Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

David Klika

Checking Primitive Component Behavior

Department of Software Engineering
Supervisor: RNDr. Jan Kofroň
Study Program: Computer Science, Software Systems

Prague, April 2007

First of all I would like to thank my advisor Jan Kofroň for his suggestion and his helpful and friendly attitude.
Not least, I would like to thank my family and friends for their support.

Contents

Název práce: Checking Primitive Component Behavior
Autor: David Klika
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jan Kofroň
E-mail vedoucího: jan.kofron@dsrg.mff.cuni.cz
Abstrakt: Software model checking je metoda ověřování vlastností programů a tedy zajišťování jejich vyšší spolehlivosti. Je však stále nutné dělit programy na části ověřované nezávisle, neboť úplné programy tvoří často příliš velký stavový prostor, který není možné prozkoumat v rozumném čase. Softwarové komponenty dělí aplikace na části vhodným způsobem. Ale vzhledem k tomu, jak fungují model checkery, je nutné jim poskytnout vhodné prostředí pro ověření každé komponenty.

Pro popis chování komponent vznikly behavior protokoly [1]. Umožňují ověřovat kompatibilitu a shodu chování komponent — to se používá již při návrhu aplikací pro odhalení možných nevhodných použití komponent. Protokoly jsou také vhodné jako popis chování komponenty, který může být srovnán s chováním konkrétní implementace komponenty.

Cílem této práce je poskytnout nástroj pro srovnání chování primitivní komponenty oproti její specifikaci. V rámci toho bude implementován generátor prostředí komponent podle frame protokolu, které umožní rozpoznat porušení specifikace chování. Bude použit Java PathFinder [2] a Fractal component model [3].
Klíčová slova: software components, behavior protocols, model checking

Title: Checking Primitive Component Behavior
Author: David Klika
Department: Department of Software Engineering
Supervisor: RNDr. Jan Kofroň
Supervisor's e-mail address: jan.kofron@dsrg.mff.cuni.cz
Abstract: Software model checking is a process of checking for properties of a software application and thus assuring the software reliability. It is still necessary to divide the software application into pieces that are checked separately; the whole application yields an enormous state space that is impossible to traverse in a reasonable time. Therefore, the use of components is a straightforward approach for dividing the entire application. Furthermore, as model checker usually works with a close code only, a suitable component environment need to be provided for each component.

Behavior protocols [1] are a method for component behavior specification. They allow for checking for components' behavior compatibility and compliance, used at the design time of an application to find possible architectural component misplacements. They are also suitable to be compared with the behavior of the component implementation.

The goal of the thesis is to provide a tool for comparing a primitive component behavior with its specification. Furthermore, a component environment generator using the component frame protocol will be implemented that would enable for checking for violation of the component behavior specification. The Java PathFinder model checker [2] and the Fractal component model [3] are to be used as a model checking platform.
Keywords: software components, behavior protocols, model checking

Chapter 1

Introduction

Software component is one of the most important terms used in software engineering today. Component-based software engineering [14] is based on building software systems from functional or logical components with well-defined contracts. It is focused on reusability of such building blocks, which leads to cost effective solutions. Component-based applications can be easily maintained and extended. Software components satisfy most of requirements posed on today's software systems.

Behavior protocols [1] were proposed to extend the contract by specification of behavior of each particular component in the system. This specification is primarily designed for verification of correctness of component systems.

Correctness of a component software system can be stated at several levels. Early at design time, it is possible to verify correctness of the architecture of a component system — behavior protocols of connected components have to conform each to other. This issue is well described and solved by using static analysis [8]. However, this analysis does not make any statement about an implementation of the designed system. To fill this gap, it must be verified that component implementations adhere their behavior specifications. This issue was not solved satisfactorily yet.

Designing a software system using components with proved behavior moves its quality and reusability of its parts to a new level.

This thesis tries to answer the problem of verification whether a behavior specification is conformed by the component implementation. The model checking technique is engaged to this process to verify any possible behavior of the component. The task is quite complex and ambitious; however, its results are expected and widely applicable.

1.1 Related problems

This section presents a brief introduction to problems related to the task of this thesis.

Software components

A software component is a black-box unit that offers some services. Such units are connected together to build-up whole software systems.

Components are able to communicate with other components through well-defined channels. These channels are *interfaces* described in a programming or a special-purpose (interface definition) language. Interfaces are *provided* representing services offered by the component, and *required*, which represent services required by the component to support the component operations.

Many component frameworks were developed, some of them are mentioned in chapter 6. This thesis is focused on Fractal component model [3].

The Fractal component model introduces hierarchical components. Components are *primitive* — implemented directly in a programming language, and *composite* — implemented by composition of other components. Composite components were introduced to provide a uniform view of applications at various levels of abstraction.

Behavior protocols

Behavior protocols [1] enhance architecture of component systems by description of component behavior. Components are described not only by provided and required interface definitions but also by behavior specification.

Behavior protocols describe all correct sequences of method calls and returns occurred on public interfaces (provided and required). A *frame* behavior protocol describes behavior of a component with respect to components on the same level. *Architecture* behavior protocols are focused on communication of first level of nesting subcomponents of a composite component.

