



Západočeská univerzita v Plzni
Fakulta aplikovaných věd

Autoreferát disertační práce

Plzeň 2005

Ing. Jaroslav Kačer

University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Simulation-Based Checking of Java Concurrent Programs

Ph.D. Thesis Abstract

Ing. Jaroslav Kačer

Supervisor: Doc. Ing. Stanislav Racek, CSc.
University of West Bohemia, Pilsen, Czech Republic
Faculty of Applied Sciences
Department of Computer Science and Engineering

Pilsen, January 2005

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Simulation-Based Checking of Java Concurrent Programs

Autoreferát disertační práce

Ing. Jaroslav Kačer

Školitel: Doc. Ing. Stanislav Raček, CSc.
Západočeská univerzita, Plzeň, Česká republika
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Plzeň, leden 2005

Tato disertační práce byla vypracována v prezenčním doktorském studiu na Fakultě aplikovaných věd Západočeské univerzity v Plzni v letech 2001-2005.

Uchazež: Ing. Jaroslav Kačer

Školitel: Doc. Ing. Stanislav Racek, CSc.
Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky
Univerzitní 22, Plzeň, 30614

Školící pracoviště: Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky
Univerzitní 22, Plzeň, 30614

Oponenti: Doc. Ing. Jan Janeček, CSc.
České vysoké učení technické
Fakulta elektrotechnická
Katedra počítačů
Karlovo náměstí 13, 12135 Praha 2

Prof. Ing. Ivo Vondrák, CSc.
Vysoká škola báňská – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky
Tř. 17. listopadu 15, 70833 Ostrava-Poruba

Autoreferát byl rozeslán dne

Obhajoba disertační práce se koná dne před komisí v oboru Informatika a výpočetní technika na Fakultě aplikovaných věd v Plzni, Univerzitní 22, v místnosti v hodin.

S disertační prací je možno se seznámit na oddělení vědecké výchovy ZČU, Univerzitní 8, místnost UR 407/408.

Doc. Ing. Karel Ježek, CSc.
předseda oborové rady

Contents

1	Introduction	3
1.1	Motivation	3
1.2	Aims of the Dissertation Work	4
2	Overview of Key Results	6
2.1	State-Space Model of Concurrent Java Programs	6
2.2	Applying the State-Space Model to Real Java Source Code	12
2.3	Reactive Java Programs & The State-Space Model	14
2.4	Model-Based Development of Java Embedded Reactive Programs	17
2.5	Support Tools	19
2.6	Feasibility Study	20
3	Summary	23
3.1	Contributions of the Work	23
3.2	Evaluation of Thesis Goals	24
3.3	Future Work	24
	List of Published Articles	26
	References	28
	Original Abstract of the Ph.D. Thesis	32
	Původní abstrakt disertace	33

1 Introduction

Testing is a necessary part of the life cycle of any software product. There are many techniques developed in software engineering, dealing with the issue of testing. Usually, they have one thing in common: they are tailored exclusively for purely sequential programs. This thesis attempts to provide a method of program checking that would be able to reveal behavioral errors in Java *concurrent and reactive* programs.

1.1 Motivation

Although the Java language provides all necessary means for multithreading already in the basic package and the language was designed to support multithreading from the very beginning, it cannot protect the programmer from all possible traps that multithreading threatens with.

Java threads are believed to run in parallel although they usually run in a pseudo-parallel manner because of a limited amount of processor resources. The virtual machine is allowed to switch threads at arbitrary places of their code which causes an unpredictable sequence of program operations during every run.

To protect shared resources and to allow for monitor methods guarding, Java introduces methods `wait()`, `notify()`, and `notifyAll()`, able to manipulate thread execution state. Using these methods is quite often a source of incorrect program behavior which can lead to liveness failures or turn objects to wrong states.

In every correct concurrent program, two properties must be fulfilled:

The Safety Property. The safety property ensures that all data shared by more threads are accessed in a way that avoids data damages and inconsistencies. This is usually accomplished by a synchronization protocol.

The Liveness Property. The liveness property ensures that anything at all will happen in the program. An example of liveness violation is a *deadlock*, *dormancy*, or *contention*.

It is obvious that the two properties go against each other. Ensuring safety may cause a liveness violation, ensuring liveness may lead to missing synchronization and therefore to safety violation. It is sometimes very difficult to balance the two opposite concerns so that the result is a viable and correct program.

Java introduces so-called **synchronized** blocks to ensure safety, i.e. to protect data from being accessed or modified from different threads simultaneously. However, the programmer is not forced to use synchronization and

therefore, as a result of that, incorrect data can be easily produced. These incorrectly synchronized reads or writes are called *safety violations*.

If synchronization is over-used or used in a wrong way, the following troubles can arise, some of which are often denoted as *liveness violations*:

- a deadlock can happen;
- the state of dormancy can happen;
- a livelock can happen;
- incorrect guarding condition may lead to a wrong update of an object's state;
- an `IllegalMonitorStateException` can be thrown out as a result of incorrect usage of methods `wait()`, `notify()`, and `notifyAll()`.

Therefore, observing the state of every thread and knowing about every state change of a thread may help the programmer to discover bugs related to liveness. The programmer is then able to compare the actual behavior of the program to the intended one and to find the buggy piece of code.

Every program has its *mission* that it should fulfill. It is a difficult task to tell whether the program does or does not fulfill the mission since one must know what should be checked, how it should be checked and how to interpret the result.

One of possible ways is to let the user define whether the program acts correctly or not by means of *state invariants*. An invariant is a statement describing a certain condition about the tested program or any outer object (a controlled device, for example) that must hold at any time. Evaluation of this statement returns a logical value. In the case of a correct program, evaluation of every invariant should always return `true`.

A little disadvantage of invariants is that a broken invariant does not point to any place in the source code that could be the cause of the problem. The cause can be virtually anything and it is up to the programmer to find it and take appropriate steps.

A concurrent environment poses a requirement on invariant evaluation: An invariant should be evaluated atomically, i.e. the data from which the invariant is computed must not change during evaluation.

