University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

Model-Based Development of Java Embedded Applications

Jaroslav Kačer, Stanislav Racek

Technical Report No. DCSE/TR-2004-03
September, 2004
Version: 1

Distribution: Public

Technical Report No. DCSE/TR-2004-03
September 2004

Model-Based Development of Java Embedded Applications

Jaroslav Kačer, Stanislav Racek

Abstract

This paper presents a model based method of testing and verification of Java concurrent control programs aimed for embedded devices. Threads of the control program are mapped onto simulation processes that can be executed using discrete model-time concept. The computer surrounding environment, including the controlled device itself, is represented by a simulation model as well. Both the control program model and the device model are written as well as executed using the J-Sim simulation tool. The method allows development of the control program even before the HW platform is developed, including tests of its behavior that can be difficult to perform using a real controlled device.

The research was in part supported by a grant of the Grant Agency of the Czech Republic – *Research of methods and tools for verification of embedded computer systems*, No. 102/03/0672.

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

Contents

List of Figures

1 Introduction

In recent years, the number of Java embedded applications has been growing constantly. Mass usage of smart mobile phones, PDAs, and consumer electronic devices has already become reality. Indeed, Java was intended to be the language of embedded systems from the very beginning. It is expected that the importance of embedded Java market will grow and so will grow the need for tested-off and correct applications, mostly employing Java concurrency as an efficient way of decomposition of the overall task complexity.

An unresolved question of a Java concurrent program is usually the proof of its correct behavior. Since there is not just one possible execution path as in the case of a sequential program, proving correctness of such program is a painful, or even impossible, task. Especially for embedded software, it is also quite problematic to test or debug it on its respective device which usually has only a limited communication interface.

The method of development described in this paper is not able to guarantee 100% correctness of the application. It is a best-effort experimental method that may (and very likely will) detect correctness violations in your program. The probability of discovering such a violation grows with the time spent on testing.

2 Problem Description

2.1 Issues with Java Concurrent Programs

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 unacceptable states such as deadlocks, livelocks, performing a currently forbidden operation, etc.

It is sometimes quite hard to detect the cause of a program failure since the program does not report any error and runs just 'fine'. Also, the correctness violations do not necessarily appear every time a piece of code is executed but usually in some program configuration only, when conditions are favorable for the fault to occur.

Our aim is to detect and report all such violations or user-defined conditions and to analyze the concurrent program at that point, which involves analysis of all possible relitionships between threads, locks, objects, their classes, and corresponding source code.

Controlling a working piece of hardware often requires real-time properties of the application. A hard real-time application cannot be achieved without using special means, such as Real-Time Java Platform [12]. However, we suppose usage of a classic JVM where the required properties are guaranteed by hardware performance and low system load.

Since it is difficult to test the control program in its destination environment, we intend to create a model of the surrounding environment that will cooperate with the tested control program as if it was a real hardware. In the area of our interest, it is of great importance to model the environment properly and to include all relevant details of the real world.

2.2 Related Work

There have been many attempts, to test concurrent programs or to prove certain properties of a concurrent program or its design. Some of them are theory-based methods of formal verification, some are experimental methods, like this one.

Petri nets and the π-calculus are examples of theory-based methods. In [5] and [6], there could be found attempts to formalize Java using the π-calculus and subsequently verify Java programs formally, however it is not possible to find any mention in the above papers how it could be really used.

The LTS Analyzer [7], using the FSP process algebra, is a helpful tool, able to diagnose a system before it is implemented. However, it is just another theoretical method. Alike the model checker SPIN [9] which uses Promela to describe the verified system. SPIN can be extended with Java PathFinder 1 which can create the desired Promela model from a real Java source code. However, the Java language is only partially supported and the state-space of the tested program must be finite.

The model checker VeriSoft [8] deals with real source code. It's main disadvantage is that it is limited to the C language and to the UNIX operating system.

Finally, the ExitBlock algorithm [10] deals with Java bytecode. It is intended for finding errors in concurrent Java programs resulting from unintended timing dependencies. The algorithm is able to execute and test all possible execution paths of the tested program thanks to the Rivet Virtual Machine (VM) that supports checkpointing and rollbacks. Main disadvantage is the key assumption about thread termination after a limited time of computation.

Java PathFinder 2 [11] is another detection tool based on a special VM and working directly with bytecode. It can check for deadlocks and for invariants defined by the user.

This work is also based on experience gained during the FIT project [15], where a similar experimental method was used to verify properties of a break-by-wire embedded application based on the Time Triggered Protocol (TTP). The application was written in ANSI C and it was in fact just a model of real software running on TTP chips and the model of Time Triggered Protocol itself. The work is described in [13], [14], and other publications.

2.3 The Proposed Development Process

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

- A model of the device and the environment has to be created. It is supposed that the model is built up on top of J-Sim [4], a discrete-time simulation library. This is unique in the area of program testing – none of the tools from section 2.2 uses joint testing of the application and a model of the environment.
- The control program source code has to be modified slightly. All threading-specific constructs have to be replaced by their J-Sim counterparts, e.g. a subclass of `Thread` will become a subclass of `JSimProcess`[1], etc. The conversion can be made either manually or (better) automatically.
- The two source code parts must be merged together into one J-Sim simulation application what means that they have common planning of simulation processes (i.e. one common event list). Some "diagnostic" parts of the code can be added moreover (e.g. see the "observer" process below).
- The simulation application can be executed and results are obtained. Just note that both the device model and the control program run in model time. The results can be e.g. collected by a special "observer" simulation process that performs all user-defined tests with a certain period. Both model and control program data are relevant to be used within the tests. It is important that all the performed tests can be made non-intrusive, i.e. they don't influence neither the control program function nor the dynamic behavior (model time is used instead of real time). The simulation model behavior can be visualized, what is convenient for the testing purpose. Any revealed bug within the control program can be easily removed and the tests can be repeated.
- At the end of the process, the (debugged) control program source code can be separated from the model and straightforwardly converted to the production form. Only the control interface object (see below in

[1] More precisely, it will become a subclass of `cz.zcu.fav.kiv.jsim.jij.JavaThread`, which in turn is subclassed from `JSimProcess`.

3.5) should be implemented other way in order to use real HW registers instead of the model data items.

The process and its key steps and components are depicted in figure 1. Just note that both the model and the application run in model time.

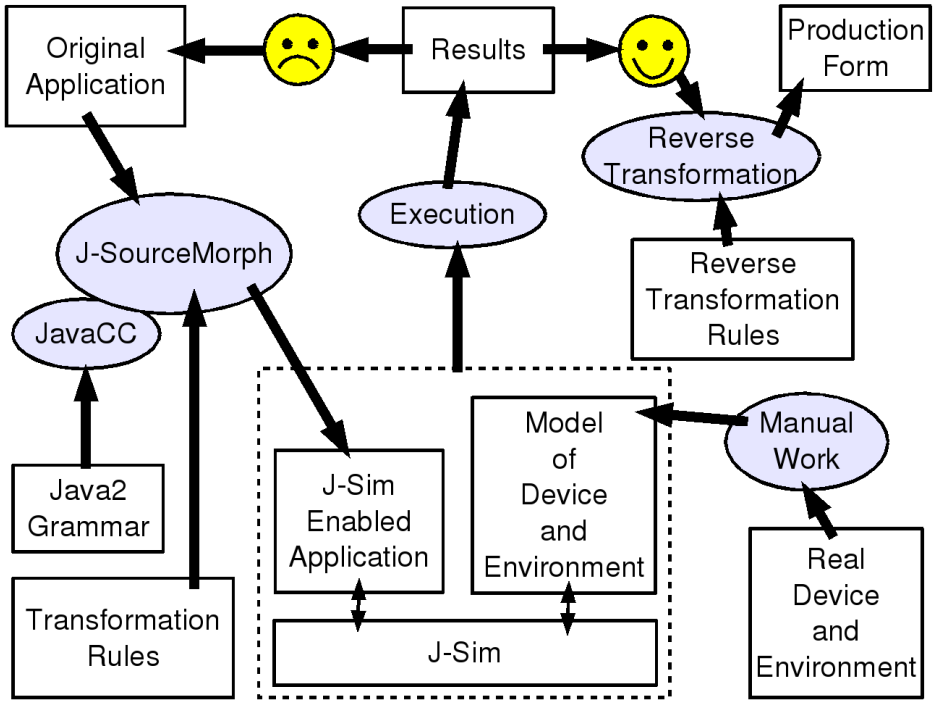

Figure 1: The Proposed Process of Developing a Concurrent Java Application for Embedded Devices

Out of scope of this paper is a detailed organization of the tests. It is obvious that a single run of the simulation is not enough to test a complex concurrent neverending program with virtually unlimited number of possible behaviors. Some general recommendations are given below in the part 5.

In general, the probability of discovering an error within the control program functionality is directly proportional to the time spent on testing and to the number of different parameter settings of the model (timing, distributions of random numbers, etc.). Unlike theoretical methods, this experimental method is not capable of 100% correctness proof (but the incorrectness can

be clearly proven with certainity).

