,34 |
| collision.5 | 19473613 | | nomem | nomem | nomem | nomem | notserialized | notserialized | notserialized |
| collision.6 | 18845059 | | nomem | nomem | nomem | timeout | notserialized | notserialized | notserialized |
| cyclic_scheduler.1 | 4606 | 20480 | 0,09 | 0,228333 | 0,823333 | 1,05333 | 1,26667 | 1,04333 | 0,56 |
| cyclic_scheduler.2 | 3302 | 7720 | 0,0733333 | 0,125 | 0,626667 | 0,836667 | 0,903333 | 0,77 | 0,423333 |
| cyclic_scheduler.3 | 229374 | 1597440 | 1,67667 | 11,7317 | 12,56 | 27,09 | 37,36 | 27,2 | 11,8933 |
| cyclic_scheduler.4 | 473414 | 1736712 | 1,66667 | 13,1067 | 14,21 | 28,67 | 42,5767 | 28,5567 | 12,6 |
| driving_phils.1 | 14889 | 28595 | 0,113333 | 0,416667 | 1,09667 | 1,49667 | 1,76 | 1,52333 | 0,956667 |
| driving_phils.2 | 33173 | 81854 | 0,183333 | 0,928333 | 1,64333 | 2,54667 | 3,29333 | 2,61667 | 1,37667 |
| driving_phils.3 | 23583477 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| driving_phils.4 | 21317183 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| driving_phils.5 | 18601771 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| elevator.1 | 8543 | 15914 | 0,09 | 0,298333 | 0,86 | 1,40667 | 1,72333 | 1,44667 | 1,02 |
| elevator.2 | 2825 | 5274 | 0,09 | 0,13 | 0,555 | 0,936667 | 0,993333 | 0,953333 | 0,71 |
| elevator2.1 | 1728 | 4768 | 0,0666667 | 0,0783333 | 0,483333 | 0,765 | 0,726667 | 0,746667 | 0,51 |
| elevator2.2 | 179200 | 1036800 | 0,806667 | 2,40167 | 7,81667 | 8,11667 | 15,6767 | 8,14667 | 3,95 |
| elevator2.3 | 7667712 | 55377920 | 18,02 | 92,1933 | 361,81 | 307,93 | 695,853 | 293,433 | 123,847 |
| elevator.3 | 416935 | 1025817 | 1,43 | 12,4483 | 10,1567 | 35,46 | 42,7833 | 34,7067 | 13,4933 |
| elevator.4 | 888053 | 2320984 | 2,41667 | 21,79 | 21,92 | 61,57 | 81,9167 | 60,2933 | 22,1367 |
| elevator.5 | 14082548 | | nomem | nomem | nomem | timeout | notserialized | notserialized | notserialized |
| elevator_planning.1 | 27630 | 163880 | 0,21 | 0,593333 | 2,01333 | 2,08 | 3,24 | 2,09333 | 1,31 |
| elevator_planning.2 | 11428767 | 93278857 | 38,79 | 205,61 | 932,38 | 2649,7 | nomem | timeout | 231,557 |
| elevator_planning.3 | 52496 | 466568 | 0,373333 | 1,05 | 3,87 | 3,22 | 6,33 | 3,3 | 1,85333 |
| exit.1 | 3239334 | 7491035 | 3,71333 | 58,8733 | 64,8467 | 177,05 | 247,52 | 173,727 | 61,2633 |
| exit.2 | 33670 | 88203 | 0,186667 | 1,74833 | 1,97667 | 4,95 | 5,54667 | 4,99 | 2,54 |

| Model | States | Transitions | C → C | C → C + Cache | Java → C | Java → C + Cache | C → Java | Java → Java | C → Java + Cache |
|---|---|---|---|---|---|---|---|---|---|
| exit.3 | 2356294 | 7047332 | 3,83667 | 88,99 | 56,4533 | 244,727 | 302,403 | 247,133 | 90,75 |
| exit.4 | 24177544 | | nomem | timeout | timeout | timeout | notserialized | notserialized | notserialized |
| exit.5 | 833226 | 3194881 | 2,24333 | 49,33 | 24,79 | 133,027 | 161,923 | 132,797 | 48,9033 |