Correctness of component systems

After behavior of each component is specified, the problem of correctness of component connections arises [8]. Behavior protocol of a component has to be conformed with protocols of components that are connected to it.

The problem is stated at several levels. Behavior *composition* test verifies conformance of behavior specifications of components on the same level of nesting. Behavior *compliance* test verifies behavior specification of a composite component with respect to behavior specifications of its sub-components.

These tests are performed just according to architecture and behavior description, no implementation is employed in this process. This aspect allows performing these checks in early stages of the system development; so, serious design errors may be eliminated very early. However, to ensure that an implementation of the designed system is correct, behavior specifications of primitive components have to be confronted with their implementations.

1.2 Problem definition

Problem addressed by this thesis is comparison of a behavior specification and a primitive component implementation. It is the final step to verify correctness of the whole component system when the other steps of the system verification are satisfactorily solved.

Two independent tools, which tries to solve this problem, were developed. However, their applicability is not fully satisfactory.

The Software Component Model Checker [5] is a well proved tool that solves the discussed problem. However, this solution has several drawbacks.

- The tool requires non-trivial additional code (component environment) to be verified with a checked component. That results in substantial time and memory requirements. In consequence, behavior protocols have to be often simplified to make the verification process feasible.
- Moreover, usage of alternative, parallel and iteration operator is limited. So, the solution is not applicable on some kinds of behavior protocols. Alternatively, such protocols may be modified by hand first to avoid unsupported situations. Consequently, some behavior errors may be omitted, when the component is verified against a modified protocol.

The Carmen project [6] had the ambition to outperform the previous solution in most aspects. The proposed concept of the solution seemed to be promising with only one obvious drawback, which had been accepted — the solution is derived from an existing model checker tool, so, an independent fork of the tool was made.

A prototype implementation was developed successfully; also presented results seemed to be persuasive. However, several problems and drawbacks were hidden.

- The prototype implementation is restricted just to primitive data types, thus all real-life examples have to be adapted and simplified to this limitation.
- Presented results were done on examples which were simplified too much and the value of such results is not sufficient.
- Moreover, the tool is not able to detect some obvious behavior violations, so, the price for the performance of the tool is indefensible.
- Also several other aspects, such as false statistics, are present.

A common drawback of both tools is that they do not provide any support for checking sufficiency of the behavior verification process.

1.3 Goals

As presented in sections above, verification of behavior of primitive component implementations is necessary to prove correctness of component-based software systems. However, the previous section demonstrated that a usable tool for this task is still missing. Thus, this thesis laid out the following goals.

- G1: Propose, design and implement a tool for verification of a primitive component behavior. The tool has to be able to perform exhaustive verification, however, the complexity and completeness of the verification may be affected by some parameters. The tool should be applied to real life instances, so, the time and memory requirements of the tool should be reasonable without any modifications of inputs of the task. Avoiding modifications of supporting tools is preferred.
- G2: Provide additional tools for analysing the process of verification of a primitive component behavior to observe the quality or deepness of the verification. For example, coverage of code of the component or coverage of states or traces of the behavior protocol should be reported to identify diversity between the behavior protocol and the component behavior or to identify insufficient range of input values.

1.4 Structure of the thesis

This chapter presents a brief introduction and definition of the problem. Goals of the thesis are presented here.

Chapter 2 gives more detailed description of the problem and also some background information about the Fractal component framework and Java PathFinder model checker. A part of this chapter is a description of practical experiences with using the Java PathFinder tool.

Chapter 3 discusses several approaches which address the primitive component behavior verification problem. The most suitable solution is selected and analysed in more details.

Chapter 4 describes the proposed solution in details. This description is sufficient for implementing the solution.

Chapter 5 presents a complex report on performance of the implementation. It is also confronted with other projects.

Chapter 6 mentions related work.

Chapter 7 concludes the thesis.

User documentation of the implementation is contained in appendices. The implementation source codes, binaries, additional documentation and some examples are stored on a supporting CD.

Chapter 2

Background

The goal of this chapter is to give a more detailed description of the problem and also some background information about the Fractal component model, behavior protocols, and Java PathFinder model checker. Last part of this chapter presents practical experiences with using the Java PathFinder tool.

2.1 Fractal component model

Fractal [3] is a modular and extensible component model. It is focused on designing, implementing, deploying, and reconfiguring complex software systems. Fractal aims to be applicable to wide range of software system types — from embedded software to application servers and information systems.

The specification of the Fractal component model introduces a hierarchical component model, where components are specified by their provided (server) and required (client) interfaces and configurable attributes. The model also supports advanced features such as introspection and intercession capabilities, component sharing, optional interfaces, collection interfaces and other.

A reference implementation of Fractal developed in Java is provided, it is called Julia. Another related project is the Fractal ADL project. It defines an XML-based architecture description language for the Fractal component model. It specifies the initial architecture of an application, values of attributes of components and other.

2.2 Behavior protocols

Behavior protocols [1] specify behavior of software components. Behavior of a component is characterised by communication on its public interfaces. This communication is a sequence of (atomic) *events* from point of view of behavior protocols. Used types of events and their meaning is listed in the table 2.1.