The idea presented here depends on and extends previous work described in [1], [2], and [4]. While [1] and [2] describe an approach to runtime analysis of a 'classic' desktop concurrent programs, [4] and other J-Sim related papers focus on creating a model from scratch. If the two software modules (the tested control program and the model of its environment) are merged together with J-Sim, one can more or less easily verify a concurrent program using a standard PC before the program is actually loaded into a device.

3 Model Architecture Overview

3.1 Model Structure

This section presents the overall structure of the simulation model, its main modules and mapping between modules used during normal operation mode and testing mode on a PC. An overview can be seen in figure 2.

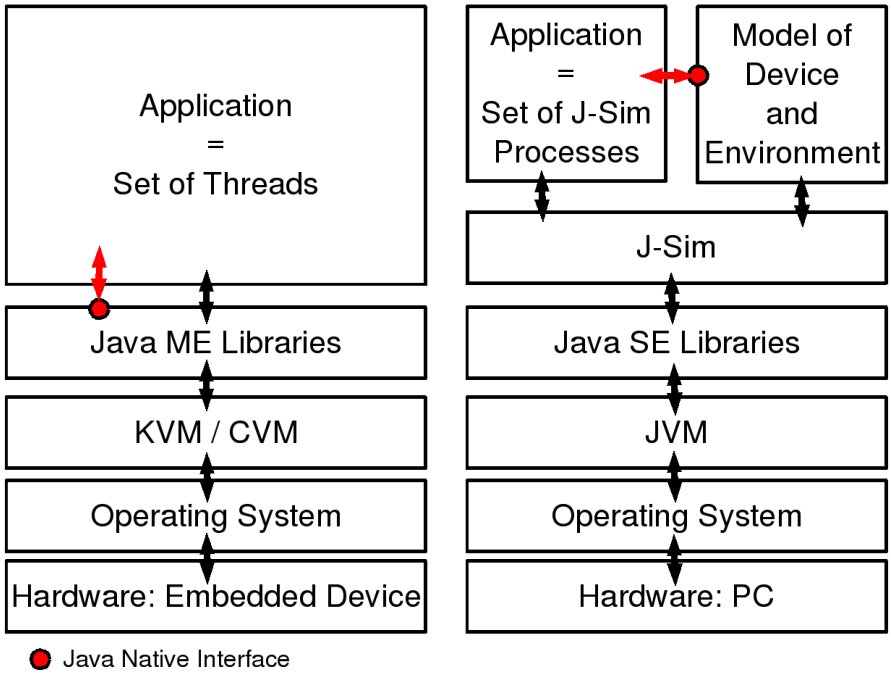

Figure 2: Modules Present During Normal Operation and During Testing

The main diffencies between the considered modes are as follows:

- The embedded computer device is replaced with a PC or any other personal computer.
- The OSes used differ, however their function stays the same (in fact the JVM masks the lower layers of SW and HW creating a kind of universal machine interface).

- In embedded computer devices, the SE Java Virtual Machine is not usually used due to its resource requirements. Instead, there is a ME equivalent called KVM or CVM. Some JVM functionality is missing and some is added. This applies mainly to Java libraries.
- The original control program is replaced with its modified version, running on top of J-Sim. The model of the environment must be added to the system in order to simulate the device and the environment. Since the original application communicates with the environment, all this communication must be replaced with calls to some parts of the model responsible for this.

Concerning the communication of the control program with lower layers, there should be a well defined interface(s) that all communication will go through. This (abstract) interface is the same for both modes. In the testing mode, the interface is implemented by a class of the environment model and changes/returns its state. In the normal operation mode, the interface is implemented by a part of the application that does the real communication with hardware, e.g. using the JNI.

The aim is to keep the application source code as little changed as possible in order not to divert from its original behavior. The changes involve only threading-specific commands (thread methods, synchronization, ...) and the communication functions. Java is extremely convenient for performing the former set of changes because the threading interface is standardized in the language. So this task can be done more or less automatically.

3.2 J-Sim

The J-Sim simulation library [16] is required by both the model and the tested control program. Actually, it is a 'glue' that links the two parts together by means of a simulation object that contains processes from both parts. The model part, created from scratch, is designed as a set of J-Sim processes from the beginning. The application part has to be converted to the desired form either manually or automatically.

The library brings the idea of discrete-time process-oriented simulation, known from Simula, to Java. A simulation is expressed as a set of processes, sharing the same simulation (model) time. Activity of a process always takes zero simulation time and the process can suspend itself between two consecutive activities which creates a 'gap' in the simulation time and advances it as

well. A process activity can finish calling either `passivate()` or `hold(gap)` Simula-like operations. In the second case, the next activity of the calling process is planned (using an event list data structure) at the simulation time `current_time + gap`. The simulation time cannot go backwards, process execution is therefore ordered according to the simulation time of their next activity. This way the simulation processes are interleaved, i.e. other processes can be scheduled within the simulation time gap between successive activities of every process.

More about writing simulation models can be found for example in [4] and on the J-Sim home page [16].

3.3 The JiJ Package

The J-Sim tool was found to be a convenient starting point for dicrete-time simulation of Java multithreading according to the rules and principles described later in section 3.7. However, it was originally designed to support a lightly different type of simulation. It was also necessary to provide 1:1 mapping between the original Java code and the converted simulation code, which was clearly impossible to be achieved with single class `JSimProcess`.

It was therefore decided to derive new classes from `JSimProcess` and `JSimSimulation`, focused strictly on Java multithreading simulation. The new classes (plus several other classes) form a new package called *JiJ*[2] – JiJ here means *Java in Java* because simulation of Java programs is executed inside Java.

The JiJ package is discussed in detail in section 4.

3.4 Conversion

Currently, conversion of contol program source code to the simulation version and back is performed manually but a conversion tool [19] has already been constructed.

The tool is based on JavaCC [18], a Java parser generator. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. Since JavaCC comes with a complete grammar of the Java language used in J2SE 1.4 (and even

[2]More precisely, `cz.zcu.fav.kiv.jsim.jij`

newer J2SE 1.5 aka Tiger), all that must be done is just to program an adequate action when a certain token is read.

The conversion tool is parametrized by an XML file where transformation rules are stored. A rule can apply to a class, an attribute, a method/constructor header, a method/constructor call or import commands. When a searched token is found in the source code, an operation can be performed on it, for example it can be deleted completely, replaced by another name, etc. Method calls can also be added to the modified code.

The conversion utility supports macros and their expansions. There is one pre-programmed macro coming with the tool, `LOCK`. It is able to find the respective lock object to a `wait()` method call. The lock object need not be always specified. In case of a synchronized method, `this` is used as the lock, or `ClassName.class` for static methods. In case of synchronized blocks with explicit lock, the (same) lock must be prepended before the `wait()` method invocation – `myLock.wait()`.

So if any call to `wait()` is replaced with `sim.getLock($LOCK$).wait_JiJ()`, the resulting source code after macro expansion can look like `sim.getLock(this).wait_JiJ()`, or `sim.getLock(ThisClass.class).wait_JiJ()`, or `sim.getLock(explicitLock).wait_JiJ()`, depending on the context.

The same macro can be used for replacing `notify()` and `notifyAll()`.

The conversion of a class involves the following changes (not all of them are listed):

- In case of a thread class, the superclass is changed to `JavaThread` or its subclass. This ensures that the thread will be executed by J-Sim in a deterministic way. The constructor has to be changed too and the `run()` method has to be converted to `run_JiJ()` appropriately.
- Checking synchronized blocks (locking and unlocking) is necessary. All locks used for synchronization must be registered and their wait sets[3] and delayed sets[4] must be maintained.
- All thread-state-manipulation methods, such as `wait()`, `notify()`,

[3] A wait set of a lock contains all threads that have invoked its `wait()` method and have not been notified yet.

[4] A delayed set of a lock containes those threads that are trying (still without success) to enter a synchronized block guarded by the lock.

etc. have to be replaced with their model version. This includes also `sleep()`, `yield()`, `setPriority()`, ...

3.5 Model of Control Interface

Control interfaces of embedded applications can be roughly distinguished as follows:

Passive Interface. Data items of the interface are written or read periodically by the activity of control program. It corresponds to embedded control systems that are denoted as *time-triggered* [17].

Active Interface. A part of the interface serves as an interrupt controller interface. When an interrupt occurs (i.e. when an event occurs within the controlled object), its service routine (part of the control program) is activated to manage the event. It corresponds to embedded control systems that are denoted as *event-triggered*.

The method described here can use both the models of passive or active interface. We use two kinds of control interface models that differs in the level of abstraction and in the corresponding Java language description:

Java Interface. This model contains only function (method) types, no implementation. It means that it is not necessary to change the model when passing from the model version to a production-version of the developed (tested) control program.

Java Class. A model of this level should to implement the corresponding abstract interface. It means that within the process of an embedded application development we need to make two implementations - the first aimed to be used as a part of simulation model and the second aimed to be used with the real control program.