1.2 Aims of the Dissertation Work

With all presented drawbacks and potential problems in mind, let's now formulate some fundamental goals that this work should fulfill:

1. A general description/analysis of concurrent Java program behavior should be presented. The analysis should identify various entities and describe their behavior during a program execution, mainly during threading-specific operations.
2. Based on the analysis, a method of reactive Java concurrent program checking should be developed. The method should target mainly:
 - the ability to execute a concurrent Java program in a managed/simulated way so that all entities identified in point 1 and operations on them are properly simulated;
 - the ability to analyze the tested program at runtime in such a way that is not affected by Java multithreading, i.e. the ability to view the program from a “global point of view” in a “consistent state” in the same way a classic sequential program can be seen;
 - the ability to discover a faulty behavior – by “faulty behavior” we will mean either multithreading-related incorrectness in the program or a situation when the tested program is not able to fulfill its task, e.g. when a controlled device gets to a wrong state;
 - the ability to test a reactive program within a close-to-real environment that supplies events and data to it – the environment should provide program input data that are expected in the target environment and it should reflect program output data by changing its state appropriately.
3. Tools fulfilling the aforementioned requirements should be developed to demonstrate that the presented theory and solutions based on it are viable.
4. The method and the tools should be demonstrated on a representative example.

2 Overview of Key Results

This chapter summarizes main ideas of the thesis and presents results achieved so far. It also describes some programming work that has been done in order to prove viability of the presented theoretical concepts.

2.1 State-Space Model of Concurrent Java Programs

The Java language supports multithreading from its nature, therefore all Java programs use the same language elements and patterns. There are no additional libraries, no API functions of an operating system, just an exactly defined set of keywords, standard classes and their methods.

This allows us to define a model of concurrent Java programs, analyze source code of any program and observe how the program behaves when mapped to the model. The model itself and its practical usage for purposes of program testing and run-time analysis are described in the following sections.

2.1.1 Basic Entities of a Concurrent Java Program

If a typical Java concurrent program is analyzed, the following basic types of entities can be found:

Threads. A dynamic set of runnable objects. Every thread has its own local data set, e.g. thread attributes, thread methods' local variables, own stack, etc.

Monitors. A dynamic set of objects holding global data that threads use for mutual communication. Methods manipulating with a monitor's data must necessarily be synchronized. Threads keep references to monitors whose data have to be shared among them.

Two types of operations can be distinguished in a thread's code:

Local Computing. A sequence of operations not manipulating with global data, only with the thread's local data.

A Call to a Monitor Method. A sequence of operations manipulating both with a monitor's (i.e. global) data and the thread's local data. However, the modification of the calling thread's local data has no effect on our model. A possible result returned from the method can be stored to a local variable. This category also includes synchronized blocks that are part of an unsynchronized method or even a synchronized block inside another synchronized block.

To explain the state-space model used as a conceptual framework for the presented method of Java program testing by simulation, the following set of assumptions will be used:

- The set of monitors used for computation is static, the cardinality of the set is m , no monitor service uses another monitor's services (a monitor cannot reference another monitor).
- The set of threads used for computation is static as well, the dimension of the set is n .
- Monitors are “passive”, i.e. they are not implemented using “internal” threads.
- All monitor methods are synchronized.

2.1.2 The State Space

A **monitor** is in a **consistent state** when no thread is currently executing its code, that is:

- when no thread is currently running inside its methods, and
- when zero or more threads are suspended inside one of its methods using `wait()`, when the monitor's lock is free.

Similarly to monitor consistent states, thread consistent states can be defined.

A **thread** is in a **consistent state** when:

- it has been born, but its `start()` method has not been invoked yet, or
- it reaches the beginning of a synchronized block of code, or
- it reaches the end of a synchronized block of code, or
- it becomes non-runnable after invoking `wait()`, `sleep()`, etc., or
- it has died already but its data are still valid.

Uniform Naming and Indexing Conventions: In the rest of the chapter, the following conventions will be used:

n	...	number of threads
m	...	number of monitors
i	...	thread index, $i = 1..n$
t_i	...	i^{th} thread
C_i	...	number of consistent states of i^{th} thread
c_{ij}	...	j^{th} consistent state of i^{th} thread, $j = 1..C_i$
$P_{i,c_{ij}}$...	number of possible transitions of i^{th} thread being in consistent state c_{ij}
n_r	...	number of runnable threads, $n_r \leq n$
$S_{\alpha\beta\dots\omega}$...	global consistent state composed of states $\alpha = c_{1a}, \beta = c_{2b}, \dots, \omega = c_{nz}$

Now let us proceed to a description of a discrete state space of a concurrent Java program computation, also called the *global state space*:

- Every thread has its own *coordinate axis within the space*, so the state-space dimension equals to n .
- Every discrete state-space coordinate axis has a limited number of points (nodes), that correspond to the thread consistent state identifiers *thread.state_pos*.
- State space then can be viewed as a bounded regular grid of nodes within an n -dimensional space.
- *State of the computation* can be defined as a point (node) within the grid, i.e. as an ordered set of single thread state identifiers.
- A *transition* is a thread activity between two consequent consistent states. During a transition, a thread can execute either its own “local” code or monitor code. Due to the fact that a transition means a single thread activity, only one state-space coordinate value changes with this transition.
- *Control flow of the computation* can be modeled as an oriented graph whose nodes are all the points of the grid. From every node

$$\sum_{i=1}^n P_{i,c_{ij}}$$

possible transitions lead to other nodes.

- *Path of computation* is a sequence of transitions that an instance of computation passes through the control-flow graph of computation.

For a concurrent Java program executed by the JVM, the following behavior of computation can be observed:

- The program can hardly reach a consistent state as defined above. If a snapshot of the program's execution had been taken at an arbitrary point, it would not have probably resided in any node of the state-space grid because the coordinate values of all axis would correspond neither to thread nor to monitor consistent states.
- So in every state of concurrent computation several transitions “have fired” and it is generally impossible to predict which state of computation will follow. Moreover it gives no great sense to stop the computation in a state that is not consistent.
- As a known consequence follows that the path of computation is quite nondeterministic. It causes majority of known troubles with concurrent programs testing and debugging.

2.1.3 Serialization of Concurrent Java Programs

*Serialization of a concurrent Java program is a method of execution that executes only one thread at a given time, stops execution in every consistent state, and allows arbitrary selection of the next thread to run in every such state while respecting all rules of original Java program scheduling.*¹

The main idea is to choose, in a defined manner, for every node of the above defined state space of concurrent computation only one thread from the set of runnable threads to perform its transition upon request. The method of serialization is able to stop a concurrent computation in every consistent state and to let the program be examined without affecting the computation in any way.

The resulting path of computation will be *one of possible computation paths, therefore it will reflect a possible real execution of the program*. The basic idea of the proof of this claim is stated in [3].

An implementation of the serialization idea should bring the following important consequences:

- The computation can be stopped in every node of its state-space and a runnable thread to perform a transition can be chosen.
- When the computation has stopped in a node, the corresponding state of computation is consistent, i.e. all monitors and all threads reside within their consistent states, which means that these states can be observed and analyzed.