| extinction.1 | 8993 | 23750 | 0,103333 | 0,666667 | 0,943333 | 1,85 | 2,02 | 1,78667 | 1,23333 |
| extinction.2 | 10061 | 26683 | 0,126667 | 0,933333 | 1,025 | 2,13 | 2,24333 | 2,09667 | 1,51667 |
| extinction.3 | 751930 | 2669267 | 2,61 | 50,77 | 22,5233 | 106,223 | 124,42 | 101,457 | 47,63 |
| extinction.4 | 2001372 | 7116790 | 5,13 | 142,93 | 60,1533 | 293,233 | 363,48 | 294,387 | 137,517 |
| firewire_link.1 | 1724 | 3301 | 0,113333 | 0,668333 | 0,533333 | 2,38667 | 2,73 | 2,45667 | 2,41333 |
| firewire_link.2 | 55887 | 134271 | 0,3 | 12,9983 | 2,18333 | 32,85 | 31,8233 | 33,7033 | 16,0767 |
| firewire_link.3 | 5683833 | | nomem | timeout | timeout | timeout | notserialized | notserialized | notserialized |
| firewire_link.4 | 24330 | 56219 | 0,193333 | 9,79667 | 1,49333 | 25,8667 | 25,3967 | 26,5133 | 12,3867 |
| firewire_link.5 | 3371219 | 59782059 | timeout | timeout | 489,24 | timeout | notserialized | notserialized | notserialized |
| firewire_link.6 | 3076225 | | timeout | timeout | timeout | timeout | notserialized | notserialized | notserialized |
| firewire_link.7 | 399598 | 1096535 | 1,68333 | 236,308 | 11,5567 | 563,87 | 594,337 | 562,14 | 232,39 |
| firewire_tree.1 | 272 | 864 | 0,07 | timeout | 0,733333 | 1,43667 | 1,8 | 1,33333 | 1,41333 |
| firewire_tree.2 | 2441 | 5692 | 0,103333 | 0,871667 | 0,61 | 2,14667 | 2,23 | 2,26667 | 1,81 |
| firewire_tree.3 | 86556 | 317063 | 0,986667 | 67,9183 | 4,37667 | 139,657 | 138,357 | 145,647 | 63,9667 |
| firewire_tree.4 | 169992 | 630811 | 1,91 | 222,123 | 8,34 | 450,02 | nomem | 477,31 | 208,553 |
| firewire_tree.5 | 776538 | 18225703 | nomem | nomem | 252,683 | timeout | notserialized | notserialized | notserialized |
| fischer.1 | 634 | 1395 | 0,0633333 | 0,0666667 | 0,726667 | 0,716667 | 0,56 | 0,596667 | 0,403333 |
| fischer.2 | 21733 | 67590 | 0,136667 | 0,343333 | 1,53 | 1,41667 | 2,00333 | 1,40333 | 0,77 |
| fischer.3 | 2896705 | 12280586 | 4,38 | 33 | 89,2767 | 95,82 | 197,173 | 95,64 | 34,3633 |
| fischer.4 | 1272254 | 4609671 | 2,33 | 17,4233 | 35,2467 | 50,4567 | 88,9667 | 49,48 | 19,7867 |
| fischer.5 | 31077246 | 477823470 | 171,997 | 1384,44 | 3418,05 | timeout | notserialized | notserialized | notserialized |
| fischer.6 | 8321728 | 33454191 | 11,7233 | 129,463 | 245,09 | 377,317 | 654,187 | 368,897 | 138,317 |
| fischer.7 | 28113001 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| frogs.1 | 5094 | 5301 | 0,0733333 | 0,14 | 0,65 | 1,21667 | 1,55 | 1,54667 | 1,77667 |
| frogs.2 | 18207 | 33209 | 0,113333 | 0,388333 | 1,18333 | 2,28333 | 3,35333 | 3,04667 | 4,41333 |
| frogs.3 | 760789 | 766119 | 1,45 | 15,3383 | 10,2233 | 51,26 | 109,243 | 75,9133 | 176,293 |
| frogs.4 | 17443219 | 36286061 | 26,18 | nomem | 319,815 | nomem | notserialized | notserialized | notserialized |
| frogs.5 | 20311538 | | nomem | nomem | timeout | nomem | notserialized | notserialized | notserialized |