The relationship between the simulation-version and production-version control interfaces is depicted in figure 3. Note that while the declaration part (a Java `interface`) remains the same, its implementation (a Java `class`) changes. However, the two different implementations are completely hidden for the control program since it communicates via an object typed as an

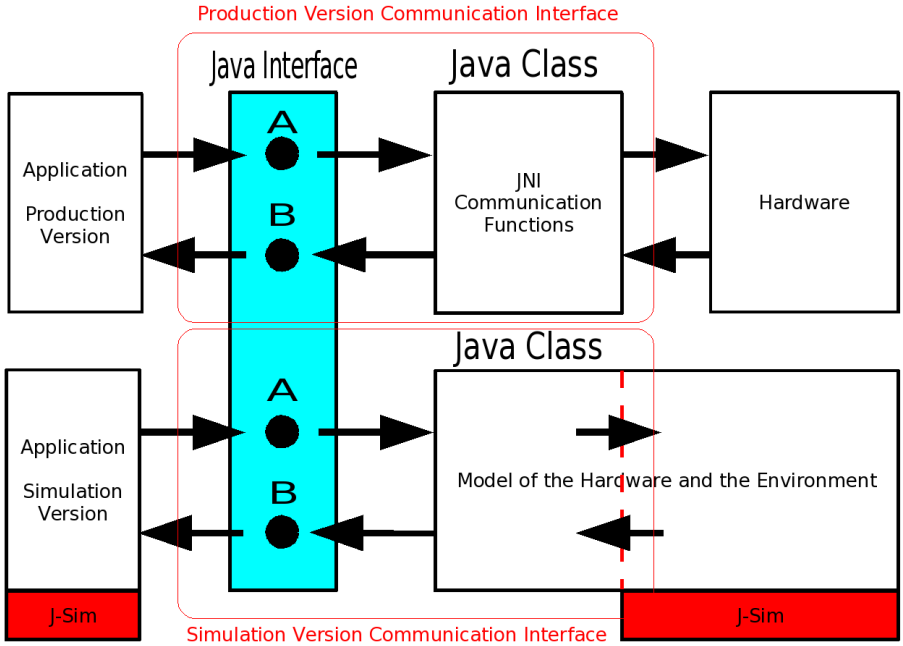

Figure 3: The Differences between the Simulation-Version and Production-Version Control Interfaces

`interface`. The only required change is the actual creation of the communication object where a different class (or constructor name) is used after the `new` keyword.

The method described here can be straightforwardly extended for a distributed embedded application as well. Then we have to use two kinds of computer-environment interface – *control interface* (i.e between a computer node and its part of controlled device) as well as *network interface* (e.g. a serial bus interface). Clearly, then the model complexity will arise rapidly, because the model should include all the nodes of distributed system and a (sub)model of communication protocol as well [15].

3.6 Model of the Environment

The proposed method assumes that the model is constructed atop 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 (statement-like value) from a data structure that serves as the model of 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. In reality the data items of computer-environment interface are usually located in HW registers and accessed via Java Native Interface methods of the control program that manipulate the data using low-level (e.g. assembler) commands. The tested application must replace these methods with methods interacting with the model before the testing experiments are started.

So the data implementing the model of control interface are in discrete-time points read/modified by:

- Simulation processes of the model of controlled environment. These processes usually implements both functional and dynamic properties of the environment.
- Simulation processes modeling the functionality and dynamics of threads of the control program.

3.7 Model of Control Program Activity

As mentioned before, not the original application program, but its model is tested by the presented method. The following list defines terms necessary to understand the process of the control program 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 (subprocesses) that are interacting using a set of monitors (shared objects). The

dimension of both sets can change but let's consider it constant for simplicity.[5]

Process of a Thread Execution. It is an independent subprocess CP_DT_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* – The thread is running or it is able to run immediately. When runnable, it can be either within a part of *local computation* (i.e. it operates with its own data only) or within a part of a *monitor method call*[6]. Both cases of computation are considered to be *atomic* in the sense that when started it has to be finished and no other thread can influence correctness of the result and consistency of the used data items. Monitor method call can change the calling thread state from *running* to *delayed*.
- *Delayed* – The thread is delayed when it tries to enter a synchronized block of code that is currently inaccessible due to its respective lock's state. The lock is owned by another thread and the thread in question must be prevented from continuing. However, this state is different from the state *non-runnable* because waiting on locks implemented in Java is *active* (in contrast to some other synchronization mechanisms using semaphores).
- *Non-runnable* – The thread waits for an event or for the end of a time interval, or both. We will recognize several reasons for a thread to become non-runnable: waiting within a monitor method on `wait()` or `wait(howLong)`, waiting caused by Java `sleep(howLong)` method call, or waiting caused by `join()`/`join(howLong)`.

  The Java thread is put to *running* or *delayed* state after the respective condition is fulfilled or the specified time elapses, whichever occurs first.

Java threading mechanism does not distinguish between the two first states. They are both merged into one state, called *runnable*.

[5] This assumption 'cuts off' the inherent indeterminism of dynamic behavior caused by the garbage collector in Java. When e.g. RT Java is used, the principles of the method still hold for a changeable set of run-time objects.

[6] It means that local part finishes when a synchronized method is called. Synchronized method call is used whenever the thread needs to interact with its environment – typically local parts of computation and monitors method calls are regularly altered.

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. One event in the sequence is a change of the interface data item including a 'time stamp' of the change.

To proceed from the real control program behavior to a model of its behavior, let us assume several restrictions as for the program control flow:

1. The control program is executed at a one-processor computer.
2. Every thread's local part of computation is executed at once.[7]
3. The same as in point 2 applies for a monitor procedure call.[8]

Note 1: The assumptions given above mean a limitation of a 'real concurrency' of CP_D. Instead of it, a kind of 'interleaved concurrency' is assumed.

Note 2: At the first glance, the assumptions given above seem to be far from reality. In fact, for embedded (real-time) applications, the computer should not be loaded too much and no part of a thread computation should not last too much as well, so for a wide variety of embedded applications the assumptions fit quite well.

Note 3: In fact these assumptions are not substantial for the presented method practical utilization. Within an application of the described method they could be "bypassed" a way. Here they are used to have more clearly defined concepts used within this explanation.

Using the given assumptions, we can introduce some additional concepts concerning the CP_D:

Consistent State of CP_D. It can be reached when all the runnable threads are stopped after the (current) local part of their computation has been performed (i.e. at the entry point of a monitor procedure call).[9] Then the data of all objects (either threads or monitors)

[7] It corresponds to the case when all the threads have the same priority and no context switching among threads with the same priority occurs.

[8] Monitor procedure execution can be interrupted using the `wait()` function. Then we assume that all the parts of the procedure computation between successive `wait()` calls are executed at once.

[9] When the assumptions apply, the consistent state is reached when (one) running thread finishes its current local part of computation.

should be consistent and a consistent state of CP_D can be constructed as a set of consistent states of all objects that the process of control program computation is composed from.

State Space Behavior. It can be defined as a sequence (possibly infinite) of consistent states that the CP_D is passing through. Both input/output and state space behavior can be taken as possible kinds of CP_D description.

Now we can 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 (i.e. a thread local part or a part of monitor procedure call) 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 by the J-Sim scheduler.

Then the CP_D^* computation can be performed as a sequence of atomic parts of all threads computations, where the order of atomic parts is randomly chosen[10] *and 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. Using common simulation time, the CP_D^* activity can be properly interleaved with the EP_D^* activity, i.e. with discrete time model of the control program environment activity. Moreover both activities (simulation worlds "control program" and "environment model") can influence each other using the (common) model of the control interface.*

To create the control program model, it is necessary to modify the original source code, therefore the static view of the model CP_S^* differs from that of the original program code CP_S. However, their cardinalities are equal and the mapping from CP_S to CP_S^* is an isomorphism, what is important, because we need to use the inverse mapping as well when passing from the (tested off) model code back to the original real-world code. The necessary changes between CP_S and CP_S^* should be as few as possible.

CP_S^* code is then merged in the conventional way with the EP_S^* (i.e with the source code of the model of the controlled environment). The reqired

[10] But still, all rules of Java scheduling must be respected.

run-time behavior of the model (i.e. discrete-time sequence of atomic parts of computation) is then reached thanks to J-Sim run-time kernel that interleaves both parts of the overall model activity using the common event list.

When we assume that the EP_D^* (i.e. the modeled environment behavior) can be made (by a 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 a proper CP_S^* and EP_S^* construction) arbitrarily close to the real system (i.e. computer + CP_S + environment) behavior.

Notes:

1. The assertion stated above can hardly be proven using a mathematical way. It would need all the concepts used above to be formally defined, including a formal description of the Java language semantics. Moreover all the possible controlled environments should be considered (and formally described), what is clearly impossible. Fortunately, the usefulness of the (experimental) method can be evaluated experimentally – using it within a variety of case studies including the model validation, i.e. a comparison of model based tests with their real-world counterparts.