¹mainly thread states, priorities, and locking discipline

- The path of computation can be made deterministic, i.e. every possible path through the control-flow graph of computation can be chosen either automatically, i.e. according a rule, or manually.
- A possibility how to construct the computation path automatically is to use discrete model time concept for transition ordering.
- The properties of reactive systems assure that the state-space graph can be explored repeatedly. Therefore, not just one straight path through the graph, from the initial consistent state to the terminal one, but a possibly infinitely long path will be taken during execution. This should assure a reasonable coverage of the state space because of random (or pseudo-random) selection of the next thread in every node of the state-space grid. The larger (more complete) the coverage, the bigger the chance that all possible paths of program execution have been tested.

2.1.4 Walking through the State Space

In this section, it will be shown why a global consistent state does not have exactly n neighboring states, as one would think. Actually, if the final consistent states symbolizing that a thread is dead are omitted, the number of neighboring states can vary from n in the simplest case to

$$\sum_{i=1}^n (C_i - 1)$$

in the most complicated case. C_i is decremented by one because no thread can return to its initial state which corresponds to the thread's creation. Therefore, one can state that the following expression

$$n \leq \sum_{i=1}^n P_{i,c_{ij}} \leq \sum_{i=1}^n (C_i - 1)$$

holds in any of the global consistent states except for all global states composed from at least one final state when at least one thread is dead already.

Just note that the above expressions do not take into account thread states. In other words, these expressions describe static properties of the program. The actual number of possible consistent states that can be reached from the current consistent state will vary from one to

$$\sum_{i=1}^{n_r} P_{i,c_{ij}}$$

where n_r is the current number of *runnable* threads. Also note that C was replaced by P since P describes the actual number of possible subsequent consistent states of a thread being in one of its consistent states while C simply denotes all consistent states of a thread.

And it is obvious that

$$\sum_{i=1}^{n_r} P_{i,c_{ij}} \leq \sum_{i=1}^n P_{i,c_{ij}}$$

just because n_r is always less or equal to n and $P_{i,c_{ij}}$ is a constant for a given i and its given thread state c_{ij} because it is a static property that can be determined even from the source code.

It can be found from the above statements that the definition of the term “neighbor state” is not a trivial task. It will be defined as follows:

*Let t_1 be a thread in a consistent state c_{1x} . A consistent state c_{1y} of the same thread t_1 is a **neighbor state** of the consistent state c_{1x} if such an execution of the thread t_1 's code exists which leads from c_{1x} to c_{1y} and this execution does not contain any t_1 's consistent state c_{1z} , $c_{1z} \neq c_{1y}$.*

As for the global consistent states, the definition of the neighborhood relation may look like this:

*Let S_1 be a global consistent state. Then S_2 is a **neighbor** of S_1 if there exists a thread t such that the sub-state of S_2 with regard to t is a neighbor of the sub-state of S_1 with regard to t in t 's code.*

2.1.5 Utilization of the State-Space Model

In general, construction of a program's state space is quite a difficult task. First, all consistent states of all threads must be enumerated. Then, an n -dimensional cuboid must be constructed from individual axes. In real programs, the dimension of the cuboid changes dynamically during program execution because new threads are created. And it is almost impossible to decide whether a global consistent state is reachable or not because a very detailed analysis of each thread's source code would be necessary. Moreover, the number of global consistent states grows exponentially in the number of threads.

The proposed method, described later, avoids these and some other well known state-space-explosion problems. It does not require the model to be constructed, neither before the testing starts nor during the testing. The following rules hold during execution:

- All threads t_i are registered. Their number n and the number of runnable threads n_r are known.
- The number of consistent states of an i^{th} thread C_i is known. It can be determined from program's source code static analysis.

- When the program is in a global consistent state $S_{\alpha\beta\dots\omega}$, the next global consistent state is determined only by the next thread selected to run. It means that only i , a thread number, will be selected automatically or manually, not a specific transition. One of $P_{i,c_{ij}}$ possible transitions will be selected and performed automatically because the transition selection, and therefore the next consistent state of the i^{th} thread and therefore the next global consistent state, depend on the selected thread's attribute values and input data.

Serialization allows to reach an arbitrary consistent state of a program if the decision to reach it is made at the beginning of program execution. This may not longer be possible if a particular state has been reached from which a path does not lead to the desired state.

Whenever a global consistent state is reached, program execution can be “frozen”. This allows the program to be analyzed without affecting it. All data values reported at this point are mutually consistent because either none or all changes of monitor data have been applied.

2.2 Applying the State-Space Model to Real Java Source Code

The last section explained the main principles of the state-space model of Java concurrent programs, its construction from a set of consistent states, and the main ideas of serialization of Java multithreaded programs.

Such abstract concepts may look to be quite far from the reality of actual source code. This section aims to prove that this feeling is not true. The chapter describes all multithreading-related concepts known from Java Virtual Machine Specification [28] and shows how they can be incorporated into the state-space model. Moreover, the chapter enumerates all important places in Java source code that normally influence program execution and therefore must be taken into account when a serialized program is executed.

2.2.1 JVM-like Entities and Their Relationships

When looking at a Java multithreaded program from a point of view that is closer rather to the level of the Java virtual machine than to the high-abstraction level of Java source code, it can be found that some entities that are not used or mentioned in the source code at all should also be taken into account. These entities are as follows:

Threads. A thread is an instance of the class `Thread` or its subclass. A thread has a state. A thread can be associated with one or more locks that it is currently holding.

Locks. Any object that is used for synchronization in the source code. Two operations are provided by locks that threads can use: `lock()` and

`unlock()`. Every lock should also provide a method replacing the behavior of original Java methods `wait()`, `notify()`, and `notifyAll()`.

Wait Set. A set of threads associated with every lock. It stores threads that have invoked the `wait()` method of the lock and can be resumed later.

Delayed Set. A set of threads associated with every lock. It stores threads that have tried to enter a region of code synchronized with its respective lock when the lock was held by another thread.

Join Set. A set of threads associated with every thread. Threads in the set are temporarily suspended, until the owner of the given join set dies.

2.2.2 Entry Points to the JVM

Now when it should be understood what the model of a concurrent Java program should deal with, points in Java source code where the program interacts with the JVM scheduling mechanism can be presented. In order to replace this default JVM mechanism with a serialized threading mechanism, it is necessary to:

1. properly detect all such places in the program's source code;
2. replace all found pieces of source code with a special-purpose library code that implements the serialization method;
3. re-implement all functionality of the original JVM scheduler in the aforementioned library to allow to monitor and control the program's behavior, but without negatively affecting the program with inappropriate changes of behavior that are not normally possible.