| gear.1 | 2689 | 3567 | 0,0633333 | 0,13 | 0,53 | 1,01667 | 1,14 | 1,14 | 1,13333 |
| gear.2 | 16689 | 21767 | 0,103333 | 0,53 | 0,995 | 2,02 | 2,33 | 2,27333 | 2,03333 |
| hanoi.1 | 6561 | 19680 | 0,0966667 | 0,198333 | 0,79 | 1,44 | 1,74667 | 1,60333 | 1,97667 |
| hanoi.2 | 531441 | 1594320 | 2,03333 | 9,915 | 14,2933 | 42,75 | 80,9067 | 50,25 | 104,533 |
| hanoi.3 | 14348907 | 43046718 | 31,73 | timeout | 357,24 | nomem | notserialized | notserialized | notserialized |
| hanoi.4 | 18890396 | | nomem | nomem | nomem | nomem | notserialized | notserialized | notserialized |
| iprotocol.1 | 6814 | 22512 | 0,0933333 | 0,23 | 0,84 | 1,02333 | 1,24667 | 1,13667 | 0,69 |
| iprotocol.2 | 29994 | 100489 | 0,183333 | 0,825 | 2,03333 | 2,08333 | 3,02667 | 2,235 | 1,40667 |
| iprotocol.3 | 1013456 | 3412754 | 2,36667 | 17,8617 | 26,9967 | 39,5167 | 72,4833 | 39,3733 | 18,2833 |
| iprotocol.4 | 3290916 | 11071177 | 5,67667 | 60 | 85,325 | 129,93 | 234,977 | 127,553 | 59,22 |
| iprotocol.5 | 31071582 | 104572634 | 59,5667 | 610,573 | 819,89 | 1265,06 | nomem | 1274,6 | nomem |
| iprotocol.6 | 41387484 | 139545158 | 76,96 | 820,543 | 1113,89 | 1694,67 | nomem | 1652,65 | nomem |
| iprotocol.7 | 24673777 | 200828479 | timeout | timeout | 1665,77 | 2795,64 | nomem | 2719,31 | nomem |
| krebs.1 | 6027 | 19040 | 0,0766667 | 0,135 | 0,786667 | 1,11 | 1,47 | 1,15 | 1,18 |
| krebs.2 | 62476 | 260542 | 0,296667 | 0,97 | 2,85 | 3,13 | 5,43667 | 3,19333 | 2,92333 |
| krebs.3 | 238876 | 1020147 | 0,89 | 2,79667 | 8,01667 | 9,13667 | 17,2833 | 10,0333 | 8,28 |
| krebs.4 | 1047405 | 5246321 | 2,58 | 12,3833 | 37,0433 | 37,2733 | 77,9367 | 36,1667 | 19,9933 |
| lamport.1 | 29242 | 77286 | 0,193333 | 0,606667 | 1,56 | 1,85 | 2,57333 | 1,96333 | 1,03 |
| lamport.2 | 110920 | 303058 | 0,363333 | 1,83833 | 3,46667 | 5,06 | 7,45667 | 5,08667 | 2,18333 |
| lamport.3 | 38067 | 102747 | 0,183333 | 0,798333 | 1,73 | 2,22333 | 3,15667 | 2,3 | 1,21 |
| lamport.5 | 1066800 | 3630664 | 1,87667 | 16,9883 | 29,4767 | 47,3133 | 75,7133 | 45,47 | 17,4167 |
| lamport.6 | 8717688 | 31502176 | 10,7 | 141,683 | 227,567 | 373,483 | 650,78 | 362,095 | 140,747 |
| lamport.7 | 38717846 | 160667630 | 61,3867 | 833,24 | 1169,87 | 2173,4 | nomem | 2132,03 | 817,29 |
| lamport.8 | 62669317 | | 99,6633 | 1382,06 | nomem | timeout | notserialized | notserialized | notserialized |
| lamport_nonatomic.1 | 20434 | 65534 | 0,136667 | 1,365 | 1,52333 | 3,84667 | 4,2 | 3,91667 | 2,02 |
| lamport_nonatomic.2 | 12958 | 41991 | 0,106667 | 0,955 | 1,06667 | 2,85667 | 3,20667 | 2,89333 | 1,52 |
| lamport_nonatomic.3 | 36983 | 123337 | 0,21 | 2,13667 | 2,21667 | 6,28 | 6,73333 | 6,27667 | 2,73333 |
| lamport_nonatomic.4 | 1257304 | 5360727 | 3,09667 | 110,585 | 42,065 | 285,733 | 318,823 | 284,527 | 110,407 |
| lamport_nonatomic.5 | 3488300 | | nomem | nomem | nomem | timeout | notserialized | notserialized | notserialized |