2. It has no great sense to try to have the model behavior exactly the same as the real-world system behavior. The environment behavior is frequently a random process (mostly infinite), so the control program execution path within its state space fluctuates as well (it could be e.g. mathematically described as a stationary random process). Practically it means that we need not to know precise values of delays of single atomic parts of computation, instead of it we should use random delays. It should lead to a better coverage of the model based tests of the control program functionality, e.g. possible troubles of concurrent computation (deadlocks, race conditions, etc.), should be revealed more easily.

3. The `wait()` and `sleep()` replacements have to declare the `InterruptedException` to be possibly thrown out, as their originals do. Then, the surrounding code (`try` and `catch` blocks) need not be modified. The exception will be actually thrown out when the `interrupt()` method of the suspended thread is invoked so exception

handlers will also be tested. This is important, because every embedded system should be at least *fail-safe*, i.e. the control program (whatever way it finishes) should be able to pass the controlled device into a safe state.

4 JiJ – Java in Java Simulation

The JiJ package is an extension of the J-Sim library for simulation of Java multithreaded programs. It provides a replacement class for `java.lang.Thread` and replacement methods for `wait()`, `notify()`, and `notifyAll()` methods of class `Object` while keeping the main properties of classic J-Sim simulation: step-by-step execution on user request and discrete values of simulation time, one value valid for a whole simulation step.

In the same time, it allows execution of 'new' processes – Java threads together with 'classic' J-Sim processes used to describe the environment model, as described in sections 3.6 and 6.3.

4.1 Requirements

Let's form some requirements that the JiJ package should meet in order to be usable for mixed-mode simulation described in chapter 3.

1. It must provide a replacement for the `Thread` class, including its most important methods, like `run()`, `start()`, `sleep()`, `join()`, `interrupt()`, `interrupted()`, etc.

2. It must be able to simulate synchronization used in monitor methods, including `Object`'s methods for controlling thread execution state: `wait()`, `notify()`, and `notifyAll()`. The simulation must be able to handle nested synchronization (nested synchronized blocks or calling synchronized methods from other synchronized methods).

3. Thread execution state must be maintained by both thread methods and lock methods. The state must be taken into account during thread scheduling as in the original Java scheduling strategy.

4. During one simulation step, either a classic J-Sim process or a JiJ `JavaThread`-process can be given control. In the former case, the simulation step finishes when a `passivate()` or `hold()` method is called. In the latter case, the simulation step finishes when a consistent state of the program is reached, which happens at the following places:

   - at the beginning/end of every synchronized region of code;
   - at the place where `wait()` is invoked;

- at the beginning/end of every thread's `run()` method when a thread 'is born' or 'dies';
- at the place where `sleep()`, `join()`, `yield()`, or `setPriority()` is called.

5. The simulation time must be shared by both the control program simulation (the JiJ part) and the outer world model (the 'classic' part). All actions in the classic part are executed in zero-long simulation time interval and the simulation time is advanced by `hold()` and `activate()` method calls, as it is usual in Simula-like systems. All actions in the JiJ part are executed in non-zero-long simulation time interval and the simulation time is advanced "automatically" by the time spent by the computation[11] when a consistent state is reached. The simulation time can be also advanced by calls to `sleep(howLong)`, `join(howLong)`, or `wait(howLong)` if there is currently no runnable thread in the JiJ part of the simulation.

6. When selecting the next thread to run, all rules of Java scheduling must be respected. The rules include the state of every tread (runnable/non-runnable), its priority, and lock requirements and ownership[12]. Usually the simulation time value or the total time spent by a thread or any other quantity should not be taken into account.

4.2 New Classes

The most important classes of the JiJ package are shown in figure 4.

They will be explained below in detail below, here is just a basic summary:

- `JavaThread` replaces `Thread` and all its methods important for our goals.
- `JavaLock` replaces functionality of `Object`'s methods `wait()`, `notify()`, and `notifyAll()`. Since any object can be used for synchronization in Java, `JavaLock` instances cannot be created in the source code of the simulation version of the tested program. Instead, they are managed by a lock repository.

[11] or any value derived from it or even a reasonable random number

[12] A thread that does not own a lock synchronizing a block of code cannot enter the block and therefore it cannot get control now. Let's call it a *delayed thread* according to terminology from section 3.7.

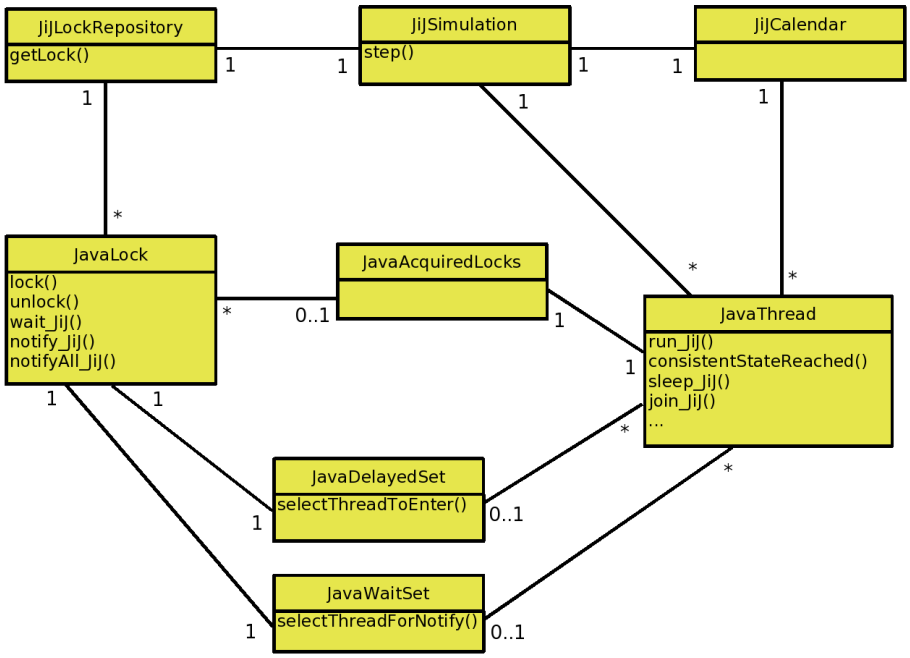

Figure 4: Classes of the JiJ Package

- A `LockRepository` instance manages relationships between original locks (instances of any class) and simulation-version locks – instances of `JavaLock`. A lock repository is able to return always the same `JavaLock` instance for the same synchronization object. If no `JavaLock` exists yet for the supplied object, a new one is created and returned.

- Various relations between threads and locks are expressed by classes `JavaAcquiredLocks`, `JavaWaitSet`, and `JavaDelayedSet`. `JavaAcquiredLocks` manages relations leading from a `JavaThread` to (possibly) multiple `JavaLocks`. It stores all locks currently owned by a thread. To 'own a lock' means to be permitted to enter code synchronized with the lock. Classes `JavaWaitSet` and `JavaDelayedSet` manage relations in the opposite direction. For a given lock, they store threads that are waiting on `wait()` inside a block of code synchronized with the lock (`JavaWaitSet`) or threads willing to enter –

still unsuccessfully – such a block (`JavaDelayedSet`).

- A `JiJCalendar` manages the simulation time and selects the next thread to run. It also manages so-called *wake-up events*. A wake-up events is created whenever a thread performs an operation that switches it to a non-runnable state for a given time period, e.g. `sleep(howLong)`. An event carries information about the respective `JavaThread` and the time when it should be re-activated.

- A `JiJSimulation` is a 'container' for all threads, locks, and other elements. It provides the `step()` method that performs exactly one simulation step.

4.3 `JavaThread` – Simulation of `java.lang.Thread` Functionality

The `JavaThread` class adds new functionality to `JSimProcess` from which it is extended. New constants, attributes, and methods have been added. A `JavaThread`'s behavior is not determined only by calls to its own methods but also by calls to methods of `JavaLock` instances during a thread execution. Since `JavaLock` method headers do not include a reference to the currently running thread[13], the necessary reference is obtained via `JSimSimulation.getRunningProcess()` inside these methods.

4.3.1 New Process States

In addition to `JSimThread`'s original states, new states are introduced in `JavaThread`:

- `STATE_RUNNABLEINCONSISTENTSTATE` – A `JavaThread` gets to this state when it reaches a consistent state and still remains in runnable state. Such a thread can be selected to run in the next simulation step.

- `STATE_BLOCKEDONJAVASYNCHRONIZATION` – A `JavaThread` gets to this state when it reaches a consistent state formed by a beginning of a syn-

[13]Specifying the running thread would not correspond to the principles used in Java. Also, it would be insecure due to possible fake reference. Moreover, it would be impossible to determine the reference in methods of classes other than thread classes, e.g. monitor classes.

chronized block (a JavaLock's lock() method) and cannot continue because the lock is already owned by another thread. The thread can get to STATE_RUNNABLEINCONSISTENTSTATE later when the lock is released in unlock() and this thread is selected to enter the synchronized block.