The interesting places in source code are as follows: beginnings and ends of synchronized blocks, calls to `wait()`, calls to `notify()` and `notifyAll()`, calls to `destroy()`, priority changes, calls to `interrupt()`, `interrupted()` and `isInterrupted()`, calls to `join()`, thread suspensions and resumes, calls to `stop()`, calls to `setDaemon()`, sleeps, calls to `yield()`, thread creations, startups and deaths.

2.2.3 Performing Behavior Changes

The first two tasks mentioned at the beginning of section 2.2.2 can be performed at the level of program source code by simple text substitutions. The third task requires a library to be written that replaces a part of JVM functionality in a manner that complies with the serialization principles.

As for the first two tasks, a tool able to perform such text substitutions in Java source code exists: J-SourceMorph. As for the last task, the required library implementing serialization principles has also been developed. It is a part of the J-Sim simulation library. The serialization library is inserted between the program and the JVM. Some method calls are diverted to the library which in turn internally uses (or is supposed to use) standard JVM services. Redirection of the selected method calls is achieved by proper modification of the checked program's source code. The rest of program functionality remains unchanged and uses standard JVM services as usual.

2.3 Reactive Java Programs & The State-Space Model

This section presents a concrete method of utilization of the serialization idea that is applied in the area of so-called reactive programs. In order to be usable for checking of reactive systems, the original method must be extended with addition concepts, also introduced in this section.

2.3.1 Reactive Systems

Reactive systems are systems that continuously interact with their environment and react upon events or data obtained from it. They usually consist of a set of periodic tasks that run in a neverending loop. Data necessary for program control are obtained on-the-fly when they are needed. Similarly, output data are also produced continuously as the program reacts and attempts to influence its environment in a way corresponding to its preprogrammed behavior.

It is assumed that reactive systems implemented in Java will typically consist of several threads executing in parallel within one Java application, i.e. sharing common memory space. If the state-space model is applied to reactive Java programs, some interesting observations can be made:

- Because all threads have periodic behavior and their state spaces are thus cyclic, the global state-space graph of the program is also cyclic.
- As a consequence, the computation path of the program is infinite and cyclic. Therefore, there exists a set R_S of global consistent states that can be repeatedly visited during the program's serialized execution, no matter in which of the states the program resides now. A set R_T of corresponding possible transitions exists for this set of global consistent states.
- Under standard conditions, the program will hardly pass through all possible states from R_S , executing all transitions from R_T . Rather, there will be an N-dimensional "strip" consisting of some of the states from R_S and their corresponding transitions from R_T . Program execution will stay within this strip for most time under standard conditions and the rest will remain unexplored.

Serialization of threads allows to select an arbitrary computation path that can theoretically be used during execution if all principles of Java scheduling are respected. However, to make testing of reactive systems more automatic, there should be a way of automatic path construction. Random or pseudo-random or manual selection does not actually lead to a realistic computation path. A solution to this issue is presented in the next section.

2.3.2 Extending the State-Space Concept for Reactive Systems

It is therefore necessary to extend the original serialization method in such a way that would:

- construct a realistic dynamic computation path that would correspond to real operation mode;
- allow this path to be intentionally “broken” or “disturbed” in order to deviate the program flow from its usual path.

This can be accomplished using the following principles:

- Instead of the real world, the program will communicate with its *model*, responsible for providing reasonable input data and accepting output data that should influence the model in the same way they influence the real world.
- The model will be composed of active entities because model data must be updated not only by the program but also by “their own will”. This simulates real-world processes that are completely independent on the program.
- The active model entities will execute using the concept *discrete model time* known from the Simula language. Execution of all active model entities will be ordered by the value of model time and interleaved in a similar way that serialization interleaves threads of the program.
- The concept of *discrete model time* will be applied to the serialized program as well. This will cause that every transition will be executed at one discrete value of model time. Model time will be advanced after every transition by a certain value. This value can (but need not necessarily) be derived from the real time spent by a transition.
- Now when both the program and the model of the real world executes using the discrete model time, their executions can be merged together and interleaved using a common value of model time. This can be accomplished by a shared calendar.

- The two parts can be connected via a model of the interface between the computer/program and the rest of the world. The model of the interface will assure that data produced by the program will be delivered to the model of the real world and, conversely, that data read by the program will be taken from the model instead of the real world itself.

2.3.3 Model of the Environment

The proposed method assumes that the model of the surrounding real world is constructed on top of J-Sim using conventional methods of discrete-time process-oriented simulation. Basically, the model will be a set of objects, some of them are discrete-time simulation processes periodically updating certain data. A process can periodically read a value from a data structure that serves as the model of the computer-environment interface, to recompute state of the model using the read value and (optionally) to write a result of computation into the object model of the control interface.

2.3.4 Model of the Control Program's Activity

As mentioned before, not the original application program, but its model, produced by serialization combined with introduction of model time, is checked by the presented method. The following list defines terms necessary to understand the process of the control program's model creation:

Control Program. The original program. The static view CP_S of the control program is a constant set of classes, either threads or monitors, i.e. "passive" objects with synchronized methods.

Process of Control Program Execution. It is the dynamic view CP_D of the control program, i.e. a set of instances of threads that are interacting using a set of monitors.

Process of a Thread Execution. It is an independent subprocess $CP_D T_i$ of the CP_D execution. It is described by the `run()` method that belongs to every thread class. The process of Java thread execution can be (roughly) in a state from the following set: running, delayed, not-runnable.

I/O Behavior. I/O behavior of the control program can be defined as a sequence (possibly infinite) of events on the control interface caused either by the control program activity or by the controlled environment.

Now let us start to construct a model CP_D^* of the control program activity CP_D . At the first – we need to pass from the continuous real time of

“physical” computation to the discrete model time of simulated computation. At the second – the data and functionality of every atomic part of computation between two consequent consistent states should be the same in both cases CP_D and CP_D^* . At the third – an atomic part of real-time computation should be performed at one point of the discrete model time and the computational delay of the atomic part of computation should be taken into account.

Then the CP_D^* computation can be performed as a sequence of atomic parts of all threads computations, where:

1. The order of atomic parts is (pseudo)randomly chosen.
2. The simulation time is advanced after every atomic part execution by the real time spent by the atomic part execution or by a value derived from it.
3. Using common model time, the CP_D^* activity can be properly interleaved with the EP_D^* activity, i.e. with the discrete-time model of the control program environment’s activity.
4. Moreover both activities can influence each other using the common model of the control interface.

When it is assumed that the EP_D^* , i.e. the modeled environment behavior, can be made – by proper EP_S^* construction – arbitrarily close to the real controlled environment behavior, then the overall model behavior, i.e. $CP_D^* + EP_D^*$, measured by both the input/output and state space behavior, can be made – by proper CP_S^* and EP_S^* construction – arbitrarily close to the real system, i.e. computer + CP_D + environment behavior.