| lann.1 | 18424 | 39673 | 0,116667 | 0,348333 | 1,25 | 1,51 | 1,71 | 1,43667 | 0,79 |
| lann.2 | 12784 | 34192 | 0,103333 | 0,388333 | 1,01667 | 1,51667 | 1,63 | 1,37667 | 0,71 |
| lann.3 | 1832139 | 8725188 | 3,85 | 28,0067 | 63,58 | 71,6767 | 140,9 | 69,995 | 27,7867 |
| lann.4 | 966855 | 3189852 | 2,41667 | 20,79 | 27,705 | 50,5633 | 75,8933 | 49,9533 | 20,8 |
| lann.5 | 993914 | 3604487 | 2,71667 | 27,4017 | 28,4433 | 63,9133 | 92,4967 | 62,1 | 25,73 |
| lann.6 | 57120827 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| lann.7 | 52237514 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| lann.8 | 45091872 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| leader_election.1 | 14252 | 52944 | 0,163333 | 1,72333 | 1,35 | 3,19 | 3,53667 | 3,32333 | 2,00333 |
| leader_election.2 | 28720 | 98528 | 0,25 | 3,01833 | 1,835 | 5,65667 | 6,07667 | 5,67333 | 3,30333 |
| leader_election.3 | 101360 | 446024 | 1,01 | 14,2533 | 4,91 | 24,2467 | 25,2933 | 24,53 | 13,71 |
| leader_election.4 | 746240 | 3795388 | 6,53667 | 145,733 | 36,0633 | 215,167 | 235,403 | 221,283 | 132,05 |
| leader_election.5 | 4803952 | 28064092 | 48,9867 | 1280,04 | 254,35 | 1897,9 | nomem | 1838,96 | 1137,95 |
| leader_election.6 | 5672319 | | timeout | timeout | nomem | timeout | notserialized | notserialized | notserialized |
| leader_filters.1 | 4966 | 9387 | 0,0833333 | 0,165 | 0,686667 | 1,09 | 1,18333 | 1,08333 | 0,77 |
| leader_filters.2 | 29284 | 66042 | 0,183333 | 0,646 | 1,42 | 2,03667 | 2,76333 | 2,005 | 1,30333 |
| leader_filters.3 | 91093 | 223980 | 0,363333 | 2,03333 | 2,67 | 5,41667 | 7,53667 | 5,27333 | 2,8 |
| leader_filters.4 | 50025 | 126784 | 0,236667 | 1,28833 | 1,97333 | 3,28333 | 4,36333 | 3,305 | 1,8 |
| leader_filters.5 | 1572886 | 4684565 | 2,88333 | 36,71 | 37,7967 | 89,9967 | 132,963 | 92,4 | 38,18 |
| leader_filters.6 | 28480454 | | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| leader_filters.7 | 26302351 | 91692858 | 47,7133 | 778,957 | 688,13 | 1878,84 | nomem | 1827,75 | nomem |
| lifts.1 | 2661 | 4486 | 0,09 | 0,318333 | 0,616667 | 1,43667 | 1,56667 | 1,53 | 1,55 |
| lifts.2 | 2664 | 4490 | 0,0733333 | 0,3 | 0,596667 | 1,45 | 1,49333 | 1,43 | 1,39333 |
| lifts.3 | 30772 | 53247 | 0,17 | 3,475 | 1,80667 | 8,86 | 8,465 | 8,87333 | 5,18 |
| lifts.4 | 112792 | 242370 | 0,436667 | 11,6767 | 3,285 | 27,32 | 27,73 | 27,03 | 13,66 |
| lifts.5 | 191567 | 409137 | 0,683333 | 19,86 | 5,01667 | 45,2867 | 47,0067 | 43,1333 | 22,1533 |
| lifts.6 | 333649 | 717892 | 1,14 | 49,8133 | 7,85333 | 108,067 | 108,38 | 103,623 | 51,8433 |
| lifts.7 | 5126781 | 13631916 | 7,81333 | 764,663 | 118,207 | 1599,95 | 1671,13 | 1594,31 | 759,457 |
| lifts.8 | 9582787 | 31993172 | 16,8133 | 1837,18 | 272,487 | timeout | notserialized | notserialized | notserialized |
| loyd.1 | 720 | 1681 | 0,06 | 0,065 | 0,7 | 0,67 | 0,67 | 0,673333 | 0,613333 |
| loyd.2 | 362880 | 967681 | 0,95 | 2,9 | 8,57 | 13,97 | 29,02 | 16,335 | 32,8167 |