- STATE_SLEEPING – A JavaThread gets to this state after invoking sleep_JiJ(howLong). After the specified (simulation) time elapses, the thread is switched to STATE_RUNNABLEINCONSISTENTSTATE during a simulation step.

- STATE_SUSPENDEDONJAVAJOIN – A JavaThread gets to this state after invoking join_JiJ() on another JavaThread if the target thread has not terminated yet. The thread is switched back to STATE_RUNNABLEINCONSISTENTSTATE when the target thread finishes.

- STATE_SUSPENDEDONJAVAWAIT – A JavaThread gets to this state after invoking wait_JiJ() on a JavaLock. It can get back to STATE_BLOCKEDONJAVASYNCHRONIZATION when it receives a notification signal via the lock's notify() or notifyAll() from another thread.

When a JavaThread is just running during a simulation step execution, it is in the STATE_ACTIVE state inherited from JSimProcess. Other inherited states (STATE_SCHEDULED, STATE_PASSIVE, STATE_BLOCKEDONSEMAPHORE, STATE_BLOCKEDONMESSAGESEND, STATE_BLOCKEDONMESSAGERECEIVE) are not used.

State switching is performed by the setProcessState() method that overrides JSimProcess.setProcessState(). A JiJInvalidJavaThreadStateException exception is thrown out if the specified state cannot follow the thread's current state. All JavaThread states and transitions between them are shown in figure 5.

4.3.2 Methods Leading to a Consistent State

The methods that lead a JavaThread to a consistent state can be roughly sorted into two main groups:

1. Methods of the class JavaThread itself.

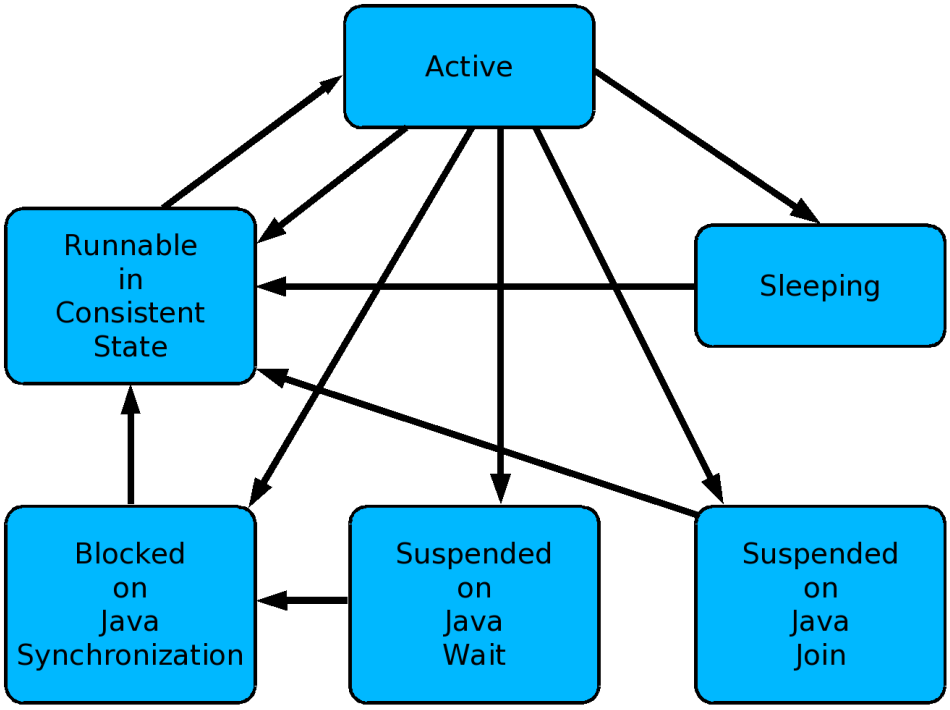

Figure 5: `JavaThread` States and Transitions Between Them

2. Methods of the `JavaLock` class. In order for a consistent state to be reached, the running thread must be determined first and then its reference must be used to invoke a `JavaThread` method corresponding to the `JavaLock` method.

The first group of methods includes: `join_JiJ()`, `sleep_JiJ()`, and `yield_JiJ()`.

The second group includes: `blockOnJavaLock()`, `passOverJavaLockBegin()`, `passOverJavaLockEnd()`, and `suspendOnJavaWait()`:

- `blockOnJavaLock()` is called from `JavaLock.lock()` if the lock is currently unavailable. It transfers the thread to state `STATE_BLOCKEDONJAVASYNCHRONIZATION` and then calls `consistentStateReached()`. There is a complementary method

`unblockFromJavaLock()` that is called from `JavaLock.unlock()` when another thread frees the lock and this selected as the only thread allowed to get the lock and enter the corresponding synchronized code. The thread is then transferred to state `STATE_RUNNABLEINCONSISTENTSTATE` and it is ready to be run during a next simulation step.

- `passOverJavaLockBegin()` is called from `JavaLock.lock()` if the lock is free and the runing thread can get over it. It transfers the thread to state `STATE_RUNNABLEINCONSISTENTSTATE` and then calls `consistentStateReached()`.

- `passOverJavaLockEnd()` is called from `JavaLock.unlock()`. It transfers the thread to state `STATE_RUNNABLEINCONSISTENTSTATE` and then calls `consistentStateReached()`.

- `suspendOnJavaWait()` is called from `JavaLock.wait_JiJ()`. It transfers the thread to state `STATE_SUSPENDEDONJAVAWAIT` and excludes the lock from the set of locks owned by this thread. Then it calles `consistentStateReached()`. There is a complementary method `resumeFromJavaWait()` that switches the thread back to state `STATE_BLOCKEDONJAVASYNCHRONIZATION`. The reason for not using directly `STATE_RUNNABLEINCONSISTENTSTATE` is that the thread must compete with other threads to get the lock as if it were entering a synchronized block of code from its beginning.

The `consistentStateReached()` method first informs the simulation that the thread is about to reach a consistent state. This allows the simulation to compute the time spent by the thread and update the value of simulation time and the JiJ calendar. Then the `mainSwitchingRoutine()` method, inherited from `JSimProcess`, is called which assures the necessary switching back to the main simulation thread from which `step()` was invoked. The `step()` method can finally be completed and the simulation step is finished.

4.4 `JavaLock` – Simulation of Synchronization

The `JavaLock` replaces threading-specific functionality encoded in the class `Object`. There are five principal methods offered by `JavaLock`:

- `lock()` – Replacement of 'invisible' action taking place at the beginning of every synchronized block. It either allows or disallows a thread

to enter a synchronized block. In case of a thread that is allowed to continue, its `passOverJavaLock()` method is called. In the other case, `blockOnJavaLock()` is called. A consistent state is reached in any case.

- `unlock()` – Replacement of 'invisible' action taking place at the end of every synchronized block. The running thread never gets blocked here but a delayed thread (if a delayed thread exists) may get permission to enter a synchronized block via its `unblockFromJavaLock()` method. The running thread's `passOverJavaLockEnd()` method is always called which leads to a consistent state.

- `wait_JiJ()` – Replacement of `Object.wait()`. Releases the lock, calls the running thread's `suspendOnJavaWait()` which in turn calls `consistentStateReached()`. If the running thread is not the lock's owner, an `IllegalMonitorStateException` is thrown out.

- `notify_JiJ()` – Replacement of `Object.notify()`. Does not release the lock. If the lock's wait set in not empty, a thread from the set is randomly selected and its `resumeFromJavaWait()` method is called, which transfers the selected thread to state `STATE_BLOCKEDONJAVASYNCHRONIZATION`. The thread must compete with other threads to become the lock's owner when the lock is released by its current owner with `unlock()`.

- `notifyAll_JiJ()` – Replacement of `Object.notifyAll()`. Its behavior is identical to `notify()`'s behavior with one exception: All threads from the wait set are resumed so the wait set is always empty after `notifyAll_JiJ()`.

`JavaLock`s cannot be created in the standard way in the tested program because their originals are not created explicitely for the purpose of synchronization. Instead, they are normal objects, as all others. One object can be used as lock at many different places which would cause difficulties if we had to create a unique `JavaLock` for every synchronization object explicitly.

Therefore, a different approach is taken in the JiJ package. There is a repository of all `JavaLock`s that have ever been created. Together with them are stored the original locks. When a `JavaLock` is needed for an object and it does not exist yet, a new one is created and returned. Otherwise the stored `JavaLock` is returned. This assures that the same `JavaLock` is always

used instead of the same original lock and that synchronization will behave in exactly the same way as in the original program.