2.4 Model-Based Development of Java Embedded Reactive Programs

This section describes how checking of an embedded reactive Java program should be incorporated into its development process and what should be accomplished in order to gain maximum profit from the presented checking method while keeping extra time expenses low.

The area of embedded programs was selected because it best shows the benefits of the presented method over some classic approaches thanks to limited possibilities of embedded program testing and debugging. Another reason is that reactive programs are in most cases used as control programs and control programs, in turn, are often put into embedded systems.

2.4.1 The Proposed Development Process

The main differences between a normal program development and the presented method are the following:

- A model of the controlled device and the environment has to be created, built up on top of J-Sim.
- The control program source code has to be modified slightly to pass from CP_S to CP_S^* .
- The two source code parts must be merged together into one J-Sim simulation application which means that they have common planning of simulation processes.
- The simulation application, i.e. the modified checked program and the program of the model of the environment, can be executed and results are obtained.
- At the end of the process, the checked-off control program source code can be separated from the model and straightforwardly converted to the production form. Only the control interface object should be implemented other way in order to use real hardware instead of the model data items.

2.4.2 Running Tests

The verification procedure is strongly application dependent, so only some general recommendations can be given here. Generally the procedure is similar as when testing the control program within its real-world environment, i.e. a well chosen set of activity scenarios is developed and then executed.

Let us assume a non-stop activity of the overall system. Then the model of the environment should issue a very long stream of pseudo-random events that the control program should cope with. The model-based execution gives us the possibility of arbitrarily detailed observation of the execution process without a “probe effect”. The execution can be deterministically repeated as well, even when a random sequence of events is used to stimulate the control program activity. There are two basic possibilities what to observe/inspect during a model-based test execution:

- *State invariants*, i.e. conditions that should apply all the time of execution, or – more generally – within a bounded part of the model execution. These invariants can be constructed using variables (object states) both from the control program and from a part of the environment model.
- *Behavioral protocols*, i.e. rules that should apply for a sequence of events, possibly including their timing, at the control interface.

If an incorrectness is found in a consistent state, the execution can be stopped and an analysis can be done.

The problem of the number and length of performed tests (diagnostic coverage, completeness) stays open – similarly as when the program is tested at a real device. But here, the possibilities of the tests organization are much extended, e.g. the possibility to use a “wild” environment activity or to change randomly the control program timing in order to reveal an incorrectness. The possibility of the model activity observation and investigation is better here as well, which means that the number and the coverage of the tests performed can be extended in this case.

The presented method seems to be a convenient complement to model checking that will very likely precede its usage in the development process. Its utilization should discover major mistakes or shortcomings in the program’s design. Some other tools could also be used in parallel to the presented checking method. For example, the Eraser tool [39] should be used to ensure proper locking discipline required by the method.

2.5 Support Tools

This section presents main software products aimed at fulfilling the requirements stated in section 1.2, namely the J-Sim simulation library, its subpackage JiJ, and J-SourceMorph, a converter of Java source code.

All three tools have been developed at the Department of Computer Science and Engineering of the University of West Bohemia in last four years and they are continuously updated. All of them are open source and available for free.

The first version of J-Sim was developed in 2001 as a master’s thesis project [16]. It has been updated many times since then, the last update is from fall 2004. It can be found at <http://www.j-sim.zcu.cz> and downloaded for free. The tool is used mainly for student projects in the course “Performance and Reliability of Computer Systems” to model and simulate behavior of parallel systems in discrete time. It is also the basic building block of JUTS, a Java simulator of urban traffic. The reader can find some useful information about using J-Sim for discrete-time simulation and simulation of queueing networks in [17], [19], [37], or directly in J-Sim documentation.

The JiJ subpackage of J-Sim was created in 2004 especially for the purpose of practical usage of the presented method of checking. Some initial ideas on JiJ appeared in [20], [18], and [22], later in [23]. JiJ was first intended to be a standalone library/simulator but it was later completely rewritten to become a part of J-Sim. The current state, including all technical details, is described in [24] or briefly in [25] or [21].

Created in 2004, J-SourceMorph is a very new piece of software. Like J-Sim, it was written as a part of a master’s thesis project [12]. It can be found at <http://www.j-sourcemorph.zcu.cz> and downloaded for free. Originally, it was intended to be used only for conversion of concurrent programs during the checking process, however, many other possible applications reveal in mind.

All the tools are described in detail in the thesis and in their respective documentation which is also available with the tools.

2.6 Feasibility Study

To demonstrate the presented method, a case study has been developed, simple enough to be used for demonstration purposes but, on the other side, not too trivial. It is an abstract embedded application that controls water level of a water station tank and several connected water sources.

There are N water sources with a pump and pipes connected to a main water tank. There are two sensors in each source: the first sensor signalling the water level being too low (the pump must be switched off) and the second one signalling the water level being high enough (the inactive pump can be switched on). Similarly, the tank is equipped with two sensors: the first one signaling low level (as many pumps as possible should be switched on) and the second one signalling high level (all running pumps should be switched off).

The aim of the control application is to keep the tank's water level between the limits, if possible, i.e. if the capacity of sources is sufficient. The application is able to switch on/off the pump placed in each source and to get its current state. There is one more sensor on the pipe leading from the main tank that reports the current output flow. As a limitation, only K pumps ($K \leq N$) can run at once due to assumed insufficient power supply.

The output flow and the quantity of water coming to each source are random processes generated in the environment model part. The sources have a certain natural level limit that can never be surpassed. This level is above the "high level limit" where a sensor is placed.

The complete feasibility study is available on-line at J-Sim home page as a part of its distribution. Source code and byte code is placed under directory `CaseStudies/1.WaterSystem`. The simulation is also packed into a JAR file that can be executed directly from the `JARs` directory using the following command:

```
java -jar JSim-0.5.0-CS1-WaterSystem.jar <parameters>
```

The simulation model program code, package `watersystem`, contains three principal parts that are described below:

- the abstract control interface, implemented by a class of the environment model;
- the control program (its simulation version);
- the model of the environment.

2.6.1 Abstract Control Interface

It's a Java interface that binds the submodel of control program and two submodels of its environment together. It is contained within the package `watersystem`.

The Java interface `CommonControlInterface` contains methods that allow reading `boolean` values from the sensors and switching the pumps on/off. The interface is implemented within the submodel of the control program environment in class `ModelInterface`. In the case of a real-world control program, this interface needs to be implemented using JNI functions to communicate with the real hardware, i.e. in the case of a real control program the interface implementation is a part of its code.