| loyd.3 | 30247886 | | nomem | nomem | timeout | nomem | notserialized | notserialized | notserialized |
| lup.1 | 1404 | 2484 | 0,0666667 | 0,106667 | 0,435 | 0,96 | 1,09667 | 1,04667 | 1,32 |
| lup.2 | 495720 | 915624 | 0,913333 | 32,995 | 9,40667 | 138,603 | nomem | 125,213 | nomem |
| lup.3 | 1948617 | 3621672 | 2,05667 | 172,145 | 36,0333 | nomem | notserialized | notserialized | notserialized |
| lup.4 | 6618725 | 13974930 | 6,43333 | nomem | 133,237 | nomem | notserialized | notserialized | notserialized |
| mcs.1 | 7963 | 21503 | 0,0966667 | 0,215 | 0,9 | 1,06333 | 1,40667 | 1,17333 | 0,74 |
| mcs.2 | 1408 | 3222 | 0,0633333 | 0,09 | 0,436667 | 0,743333 | 0,723333 | 0,73 | 0,483333 |
| mcs.3 | 571459 | 2077384 | 1,47 | 10,255 | 16,75 | 28,3467 | 46,5833 | 28,16 | 10,93 |
| mcs.4 | 16384 | 53248 | 0,133333 | 0,496667 | 1,30333 | 1,69667 | 2,22 | 1,68 | 0,876667 |
| mcs.5 | 60556519 | | 120,287 | 1426,15 | nomem | timeout | notserialized | notserialized | notserialized |
| mcs.6 | 332544 | 1329920 | 1,13 | 7,92833 | 10,8433 | 21,4233 | 32,1 | 21,5833 | 8,12667 |

| Model | States | Transitions | C → C | C → C + Cache | Java → C | Java → C + Cache | C → Java | Java → Java | C → Java + Cache |
|---|---|---|---|---|---|---|---|---|---|
| msmie.1 | 2334 | 3097 | 0,0666667 | 0,15 | 0,506667 | 0,943333 | 1,05333 | 1,04 | 0,763333 |
| msmie.2 | 10558 | 11878 | 0,113333 | 0,886667 | 0,873333 | 2,42333 | 2,57 | 2,64 | 1,45333 |
| msmie.3 | 134844 | 200614 | 0,443333 | 11,1183 | 3,02333 | 28,835 | 29,5967 | 28,1933 | 12,1667 |
| msmie.4 | 7125441 | 11056210 | 6,09667 | 1274,97 | 117,26 | 2937,13 | 2843,66 | 2840,03 | 1221,94 |
| needham.1 | 497 | 753 | 0,08 | 0,423333 | 0,576667 | 0,763333 | 0,85 | 0,88 | 0,783333 |
| needham.2 | 54976 | 136377 | 0,243333 | 2,37667 | 1,93667 | 6,345 | 7,75 | 6,42333 | 3,17 |
| needham.3 | 206925 | 567099 | 0,73 | 9,12667 | 5,73667 | 23,0033 | 27,85 | 23,15 | 9,98333 |
| needham.4 | 6525019 | 22203081 | 9,94333 | 382,533 | 168,807 | 910,327 | 1114,64 | 888,443 | 375,34 |
| peg_solitaire.1 | 32181 | 155814 | 0,22 | 2,21167 | 2,30333 | 9,7 | 12,1767 | 8,70333 | 28,06 |
| peg_solitaire.2 | 13264450 |  | timeout | nomem | timeout | nomem | notserialized | notserialized | notserialized |
| peg_solitaire.3 | 5439599 |  | timeout | nomem | timeout | nomem | notserialized | notserialized | notserialized |
| peg_solitaire.4 | 873326 | 5473290 | 3,26333 | 97,6083 | 37,6367 | 319,453 | nomem | 525,76 | nomem |
| peg_solitaire.5 | 84191 | 324648 | 0,546667 | 29,3017 | 3,56 | 66,8033 | 95,8367 | 64,01 | 231,543 |
| peg_solitaire.6 | 7330153 |  | nomem | nomem | timeout | nomem | notserialized | notserialized | notserialized |
| peterson.1 | 12498 | 33369 | 0,103333 | 0,231667 | 1,02 | 1,215 | 1,64667 | 1,22333 | 0,706667 |
| peterson.2 | 124704 | 399138 | 0,456667 | 1,45 | 3,935 | 4,17667 | 7,47667 | 3,90333 | 1,89667 |