The lock repository is an instance of `JiJLockRepository` and there is one repository per a JiJ simulation. It is accessed via the `getLockForObject()` method of the JiJ simulation[14]. The reference is not stored to any variable but immediately used for method invocation:

```
simulation.getLockForObject(this).wait_JiJ();
```

4.5 Thread-Lock Relationships

The relationships between threads and locks are maintained by 3 classes:

- The `JavaAcquiredLocks` stores all locks that are currently owned by a thread. Every `JavaThread` contains a `JavaAcquiredThreads` instance. Together with the locks are stored their levels of nesting. If, for example, a thread passes twice over the `lock()` method of the same lock, its level of nesting is equal to 2. If then the thread passes over `unlock()`, the lock is still owned by the thread. The level of nesting must reach zero in order for the lock to be marked as free.
- The `JavaWaitSet` and `JavaDelayedSet` classes work in exactly the opposite way: for a given lock, they remember all threads that are suspended inside `wait()` of the lock or threads that try – still unsuccessfully – to pass over the `lock()` method of the lock. Both of them have a method that inserts a thread to the set – `insertThread()`. `JavaDelayedSet` also provides `selectThreadToEnter()` that selects a thread that will be allowed to get over `lock()`. `JavaWaitSet` has `selectThreadForNotify()` and `selectThreadsForNotifyAll()` that select one thread/all threads that will be notified by `notify()`/`notifyAll()`.

4.6 `JiJCalendar` and Scheduling Principles

As stated in section 3.7, two parts execute together during JiJ simulation: the model of the control program and the model of the outer environment.

[14] A reference to the simulation should always be added to attributes of every class of the tested control program.

Since the model of the control program runs on top of J-Sim JiJ package and the the model of the environment runs on top of 'classic' J-Sim, different scheduling principles hold in each part.

The 'classic' J-Sim part uses scheduling princliples known from the Simula language:

- All processes of the simulation run 'in parallel'. It means that every process has its own axis of simulation time.
- Every action execution is zero time units long (measured in simulation time), i.e. it is just a point on the time axis of a process.
- There are time gaps between successive actions of the same process. The simulation time changes directly from the last action's time to the new action's time. There is nothing in between.
- During simulation, all events of all processes are put together and sorted by their simulation time. Then they are executed in this order so between two actions of the same process there may be a number of actions of other processes. This principle is known as *event interleaving*.

On contrary to the above, the following principles hold in the JiJ part:

- Threads do not run in parallel because there is just one processor shared by all of them. If thread T_1 runs at simulation time t_1, thread T_2 cannot run at the same time and its execution must be postponed to the future.
- Every action execution is non-zero time units long, i.e. it is a non-zero time interval on the common time axis.
- If there are runnable threads, there are no gaps between activities of different threads or between activities of one thread. By activity, we mean a transition from a consistent state to another consistent state, i.e. execution of a part of a thread's code. If there are currently no runnable threads, there may be a gap where no thread can be scheduled. This situation can happen if all threads are sleeping using `sleep()`.

- Although simulation time is advanced during execution of `JavaThread`s, it has no direct impact on scheduling, i.e. selection of the next thread. The thread to run in the next simulation step is selected using Java scheduling rules (thread state, priorities, ...) and also random selection.

Let's consider a joint simulation of two `JavaThread`s T_1 and T_2 and two `JSimProcess`es P_1 and P_2. A possible scenario of its execution is shown in figure 6.

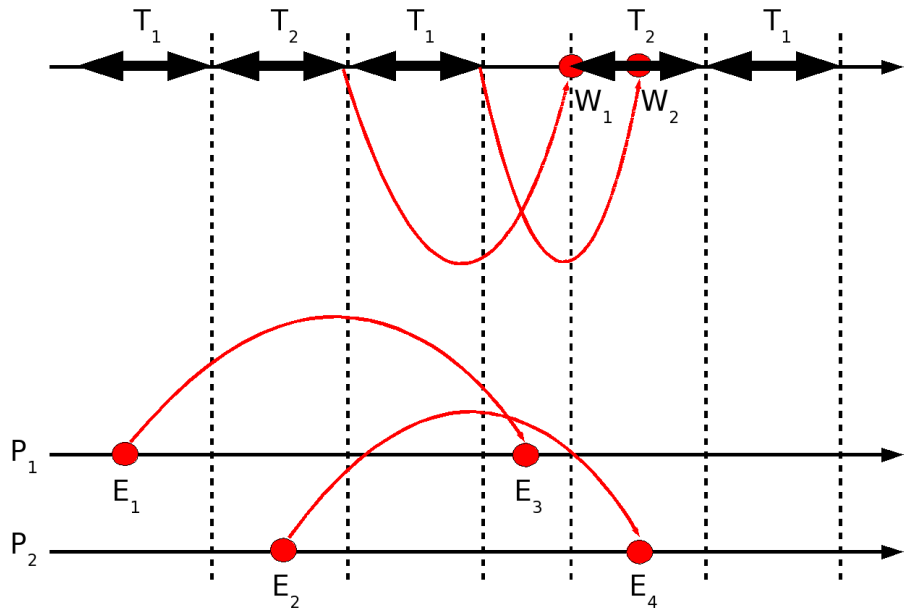

Figure 6: Joint Execution of Classic and JiJ Parts of Simulation

Let's describe what happens during the execution:

1. Thread T_1 gets control and runs until its next consistent state. When the consistent state is reached, the simulation time is updated.
2. Process P_1 is given control because it has an event in the classic J-Sim calendar with simulation time less than the current simulation time

– E_1. The simulation time is teporarily shifted to the past because classic J-Sim processes must run exactly the same points of simulation time they were scheduled for.

3. Thread T_2 gets control and runs until its next consistent state. When the consistent state is reached, the simulation time is updated. Because the thread calls `sleep()`, a new wake-up event W_1 is inserted to the JiJ calendar and the thread becomes non-runnable.

4. Process P_2 is given control because it has an event in the classic J-Sim calendar with simulation time less than the current simulation time – E_2. Again, the simulation time is teporarily shifted to the past.

5. Thread T_1 gets control and runs until its next consistent state. When the consistent state is reached, the simulation time is updated. Because the thread calls `sleep()`, a new wake-up event W_2 is inserted to the JiJ calendar and the thread becomes non-runnable.

6. There is no runnable thread in the JiJ part of the simulation. The event with least simulation time is E_3. The simulation time is shifted forward and process P_1 gets control.

7. There is still no runnable thread. The event with least simulation time is W_1 which is a wake-up event that has to reactivate thread T_2.The simulation time is shifted forward, the event is interpreted and T_2 becomes runnable.

8. Thread T_2 gets control and runs until its next consistent state. When the consistent state is reached, the simulation time is updated.

9. Process P_2 is given control because it has an event in the classic J-Sim calendar with simulation time less than the current simulation time – E_4 and E_4's time is less than W_2's time. Again, the simulation time is teporarily shifted to the past.

10. Wake-up event W_2 is interpreted and thread T_1 becomes runnable. Because the current simulation time is greater than W_2's time, we know that W_2 was interpreted too late, i.e. T_1 slept for a longer time than it had to. This situation could not be prevented because W_2 could not be interpreted before T_2 got control, i.e. it could not be interpreted "in the future" because there was a runnable thread that had to run "now".

11. Both T_1 and T_2 are runnable so any of them can run. Let's say that T_1 is selected to run...

Wake-up events are stored in the JiJ calendar and sorted by their simulation time. Every wake-up event holds information about the simulation time when the event should be interpreted, the thread to be woken up and the type of the event: sleep-type, wait-type, or join-type.

If there are classic J-Sim events and JiJ wake-up events for the same simulation time and the current value of simulation time is equal to their time, the following priorities apply:

1. First, classic J-Sim events are interpreted (i.e., a J-Sim process is given control) because they do not shift the simulation time to the future.
2. Second, JiJ wake-up events are interpreted, i.e. a `JavaThread` gets runnable but it does not run. Neither this type of event does shift the simulation time to the future.
3. Third, a runnable `JavaThread` is selected and permitted to run until its next consistent state. The simulation time will be adjusted at the end of the step.

These priorities a encoded in the next-step-selection algorithm in `JiJSimulation.step()`.

4.7 `JiJSimulation`

Every JiJ simulation is controlled using a `JiJSimulation` instance that must be created prior to any `JSimProcess` or `JavaThread` creation. It can be seen as a container for both types of objects.

There is just one important method: `step()`. The method performs one simulation step which can be of one of the following types:

- Classic J-Sim step – A `JSimProcess` is run.
- JiJ wake-up step – A `JavaThread` is transferred from non-runnable to runnable state.
- JiJ run step – A `JavaThread` is run.

In any case, the calling thread is suspended during execution of the step, i.e. the method does not return immediately. It returns when the step is really completed – after a `JavaThread` reaches a consistent state or a `JSimProcess` calls `hold()` or `passivate()`. The return value (`true`/`false`) indicates whether there are still threads that can be run and therefore whether it makes sense to call `step()` again.

5 Control Program Verification Procedure

The verification procedure is strongly application dependent, so we can only give some general recommendation 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 (tests) is developed and then executed.

Let us assume a non-stop activity of the overall system. Then the model of environment should issue a (very long) stream of (possibly 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", i.e. the model-time dynamics of the modelled system is not influenced by the observation. The execution can be deterministically repeated as well, even when a random sequence of events is used to stimulate the control program activity[15]. We have two basic possibilities what to follow 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 (objects states) from both the control program and environment parts of the simulation model. As for the timing of invariants evaluation we have two possibilities again: to do it either with every change of the model state (i.e. after every step of the dicrete-time simulation run) or regularly – using a special "Sample and Evaluate" simulation process.
- *Behavioral protocols*, i.e. rules that should apply for a sequence of events (possibly including their timing) at the control interface[16].