2.6.2 Model of the Control Program

The control program model code (package `controlprogram`) contains the following components, i.e. classes:

- For each source, there is a monitor object that is able to report a consistent state of all its sensors. It is also able to switch the pump on/off.
- For each source, there is a control thread. It is an endless loop consisting of periodic sensor data reading, evaluation, and possible reaction. When the level gets over/below the high/low limit, the pump is switched on/off by the control thread. During every loop, the control thread sleeps for a fixed time interval.
- For the main water tank, there is a monitor object that is able to report a consistent state of all its sensors. It also provides two “semaphore functions P and V”. If K pumps are already running or the tank is already full, the monitor suspends the calling control thread of a source willing to switch its pump on. The control thread can be later resumed when another control thread switches its pump off or by the station coordinating thread.
- For the main water tank, there is a periodic coordinating thread that reads the high-level sensor data. Whenever it finds that the level in the tank has dropped below the high-level limit, it unblocks control threads that may be blocked inside the station monitor.
- Class `WaterStationControl` encapsulates the whole control program part of the simulation.

2.6.3 Model of the Control Program Environment

The control program environment model part is divided into three subparts. The first part, package `iohw`, contains a model of HW components

that are parts of the embedded device and that the control program communicates with. The second, package `environment`, part contains a model of processes that the control system interacts with. The third part – class `ModelInterface` – encapsulates all the control program environment part and implements `CommonControlInterface`.

2.6.4 Invariants

In every simulation step, validity of several invariants is verified – see function `checkInvariants()`, class `ModelInterface`:

- Invariant no. 1 – The number of running pumps should not exceed the value K .
- Invariant no. 2 – If the tank's volume is at the high level, no pump should be running.
- Invariant no. 3 – If any source is below low level, its pump must be switched off.
- Invariant no. 4 – If the tank's volume is below the low level and number of sources at the high level is higher than the limit K , then the number of running pumps should be equal to K .

Note: In fact, during the model run, an invariant can be invalid for a short time interval. The control program will react in a future step – not necessarily the next one – as soon as it reads respective sensor values. The reaction itself, after sensor data are read, may take several simulation steps because it usually involves invocation of a synchronized method.

Measured in simulation time, state invariants of the controlled objects can be invalid for a time interval that is less than or equal to the control system's response latency – the sleep time of control threads plus some time for code execution.

3 Summary

3.1 Contributions of the Work

The thesis has presented a method of reactive Java program checking using the concepts of *thread serialization*, *step-by-step simulation*, *partial JVM functionality replacement*, and *joint execution of simulation of the tested program and of the program's environment*.

Serialization of threads is a method of execution of multithreaded Java programs where just one thread can run at a given time, thread switching is allowed in consistent states only, and the computation path can be controlled by a user algorithm.

In order for the serialization to be able to operate, a *new layer* is inserted between the program and the JVM. The layer has a form of a Java library that replaces some functionality of the JVM, mainly that related to threading. The new layer requires that some method calls be redirected to it which is accomplished by source code modifications.

A program to be checked is executed together with a model of its environment. Both parts run on top of a discrete-time simulation library that implements the idea of thread serialization and adds the concept of discrete simulation time. The resulting simulation is executed in a *step-by-step* manner, one step per one value of simulation time. The environment model has to be created by hand, the checked program can be converted to a convenient form more or less automatically.

The overall concept was demonstrated on a representative feasibility study. It showed an embedded Java control program, its decomposition into smaller parts, and their interconnection. The study also showed what state invariants could be created to be periodically checked during checking.

The method described in this work has brought some ideas that none of other checking tools the author is aware of uses. It has four general advantages over model checking:

- Real Java code is tested, not just a model.
- The program is not tested alone but in conjunction with a model of its environment described by means of Java code.
- There is a notion of time which is completely omitted in other tools. Thanks to introduction of model time and the environment model, a program can be tested with realistic input data coming at the right (realistic) time.
- The method is convenient for infinite reactive programs where model checkers and some other tools fail because they are unable to enumerate all possible states whose number is infinite.

The presented verification method cannot be probably used as the only one during program development. It should rather act as one link of the complete chain of verification tools used from the beginning of a product's development until its launch into real operation.

3.2 Evaluation of Thesis Goals

The author of this thesis is convinced that all four main goals stated in section 1.2 have been achieved:

1. A description/analysis of concurrent Java program behavior was presented and its connection to real Java source code was also given. The identified entities are threads, locks, wait sets, delayed sets, and join sets.
2. A method of reactive Java program development and checking was presented. Some basic concepts, such as consistent states and serialization of threads, were also described. These two features of the presented method provide the ability of managed execution of Java concurrent programs and the ability to take a consistent snapshot of program data. Faulty behavior can be discovered by means of state invariants, behavior protocols, or by analysis of program entities mentioned in point 1. A close-to-real environment is achieved using a model of the checked program's environment and mutual time synchronization of the checked program with the environment model using a common model time.
3. Tools – J-Sim + JiJ and J-SourceMorph – implementing the presented concepts have been developed and are available on-line at <http://www.j-sim.zcu.cz> and www.j-sourcemorph.zcu.cz.
4. A case study was presented. It is a part of the J-Sim distribution archive and can be directly executed from a JAR file.

The thesis has presented a *conceptual and experimental framework* that can be later extended with other concepts and tested on more case studies focused on a variety of problems. In other words, this work should not be understood as a tutorial to a general take-and-use Java software testing method but rather as a baseline for further research, improvements, and extensions.

3.3 Future Work

Future improvements of the method and the tools will be focused mainly on new technologies recently adopted in the industry: RT-Java and the Rate Monotonic Analysis. Both of them are related to real-time computation.

Some other extensions and improvements are planned for the method itself – there is an aim to make the method more versatile to be usable for a broader range of Java programs and, on the other hand, more formal to guarantee certain properties of the testing procedure.

Real-Time Java is a specification and reference implementation of the Java technology for real-time environments. Its specification brings some updates and additions to both the Java Virtual Machine Specification and the Java Language Specification. Improvements of this work over the current state will include new classes in the JiJ package to simulate real-time threads and synchronization, possibly with simulation of real-time garbage collector or a part of an underlying RT operating system.

The Rate Monotonic Analysis is a method of assigning fixed priorities to periodic tasks of a real-time program to maximize the program's schedulability. Every task of the program is given a period T_i and a worst-case execution time C_i , $C_i < T_i$. The program is schedulable if all tasks meet their deadlines, i.e. if they *complete all work before their deadlines* which are by default at the end of their periods.

Multithreaded program schedulability and meeting threads' deadlines could be also verified by simulation in a similar way that other properties are verified, as described in this work. However, the relation between the simulation time and the real time must be carefully defined because the real time is a key element in the area of real-time systems and the Rate Monotonic Analysis.