| peterson.3 | 170156 | 538509 | 0,603333 | 1,83333 | 5,04667 | 5,23333 | 9,68 | 5,095 | 2,42 |
| peterson.4 | 1119560 | 3864896 | 2,17 | 13,0633 | 29,6567 | 34,9967 | 70,6533 | 35,7933 | 14,23 |
| peterson.5 | 31656891 |  | 241,477 | 1520,69 | timeout | timeout | notserialized | notserialized | notserialized |
| peterson.6 | 31762156 |  | nomem | nomem | nomem | timeout | notserialized | notserialized | notserialized |
| peterson.7 | 29618148 |  | nomem | nomem | timeout | timeout | notserialized | notserialized | notserialized |
| pgm_protocol.1 | 10175 | 17673 | 0,133333 | 1,94333 | 0,98 | 4,51 | 4,98667 | 4,61333 | 2,94333 |
| pgm_protocol.2 | 17096 | 32486 | 0,176667 | 2,98 | 1,35333 | 6,87667 | 7,06333 | 6,92667 | 3,84333 |
| pgm_protocol.3 | 200453 | 386407 | 1,10667 | 30,9867 | 5,45333 | 68,2033 | 74,4567 | 68,9733 | 30,65 |
| pgm_protocol.4 | 39832 | 94166 | 0,316667 | 6,785 | 1,84667 | 14,3867 | 15,5533 | 14,73 | 7,39667 |
| pgm_protocol.5 | 382731 | 894800 | 1,92 | 58,9617 | 10,48 | 127,527 | 138,817 | 128,187 | 57,41 |
| pgm_protocol.6 | 2659550 | 8510552 | 10,9567 | 419,51 | nomem | 901,843 | 1012,86 | 904,087 | 390,183 |
| pgm_protocol.7 | 322585 | 831133 | 1,75 | 48,805 | 9,48667 | 109,687 | 118,347 | 106,757 | 48,7333 |
| pgm_protocol.8 | 3069390 | 7125121 | 10,02 | 474,412 | nomem | 1014,93 | 1166,66 | 1038,08 | 472,353 |
| phils.1 | 80 | 212 | 0,113333 | 0,403333 | 0,5 | 0,63 | 0,366667 | 0,44 | 0,353333 |
| phils.2 | 581 | 2350 | 0,0733333 | 0,0733333 | 0,746667 | 0,716667 | 0,573333 | 0,61 | 0,38 |
| phils.3 | 729 | 2916 | 0,0633333 | 0,075 | 0,513333 | 0,646667 | 0,566667 | 0,636667 | 0,356667 |
| phils.4 | 340789 | 3123558 | 1,89333 | 7,4 | 21,36 | 23,8167 | 44,5633 | 23,1967 | 7,93333 |
| phils.5 | 531440 | 4251516 | 2,61667 | 11,4483 | 29,6067 | 34,97 | 61,8433 | 34,3333 | 11,71 |
| phils.6 | 14348906 | 143489055 | 79,0767 | 426,357 | nomem | timeout | nomem | nomem | nomem |
| phils.7 | 19548203 |  | nomem | nomem | nomem | timeout | notserialized | notserialized | notserialized |
| phils.8 | 20328500 |  | nomem | nomem | nomem | timeout | notserialized | notserialized | notserialized |
| pouring.1 | 503 | 4481 | 0,106667 | 1,37444 | 0,88 | 3,46667 | 4,56333 | 4,44333 | 7,48667 |
| pouring.2 | 51624 | 1232712 | 0,59 | 22,6767 | 8,15667 | 64,2433 | 72,08 | 64,5967 | 27,1533 |
| production_cell.1 | 14586 | 39210 | 0,11 | 0,346667 | 1,13667 | 1,37333 | 1,80667 | 1,37333 | 0,786667 |
| production_cell.2 | 9003 | 21202 | 0,0866667 | 0,213333 | 0,866667 | 1,13 | 1,33 | 1,09333 | 0,583333 |
| production_cell.3 | 822612 | 2496342 | 1,83 | 15,0383 | 20,9567 | 37,8067 | 60,99 | 36,9533 | 15,79 |
| production_cell.4 | 340685 | 968176 | 0,956667 | 5,12333 | 8,46 | 13,69 | 22,2767 | 13,9333 | 5,65333 |
| production_cell.5 | 4211856 | 13072120 | 5,84 | 88,5017 | 108,517 | 222,813 | 337,58 | 220,04 | 85,2167 |