Clearly, the problem of the performed tests number and length (diagnostic coverage, completeness) stays open – similarly as when we are testing the program at a real device. But here we have extended possibilities of the tests organization (e.g. the possibility to use a "wild" environment activity or to change randomly the control program timing in order to reveal race conditions). The possibility of the model activity observation and investiga-

[15] The used J-Sim's random numbers generators can be started from a given "seed".

[16] It means that the model of control interface should be constructed as a state machine that is able to pass to a "wrong state" when the sequence of events does not correspond to the (checked) behavioral protocol. Generally every object behavior within the simulation model can be checked this way.

tion is better here as well, what means that the number (and the coverage) of the tests performed can be extended in this case.

6 Case Study

To demonstrate the presented method, a case study has been developed. It should be simple enough to be used for the demonstration purpose. On the other side it should not be trivial. We chose an abstract embedded application that controls water level of a water station tank and several connected water sources. See figure 7 for overview.

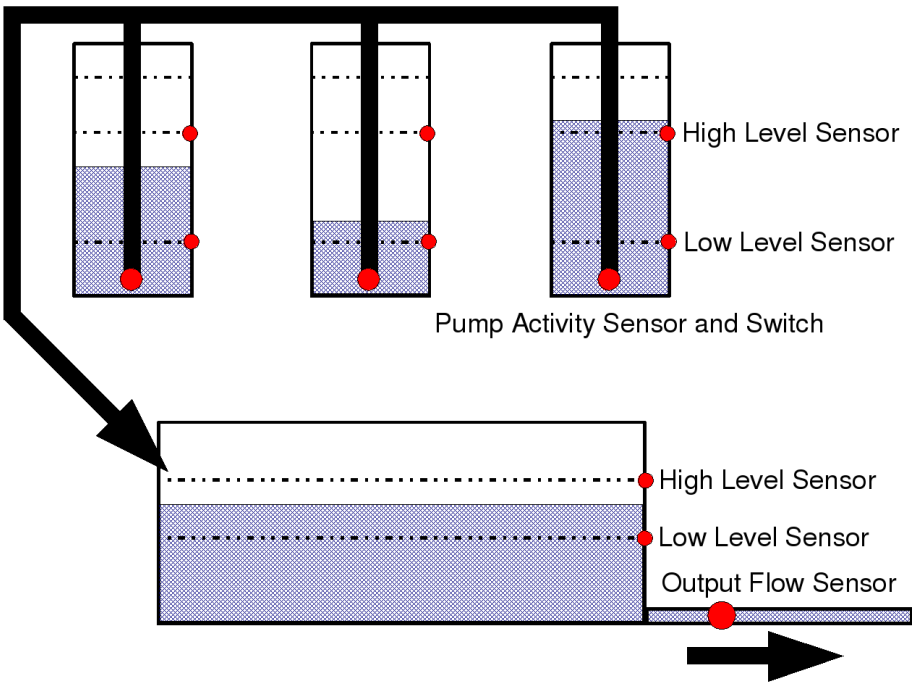

Figure 7: Case Study – Overview of the Controlled Water System

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 signaling low level (as many pumps as possible should be switched on) and the second signalling high level (all the running pumps should be switched off).

Our aim is to keep the tank level between the limits, if possible (i.e. if the capacity of sources is sufficient). We are able to switch on/off the pump placed in each source and we are able 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) unsufficient 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 overpassed. This level is above the 'high level limit' where a sensor is placed.

The demonstration application can be downloaded from the J-Sim web page as a part of the J-Sim distribution archive, directory `CaseStudies\1_WaterSystem`.

The simulation model program code (package `watersystem`[17]) 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.

A modular structure of the source code is depicted in figure 8.

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 to read the sensor values (`boolean`) and to switch on/off the pumps. The interface is implemented within the submodel of control program environment (class `ModelInterface`). In the case of 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 the real control program the interface implementation is a part of its code).

[17] `cz.zcu.fav.kiv.jsimcasestudies.watersystem`, more precisely

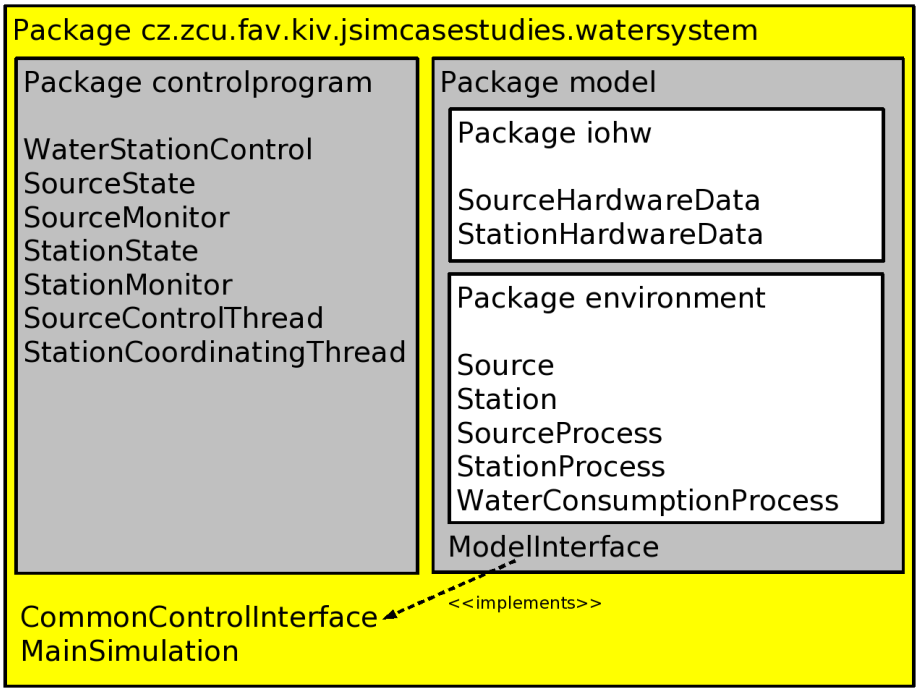

Figure 8: Case Study – Modular Structure of the Water System Simulation Source Code

6.2 Model of Control Program

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

- For each source, there is a monitor object (class `SourceMonitor`) 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 (class `SourceControlThread`). It is an endless loop consisting of periodic sensor data reading, evaluation, and possible reaction. In most cases, there is no reaction. Just in the situation when the level has just got over/below the high/low limit, the pump must be switched on/off. During every loop, the control thread sleeps for a fixed time

interval.[18] There is a hysteresis between the low and high water level, i.e. the pump can be switched on only if the level is above or equal to the high limit and must be switched off when the water level falls below the low limit. Before switching on the pump, using the source monitor object, the thread must pass via a 'semaphore P-function' of the main water tank's monitor, as described below.

- For the main water tank, there is a monitor object (class `SourceMonitor`) that is able to report a consistent state of all its sensors. It also provides two 'semaphore functions P and V' `requestActivity()` and `releaseActivity()`. 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 in `releaseActivity()` when another control thread switches its pump off or in `wakeUpBlockedThreads()` by the station coordinating thread, described below.

- For the main water tank, there is a periodic thread – called station coordinating thread – that reads the high-level sensor data. Whenever it founds that the level in the tank has dropped below the high-level limit, it invokes method `wakeUpBlockedThreads()` of the station monitor. This is necessary because it may happen that all control threads are blocked in the `requestActivity()` method of the monitor because of the tank's water level being too high. The three monitor methods share the same lock – the monitor itself – and therefore it is possible to wake the blocked control threads with a simple `notifyAll()` call.

- Class `WaterStationControl`. An encapsulation of the whole control program part of the simulation. Constructor of the class creates objects (monitors) of water sources as well as the object (monitor) of the water station. It also creates threads responsible for controlling pumps of water sources. Source code of this class needs to be edited when passing it from the model to the production version.

[18] This may lead to a temporary corruption of invariants since the control thread may appear not to be 'quick enough' if the invariants are tested just after an important change occures but before the control thread can react.

6.3 Model of Control Program Environment

The control program environment model part (package `model`[19]) is divided into three (sub)parts. The first part (package `iohw`[20]) contains a model of HW components that are a part of embedded device and that is the control program communicating with (e.g. parallel ports of the assumed microcomputer and the sensors connected to single bits of these ports). The second part (package `environment`[21]) contains a model of processes that is the control system (i.e. control program plus the embedded device HW) interacting with. The third part (class `ModelInterface`) encapsulates all the control program environment part and implements the `CommonControlInterface`.

Principally: the part `environment` communicates with the part `iohw` and this part communicate with `controlprogram` (using implemented functions from `CommonControlInterface`). The parts `controlprogram` and `iohw` together form a model of control system (i.e. the embedded computer device including the connected sensors and actuators). The part `environment` is the model of the controlled environment. When passing from the model to the real-world version of the control system, the part `controlprogram` should not change (or as few as possible) and the part `iohw` should be straightforwardly replaced by real HW devices (the functionality assumed within `iohw` should not change).