List of Published Articles

The following papers were published in conference proceedings:

1. J. Kačer: J-Sim – A Java-based Tool for Discrete Simulations. Proceedings of the 23rd International Autumn Colloquium ASIS-2001: Advanced Simulation of Systems. MARQ, Ostrava, Czech Republic, September 2001.
2. J. Kačer: Java Programs Serialization. Proceedings of the 5th International Conference ISM-2002: Information Systems Modelling. MARQ, Ostrava, Czech Republic, April 2002.
3. J. Kačer: Discrete Event Simulations with J-Sim. Proceedings of the Inaugural Conference on the Principles and Practice of Programming in Java (PPPJ-2002). Department of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland, June 2002.
4. J. Kačer, S. Racek: A Method of Java Concurrent Programs Debugging. Proceedings of the 5th International Scientific Conference ECI-2002: Electronic Computers and Informatics. Vienala Press, Košice, Slovakia, October 2002.
5. D. Cudl, J. Kačer, T. Koutný: Comparison-Evaluation of Java-Based Discrete-Time Simulation Tools. Proceedings of the 37th International Conference MOSIS-2003: Modelling and Simulation of Systems. MARQ, Ostrava, Czech Republic, April 2003.
6. D. Hartman, J. Kačer: JUTS – J-Sim Urban Traffic Simulator. Proceedings of the Second International Conference on the Principles and Practice of Programming in Java (PPPJ-2003). Computer Science Press, Trinity College Dublin, Ireland, June 2003.
7. P. Grillinger, J. Kačer: Java Model of Basic Algebraic Structures. Proceedings of the 25th International Autumn Colloquium ASIS-2003: Advanced Simulation of Systems. MARQ, Ostrava, Czech Republic, September 2003.
8. J. Kačer, S. Racek: Model Based Development of Java Embedded Applications. Book of Abstracts of the 5th EuroSim Congress on Modeling and Simulation. EuroSim-FrancoSim-ArgeSim, France, September 2004.
9. J. Kačer: Testing Java Software for Embedded Devices Using J-Sim and Serialization of Threads. Proceedings of the 6th International Scientific Conference ECI-2004: Electronic Computers and Informatics. Vienala Press, Košice, Slovakia, September 2004.

Books or parts of books:

1. J. F. Power, J. T. Waldron (Eds.): Recent Advances in Java Technology: Theory, Application, Implementation. Chapter 3 by J. Kačer: Discrete-Time Process-Oriented Simulation with J-Sim. Computer Science Press, Trinity College Dublin, Dublin, Ireland, November 2002, first edition.

All technical reports from the following list are available on-line at <http://www.kiv.zcu.cz/publications/techreports.php>.

1. J. Kačer: J-Sim – A Java-based Tool for Discrete Simulations. Technical Report DCSE/TR-2001-05.
2. J. Kačer: J-Serializer – Java Programs Serializer. Technical Report DCSE/TR-2001-07.
3. J. Kačer: An Approach to Concurrent Java Programs Debugging and Run-Time Analysis. Technical Report DCSE/TR-2003-14.
4. J. Kačer, S. Racek: Model-Based Development of Java Embedded Applications. Technical Report DCSE/TR-2004-03.

Articles published on WWW:

1. J. Kačer: Lehký úvod do multithreadingu OS/2 (A Gentle Introduction to OS/2 Multithreading). Published on server www.os2.cz.
2. J. Kačer: Tvorba distribučních archivů pro instalátor WarpIN (Creating Distribution Archives for the WarpIN Installer). Published on server www.os2.cz.

References

- [1] G. Brat, K. Havelund, S.J. Park, W. Visser: Java PathFinder – Second Generation of a Java Model Checker. Proceedings of the Workshop on Advances in Verification, Chicago, Illinois, USA, 2000.
- [2] D. Bruening: Systematic Testing of Multithreaded Java Programs. Master’s Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1999.
- [3] D. Bruening, J. Chapin: Systematic Testing of Multithreaded Programs. MIT/LCS Technical Memo, LCS-TM-607, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 2000.
- [4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, H. Zheng: Bandera: Extracting Finite-state Models from Java source Code. Proceedings of the 22nd International Conference on Software Engineering, 2000.
- [5] P. Godefroid: Model Checking for Programming Languages using VeriSoft. Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France, 1997.
- [6] J. Gosling, B. Joy, G. Steele, G. Bracha: The Java Language Specification, Second Edition. Sun Microsystems, 2002.
- [7] A. M. Hagalisletto: Introduction to π -calculus. Lecture notes for DBSEM, Department of Informatics, University of Oslo, Oslo, Norway, 2003.
- [8] J. Hatcliff, M. Dwyer: Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software. Proceedings of CONCUR 2001 – Concurrency Theory: 12th International Conference, Aalborg, Denmark, 2001.
- [9] K. Havelund: Using Runtime Analysis to Guide Model Checking of Java Programs. The 7th International SPIN Workshop, Stanford, California, USA, 2000.
- [10] K. Havelund, T. Pressburger: Model Checking Java Programs Using Java PathFinder. International Journal on Software Tools for Technology Transfer, Vol. 2, No. 4, 2000.
- [11] P. Henderson: Formal Models of Process Components. Workshop on Foundations of Component-Based Systems, Zürich, Switzerland, 1997.
- [12] J. Herma: Konverze zdrojových textů jazyka Java dle externích uživatelských pravidel. (Conversion of Java Source Code Driven by External User-Defined Rules). Master’s Thesis, University of West Bohemia, FAV-KIV, Pilsen, Czech Republic, 2004.