| production_cell.6 | 14520700 | 45593810 | 18,9567 | 366,92 | 375,653 | 917,517 | 1282,42 | 926,273 | 369,72 |
| protocols.1 | 2430 | 6480 | 0,0633333 | 0,1 | 0,535 | 0,77 | 0,66 | 0,68 | 0,38 |
| protocols.2 | 11286 | 42255 | 0,1 | 0,185 | 1,13667 | 1,05 | 1,47667 | 1,015 | 0,476667 |
| protocols.3 | 2817 | 7826 | 0,06 | 0,095 | 0,565 | 0,823333 | 0,763333 | 0,78 | 0,463333 |
| protocols.4 | 439245 | 1454834 | 1,11 | 3,68167 | 11,96 | 11,33 | 22,5467 | 10,51 | 4,17 |
| protocols.5 | 996345 | 3272786 | 1,81667 | 7,40833 | 25,7033 | 23,14 | 48,0533 | 21,59 | 7,93 |
| public_subscribe.1 | 580 | 867 | 0,0733333 | 0,426667 | 0,733333 | 0,826667 | 0,88 | 0,84 | 0,703333 |
| public_subscribe.2 | 1846603 | 6087556 | 3,5 | 87,5817 | 48,84 | 205,093 | 260,713 | 209,277 | 84,9267 |
| public_subscribe.3 | 1846603 | 6087556 | 3,77333 | 93,6933 | 50,78 | 228,123 | 270,573 | 223,055 | 90,5233 |
| public_subscribe.4 | 1846603 | 6087556 | 4,10333 | 102,755 | 49,8267 | 249,253 | 285,653 | 249,303 | 98,9933 |
| public_subscribe.5 | 16788621 |  | nomem | timeout | timeout | timeout | notserialized | notserialized | notserialized |
| reader_writer.1 | 2666 | 10658 | 0,07 | 0,148333 | 0,583333 | 0,93 | 1,03 | 0,93 | 0,51 |
| reader_writer.2 | 4104 | 49190 | 0,09 | 0,266667 | 0,98 | 1,35 | 1,56 | 1,37 | 0,71 |
| reader_writer.3 | 604498 | 4125562 | 2,57 | 28,225 | 29,78 | 83,62 | 107,623 | 83,9633 | 27,57 |
| resistance.1 | 8183469 | 32052024 | 10,57 | 98,1767 | 257,267 | 292,03 | 636,913 | 304,807 | 156,833 |
| resistance.2 | 51516701 | 293956164 | 100,687 | 846,643 | 2210,52 | 2519,24 | nomem | 2479,24 | 996,623 |
| rether.1 | 2458 | 2755 | 0,0766667 | 0,173333 | 0,523333 | 0,883333 | 0,94 | 0,926667 | 0,673333 |
| rether.2 | 9278 | 10329 | 0,0966667 | 0,446667 | 0,79 | 1,63 | 1,77333 | 1,66333 | 1,13333 |
| rether.3 | 305334 | 334516 | 0,896667 | 16,1167 | 5,07667 | 38,01 | 40,68 | 38,035 | 16,8067 |
| rether.4 | 1157052 | 1535386 | 2,18667 | 80,4317 | 19,3533 | 186,403 | 195,71 | 180,453 | 78,1433 |
| rether.5 | 3017044 | 3302351 | 3,69333 | 257,892 | 43,1233 | 588,113 | 591,737 | 557,817 | 254,593 |
| rether.6 | 5919694 | 7822384 | 6,25667 | 546,007 | 93,93 | 1243,87 | 1273,75 | 1242,24 | 529,983 |
| rether.7 | 4789409 | 5317199 | 5,27333 | 543,805 | 71,78 | 1273,27 | 1288,09 | 1225,18 | 515,667 |
| rushhour.1 | 1048 | 5446 | 0,0766667 | 0,113333 | 0,466667 | 0,993333 | 1,08 | nomem | 1,13 |
| rushhour.2 | 2242 | 12603 | 0,0933333 | 0,191667 | 0,606667 | 1,23 | 1,31667 | 1,34 | 1,64 |
| rushhour.3 | 156723 | 1583980 | 2,32333 | 16,7767 | 13,04 | 56,66 | 88,1433 | 66,24 | 121,71 |
| rushhour.4 | 327675 | 3390234 | 4,43667 | 35,735 | nomem | 116,843 | nomem | 146,7 | nomem |
| schedule_world.1 | 23061 | 143130 | 0,18 | 0,606667 | 1,89667 | 1,92667 | 3,04 | 2,00333 | 1,05333 |
| schedule_world.2 | 1570340 | 14308706 | 6,49333 | 37,505 | 94,3633 | 111,43 | 201,723 | 107,86 | 37,3067 |