The code of `iohw` part contains the following classes:

- Class `StationHardwareData`. It contains an encapsulation of hardware data provided by sensors of the main water station tank, i.e. it is the data model of HW interface that should store the data.
- Class `SourceHardwareData`. It contains an encapsulation of hardware data provided by sensors of a water source, i.e. it is the data model of HW interface that should store the data.

The code of `environment` part contains the following classes:

[19] `cz.zcu.fav.kiv.jsimcasestudies.watersystem.model`, more precisely
[20] `cz.zcu.fav.kiv.jsimcasestudies.watersystem.model.iohw`
[21] `cz.zcu.fav.kiv.jsimcasestudies.watersystem.model.environment`

- Class `Source`. For each water source there is an object holding its current state (level of water) and a reference to sensors data. The same applies to the main tank (class `Station`).
- For each water source, there is a J-Sim process adjusting its water level (class `SourceProcess`). The value of the level depends on the amount of water incoming from outside (modeled as a random number) and the amount of water pumped away if the pump is running (we assume a steady flow). The process also sets the sensors of low/high water level. It cannot, however, switch the pump on/off because this is a function of the control program.
- For the main tank, there is a J-Sim process adjusting its water level (class `StationProcess`). It works similarly to the source process.
- There is one more J-Sim process that simulates the changes of the output flow (class `WaterConsumptionProcess`), i.e. it empties the tank and it sets the output flow register. The function modeling this value should behave reasonably, according to a possible daily consumption of water. But for testing purposes, it can be virtually any random number generator.

The third part of the `model` package (class `ModelInterface`) encapsulates all the control program environment part. It creates the model objects and binds them together. Moreover it implements the `CommonControlInterface`, i.e. it provides methods for reading/writing some data items of the model, especially values of the sensors and the pumps activity flags.

6.4 Overall Model Activity

The main program of the simulation application is contained in the class `MainSimulation` right in the package `watersystem`. It exists merely to create the simulation version of the control interface (instance of `ModelInterface`) and the control program itself (instance of `WaterStationControl`), both of them being parts of a JiJ simulation (instance of `JiJSimulation`).

The simulation is then executed in a step-by-step manner. After every completed step, the basic state of the system is printed out to the console and all invariants are checked. The quantity of information

available to the user depends on the mode in which the application is run; see below.

The source code should be rewritten substantialy when converted to the real-world version of the control system. Actually, the only things that will remain in the production version are the creations of the control interface (instance of `NativeInterface` or whatever name it will be) and the control program.

The simulation of the water station control system has to be started with two parameters:

```
java cz...watersystem.MainSimulation MaxTime RunMode
```

where:

`MaxTime` means the overall time of simulation in "seconds",

`RunMode` prescribes a mode of simulation program activity:

- Value **0** – Time, check of invariants, current output flow and tank volume are printed in every simulation step, check of invariants doesn't influence flow of computation.
- Value **1** – Like 0, but a state of the environment and the control program is printed moreover, an invalid invariant passes computation to a step-by-step mode (RunMode = 2).
- Value **2** – Like 1, but every simulation step is triggered by a key. The key 'Q' means "to finish".

In every step[22], validity of chosen invariants is verified (see function `checkInvariants()`, class `ModelInterface`):

- Invariant no. 1 – Number of running pumps should not exceed the value K.
- Invariant no. 2 – If the tank 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.

[22]One step means to perform a part (an activity) of simulation process that is at the head of the planning list (calendar). Every activity is performed in one model-time point (discrete-time simulation).

- Invariant no. 4 – If the tank volume is below low level and number of sources at the high level is higher than the limit K, then the number of running pumps should equal 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 or equal then the control system response latency – the sleep time of control threads plus some time for code execution.

In the present version of the demonstration application we test only the state invariants of the controlled environment[23]. Generally an arbitrary kind of checks can be performed either in every simulation step or ocassionally. A special simulation process "observer" can be constructed to periodically evaluate the model activity. All the tests are performed (due to the model time concept) without any influencing of the modeled control program activity (i.e. no "probe effect" occurs).

The model runs for a given stationary random process of the tank output flow that models a behavior of water consumers (possibly "wild", when it is this way programmed within `WaterConsumptionProcess`).

[23]State invariant tests are not quite sufficient here due to the built-in hysteresis of the modeled system behavior (e.g. between source level limits the pump can be either passive or active, depending on the "history of pumping"). So we prepare a version with the control interface behavioral protocol checking and reporting.

7 Conclusion and Future Work

The paper presents a method of model based development and (partial) verification of concurrent Java-written control programs that are embedded within a device. The method combines model-time serialization of the control program threads with the conventional discrete-time simulation of the control program environment. Unlike other existing methods, it is an experimental method targetted on testing real Java code[24] and not a theoretical system model. Moreover, the method does not require any special Java virtual machine and is convenient (in contrast to other verification methods) for non-terminating programs as well.

The method takes into account the control program environment behavior and tests both parts of the (designed) control system together which is completely omitted in other program verification approaches.

Further work will be focused mainly on customization of the Java source code conversion tool for purposes of this project and also on the J-Sim JiJ (Java in Java) subpackage improvements and extensions. The set of demonstration applications will be extended in order to gain more experience. We plan to extend this set for distributed (e.g. CAN or TTP/C serial bus based) control applications as well.

Extensions of the JiJ package for Real-Time Java and RMA (Rate Monotonic Analysis) are planned. We currently study their theoretical backgrounds.

[24] That is straightforwardly modified into a model-form and returned back after the performed tests.

References

[1] *Kačer J., Racek S.*: **A Method of Java Concurrent Programs Debugging**. Proceedings of the 5^{th} International Scientific Conference ECI-2002, pp. 80-85. ISBN 80-7099-879-2.

[2] *Kačer J.*: **Java Programs Serialization**. Proceedings of the 5^{th} International Conference ISM-2002, pp. 69-76. ISBN 80-85988-70-4.

[3] *Štika J.*: **Návrh simulační metody verifikace RT programů** (Design of a Simulation Method for Verification of RT Programs). Ph.D. Thesis, University of West Bohemia, 2000.

[4] *Kačer J.*: **Discrete Event Simulations with J-Sim**. Proceedings of the Inaugural Conference on the Principles and Practice of Programming in Java, pp. 13-18. ISBN 0-901519-87-1

[5] *Jacobs B., Piessens F.*: **A π-Calculus Semantics of Java: The Full Definition**. Report CW 355, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, 2003.

[6] *Igarashi A., Kobayashi N.*: **A Generic Type System for the π-Calculus**. ACM SIGPLAN Notices, Proceedings of the 28^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Vol. 36 Issue 3, 2001.

[7] *Magee J.*: **LTSA – Labelled Transition System Analyser**. University of London, Imperial College of Science Technology and Medicine, Department of Computing, London, UK, 1999.

[8] *Godefroid P.*: **Model Checking for Programming Languages using VeriSoft**. Proceedings of the 24^{th} ACM Symposium on Principles of Programming Languages, Paris, France, 1997.

[9] *Holzmann G.J.*: **The Model Checker SPIN**. IEEE Transactions on Software Engineering, Vol. 23, No. 5, 1997.

[10] *Bruening D., Chapin J.*: **Systematic Testing of Multithreaded Programs**. MIT/LCS Technical Memo, LCS-TM-607, Massachusettes Institute of Technology, Cambridge, Massachusettes, USA, 2000.

[11] *Brat G., Havelund K., Park S., Visser W.*: **Java PathFinder – Second Generation of a Java Model Checker**. Proceedings of the Workshop on Advances in Verification, Chicago, Illinois, USA, 2000.

[12] *The Real-Time for Java Expert Group*: **Java Specification Request #1: Real-time Specification for Java**. `http://jcp.org/en/jsr/detail?id=1`

[13] *Grillinger P., Racek S.*: **Transient Faults Robustness Evaluation of Safety Critical System Using Simulation**. Baltic Electronic Conference 2002, Tallin, Estonia, 2002, pp. 257-260, ISBN 9985-59-292-1.

[14] *Herout P., Racek S., Hlavička J.*: **Model-Based Dependability Evaluation Method for TTP/C Applications**. Fourth European Dependable Computing Conference, Toulouse, France, 2002, pp. 271-282, ISBN 3-540-00012-7.

[15] **FIT – Fault Injection for TTA**. `http://www.fit.zcu.cz`

[16] **J-Sim Home Page**. `http://www.j-sim.zcu.cz`

[17] *Kopetz H.*: **Real-Time Systems, Design Principles for Distributed Embedded Applications**. Kluwer Academic Publishers, 1997.

[18] **JavaCC Home Page**. `https://javacc.dev.java.net`

[19] *Herma J.*: **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, Faculty of Applied Sciences, Department of Computer Science and Engineering, Pilsen, Czech Republic, 2004.