- [13] G. J. Holzmann: The Model Checker SPIN. IEEE Transactions on Software Engineering, Vol. 23, No. 5, 1997.
- [14] A. Igarashi, N. Kobayashi: A Generic Type System for the π -Calculus. ACM SIGPLAN Notices, Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Vol. 36 Issue 3, 2001.
- [15] B. Jacobs, F. Piessens: A π -Calculus Semantics of Java: The Full Definition. Report CW 355, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, 2003.
- [16] J. Kačer: J-Sim – A Java-Based Tool for Discrete Simulation. Master's Thesis, University of West Bohemia, FAV-KIV, Pilsen, Czech Republic, 2001.
- [17] J. Kačer: J-Sim – A Java-based Tool for Discrete Simulations. Proceedings of the 23rd International Autumn Colloquium on Advanced Simulation of Systems. Ostrava, Czech Republic, 2001.
- [18] J. Kačer: Java Programs Serialization. Proceedings of the 5th International Conference on Information Systems Modelling. Ostrava, Czech Republic, 2002.
- [19] J. Kačer: Discrete Event Simulations with J-Sim. Proceedings of the Inaugural Conference on the Principles and Practice of Programming in Java. Maynooth, Co. Kildare, Ireland, 2002.
- [20] J. Kačer: J-Serializer – Java Programs Serializer. Technical Report DCSE/TR-2001-07. University of West Bohemia, FAV, KIV, Pilsen, Czech Republic, 2001.
- [21] J. Kačer: Testing Java Software for Embedded Devices Using J-Sim and Serialization of Threads. Proceedings of the 6th International Scientific Conference ECI-2004. Viena Press, Košice, Slovakia, 2004.
- [22] J. Kačer: Observation and Control of Multithreaded Java Program Execution. Slides from a DSS Seminar. University of West Bohemia, FAV, KIV, Pilsen, Czech Republic, 2003.
- [23] J. Kačer, S. Racek: A Method of Java Concurrent Programs Debugging. Proceedings of the 5th International Scientific Conference on Electronic Computers and Informatics. Košice, Slovakia, 2002.
- [24] J. Kačer, S. Racek: Model Based Development of Java Embedded Applications. Book of Abstracts of the 5th EuroSim Congress on Modeling and Simulation. ESIEE Paris, Noisy-le-Grand, Marne la Vallée, France, 2004.

- [25] J. Kačer, S. Racek: Model-Based Development of Java Embedded Applications. Technical Report DCSE/TR-2004-03, University of West Bohemia in Pilsen, FAV-KIV, Pilsen, Czech Republic, 2004.
- [26] H. Kopetz: Real-Time Systems, Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, Netherlands, 1997.
- [27] D. Lea: Concurrent Programming in Java – Design Principles and Patterns, Second Edition. Addison-Wesley Longman, USA, 2000. ISBN 0-201-31009-0.
- [28] T. Lindholm, F. Yellin: The Java Virtual Machine Specification, Second Edition. Sun Microsystems, 1999.
- [29] J. Magee: LTSA – Labelled Transition System Analyser. University of London, Imperial College of Science Technology and Medicine, Department of Computing, London, UK, 1999.
- [30] J. Magee, J. Kramer: Concurrency – State Models and Java Programs. John Wiley and Sons, UK, 1999. ISBN 0-471-98710-7.
- [31] J. Manson, W. Pugh: Semantics of Multithreaded Java. Technical Report CS-TR-4215, Department of Computer Science, University of Maryland, College Park, USA, 2001.
- [32] R. Milner: The Polyadic π -Calculus: A Tutorial. LFCS Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, Edinburgh, UK, 1991.
- [33] J. S. Moore, R. Krug, H. Liu, G. Porter: Formal Models of Java at the JVM Level – A Survey from the ACL2 Perspective. Presented at the Workshop on Formal Techniques for Java Programs, University Eötvös Loránd, Budapest, Hungary, 2001.
- [34] G. Naumovich, G. S. Avrunin, L. A. Clarke: Data Flow Analysis for Checking Properties of Concurrent Java Programs. Technical Report 98-22, Computer Science Department, University of Massachusetts at Amherst, USA, 1998.
- [35] G. Naumovich, G. S. Avrunin, L. A. Clarke: An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. Technical Report 98-44, Computer Science Department, University of Massachusetts at Amherst, USA, 1998.
- [36] M. Odersky, N. Mihaylov: From π to Java and Back. Lecture notes for *Concurrency: Languages, Programming and Theory*, Programming Methods Laboratory, École polytechnique fédérale de Lausanne, Lausanne, Switzerland, 2003.

- [37] J. F. Power, J. T. Waldron (Eds.): Recent Advances in Java Technology: Theory, Application, Implementation, Chapter 3: Discrete-Time Process-Oriented Simulation with J-Sim. Computer Science Press, Trinity College Dublin, Dublin, Ireland, 2002.
- [38] A. Roychoudhury, T. Mitra: Specifying Multithreaded Java Semantics for Program Verification. Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, 2002.
- [39] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems, Vol. 15, Issue 4, 1997.
- [40] S. D. Stoller: Model-Checking Multi-Threaded Distributed Java Programs. Technical Report 536, Computer Science Department, Indiana University, Bloomington, USA, 2000.
- [41] A. Zimmermann: Petri Nets. Technische Universität Berlin, Germany.

Original Abstract of the Ph.D. Thesis

This thesis deals with a simulation-based method of reactive Java program checking. Reactive programs are mostly infinite concurrent programs that receive their input data during their execution, not at the time when they are started. The presented method is based on serialization of threads and joint simulation of the checked program together with its environment model.

Serialization is a method of concurrent program execution that allows only one thread of a program to run at a given time. Thread switching occurs at exactly defined places only and it is fully under control of a serialization execution engine that makes (pseudo)random decisions about thread switching. A great benefit of the serialization method is a possibility to take a consistent snapshot of a concurrent Java program at defined places.

Joint simulation of both parts can be performed thanks to the fact that both parts run on top of the same simulation library – J-Sim. The library uses model-time and scheduling principles known from the Simula language. The environment model is created as a set of simulation processes and passive data. The checked program is transformed from the original to the required form by a special converter.

The work also presents tools that are used during the checking process. At the end, a representative case study shows how the checking method can be used during development of an embedded reactive program.

Původní abstrakt disertace

Disertační práce se zabývá metodou ověřování reaktivních javovských programů, která je založena na simulaci. Reaktivní programy jsou převážně nekonečné paralelní programy, které získávají vstupní data během svého běhu, nikoliv při startu. Předkládaná metoda je založena na serializaci vláken a společné simulaci ověřovaného programu spolu s modelem svého okolí.

Serializace je metoda běhu paralelních programů, která dovoluje běh pouze jednoho vlákna programu v daném čase. Přepínání vláken je povoleno pouze na přesně definovaných místech a je plně pod kontrolou serializačního stroje, který činí (pseudo)náhodná rozhodnutí o přepínání vláken. Velkou výhodou metody serializace je možnost pořízení snímku konzistentního stavu paralelního javovského programu na určených místech.

Společná simulace obou částí je možná díky tomu, že obě části běží nad stejnou simulační knihovnou J-Sim. Knihovna je založena na principech modelového času a plánování procesů, které jsou známy z jazyka Simula. Model okolí je reprezentován jako množina simulačních procesů a pasivních objektů. Ověřovaný program je převeden z původní do požadované podoby speciálním převaděčem.

Práce také zmiňuje nástroje, které jsou pro ověřování danou metodou potřebné. Ke konci práce je uvedena reprezentativní případová studie, která demonstruje použití předložené metody během vývoje reaktivního programu pro vestavěné systémy.