| schedule_world.3 | 14420303 |  | nomem | timeout | timeout | timeout | notserialized | notserialized | notserialized |
| sokoban.1 | 91453 | 228313 | 0,483333 | 3,02 | 3 | 8,53333 | 11,8467 | 8,13333 | 19,5133 |
| sokoban.2 | 761633 | 2012841 | 2,46333 | 22,225 | 18,3533 | 67,01 | 100,283 | 60,0367 | 169,987 |
| sokoban.3 | 7034432 |  | nomem | nomem | nomem | nomem | notserialized | notserialized | notserialized |
| sorter.1 | 20544 | 30697 | 0,12 | 0,6 | 1,17667 | 1,91 | 2,21 | 1,96667 | 1,10333 |
| sorter.2 | 7592 | 10490 | 0,09 | 0,265 | 0,793333 | 1,29 | 1,42667 | 1,245 | 0,783333 |
| sorter.3 | 1288478 | 2740540 | 2,02 | 30,235 | 24,5733 | 75,54 | 102,67 | 75,2867 | 30,2233 |
| sorter.4 | 12958752 | 27051822 | 11,1267 | 296,643 | 234,097 | 771,275 | 1088,01 | 789,63 | 293,99 |
| sorter.5 | 296148 | 630246 | 0,88 | 7,46167 | 6,47667 | 20,0233 | 24,87 | 20,5867 | 8,07333 |
| synapse.1 | 46756 | 190843 | 0,246667 | 3,255 | 2,32333 | 8,6 | 9,64667 | 8,65333 | 5,09667 |
| synapse.2 | 61048 | 125334 | 0,27 | 4,30333 | 2,09 | 11,9467 | 11,83 | 11,96 | 6,51333 |
| synapse.3 | 390317 | 826864 | 1,03333 | 29,3733 | 8,52 | 69,87 | 79,2067 | 68,12 | 36,42 |
| synapse.4 | 2292286 | 4921830 | 2,64333 | 184,067 | 45,6867 | 439,707 | 478,4 | 430,243 | 214,743 |
| synapse.5 | 83263 | 189639 | 0,36 | 7,235 | 2,49 | 18,9667 | 19,33 | 18,895 | 9,51667 |
| synapse.6 | 625175 | 1190486 | 1,47333 | 58,4317 | 12,5433 | 133,48 | 140,557 | 134,98 | 61,13 |
| synapse.7 | 6465201 | 19893297 | 9,41667 | 1091,79 | 187,843 | 2495,38 | 2643,96 | 2419,7 | 1138,47 |
| szymanski.1 | 20264 | 56701 | 0,14 | 0,798333 | 1,44333 | 2,37333 | 2,79333 | 2,42 | 1,44333 |
| szymanski.2 | 31875 | 88521 | 0,176667 | 1,16333 | 1,54 | 3,18 | 3,81667 | 3,10667 | 1,84 |
| szymanski.3 | 1128424 | 4234041 | 2,25333 | 39,2033 | 31,3767 | 97,1967 | 125,74 | 95,8133 | 39,12 |
| szymanski.4 | 2313863 | 8550392 | 3,59 | 77,9017 | 63,3667 | 199,353 | 257,33 | 190,91 | 76,1 |
| szymanski.5 | 29555064 | 375297913 | 161,75 | timeout | 2700,83 | timeout | notserialized | notserialized | notserialized |
| telephony.1 | 1280 | 3497 | 0,0766667 | 0,115 | 0,446667 | 0,806667 | 0,89 | 0,84 | 0,7 |
| telephony.2 | 51826 | 200322 | 0,25 | 1,895 | 2,32333 | 5,14333 | 6,44667 | 5,25 | 3,16333 |
| telephony.3 | 765379 | 3155026 | 1,95333 | 28,11 | 24,0267 | 70,9533 | 96,7833 | 69,86 | 31,6533 |
| telephony.4 | 12291552 | 64110312 | 22,59 | 533,93 | 426,09 | 1232 | 1823,31 | 1227,3 | 544,33 |
| telephony.5 | 21128662 |  | nomem | timeout | timeout | timeout | notserialized | notserialized | notserialized |
| telephony.6 | 23825732 |  | nomem | timeout | timeout | timeout | notserialized | notserialized | notserialized |
| telephony.7 | 21960308 | 114070470 | 48,1033 | 1069,74 | 794,247 | 2566,94 | nomem | 2574,93 | 1085,59 |
| telephony.8 | 22380334 |  | nomem | timeout | timeout | timeout | notserialized | notserialized | notserialized |