Formally, the behavior protocol is a regular-like expression composed of event tokens, operators, and parentheses. Behavior operators and their meaning is listed in table 2.2.

*!intf.meth^*	Emitting invocation of method
*?intf.meth^*	Accepting invocation of method
!intf.meth$	Emitting response
?intf.meth$	Accepting response

Table 2.1: Behavior protocol event types for method *meth* on interface *intf*.

Operator	Arity	Name	Meaning
;	binary	Sequence	*a;b* means that *a* is followed by *b*
+	binary	Alternative	*a+b* means that either *a* or *b* is performed
*	unary	Iteration	*a** means that *a* is repeated zero to finite number of times
|	binary	And-parallel	*a|b* represents all interleaving of sequences generated by *a* and *b*

Table 2.2: Behavior protocol operators.

Moreover, behavior protocols define several abbreviations, which are listed in table 2.3. The and-parallel operator can be also expressed by using other behavior operators, therefore, it is an abbreviation.

Abbreviation	Stands for
!intf.meth	*!intf.meth^ ; ?intf.meth$*
?intf.meth	*?intf.meth^ ; !intf.meth$*
!intf.meth{expr}	*!intf.meth^ ; expr ; ?intf.meth$*
?intf.meth{expr}	*?intf.meth^ ; expr ; !intf.meth$*

Table 2.3: Behavior protocol abbreviations.

The term *trace* represents a particular finite sequence of events. Every behavior protocol specifies a finite or infinite set of traces. Therefore, every behavior protocol specifies a (regular) language — regular operators are used by the behavior protocol grammar only. Alternatively this language can be specified by a finite-state automaton. States or state space of a protocol refer to states or state space of the appropriate automaton.

2.3 Checking component system corectness

Correctness of a component system can be stated at several levels [8].

Behavior composition test verifies conformance of behavior specifications of components on the same level.

Behavior compliance test verifies behavior specification of a composite component with respect to behavior specifications of its sub-components. This test uses the consent operator, which defines behavior of the composition of subcomponent frame protocols.

Run-time checking inhere in monitoring events occurred on public interfaces of a component and checking whether this acquired sequence of events (trace) is allowed by the behavior protocol. Run-time checking is not an exhaustive verification of behavior of the system, however, it is applicable to composite as well as primitive components. It is also the only way to check behavior of dynamic architectures, which cannot be verified statically.

Code analysis of a primitive component implementation verifies, whether the component adheres its behavior protocol — whether the component can accept and emit events only in sequences that are determined by its behavior protocol.

2.3.1 Behavior protocol checker

Behavior protocol checker (BPChecker, BPC) is a piece software, which is able to perform composition and compliance test of behavior protocols, as well as trace vs. behavior protocol test. In case of code analysis, backtracking of the BPChecker state may be a helpful feature.

This thesis is focused on code analysis, therefore performing compliance and composition tests is not required.

2.3.2 Possible behavior violations

Several types of composition errors may occur when building component systems with behavior specifications.

A *bad activity* occurs when the component A tries to call a method of the component B when it is not expected by the behavior protocol of the component B.

No activity (or deadlock) occurs when no component is allowed to emit an event, and at least one of the components has not finished its computation.

An *infinite activity* (divergence) occurs when computation of a component system cannot stop, but components are not blocked.

In connection with optional interfaces, an *unbound requires error* may occur. It occurs when a component tries to call a method on its required interface which is unbound.

More detailed definitions and description is in [8] and [13].

2.4 Model checking

Model checking [15] is the process of checking whether a given model satisfies a given property. The model is often derived from a hardware or software design and described in a special-purpose language. The property is usually expressed by a temporal logic formula (LTL, CTL).

Model checker tools explore all states in which the verified model can be — state space of the model — and reason that the property is satisfied or find a counter example showing the violation of the property.

Explored states may be represented explicitly or implicitly by the model checker. Explicit representation works usually faster, however, memory requirements are higher.

In case of software systems, the model checker may fail to prove or disprove a property because of undecidability. State space of the model may be unbounded.

2.5 Java PathFinder

Java PathFinder (JPF) [2] is an explicit software model checker developed by NASA Ames Research Center [9]. JPF checks models described by Java byte code — JPF acts as a Java Virtual Machine (JVM, VM). However, the code is executed not just once, JPF systematically explores all potential execution paths to find violations of properties. The checked properties are deadlocks and uncaught exceptions. Therefore, JPF is able to check any condition that can be expressed in the Java code.

JPF employs a lot of heuristics and optimizations. It is also highly configurable and extensible. Thanks to this, JPF may be viewed as a framework for Java bytecode verification techniques.

The rest of this section presents some fundamental aspects of the JPF design; also some practical experiences and recommendation are discussed.

2.5.1 Java PathFinder basics

This section presents fundamental information about Java PathFinder. They are gathered from the JPF documentation [2], where more detailed description is presented. Also some more advanced topics (state abstraction, heuristic choice generators, search strategies) and examples can be found there. This section is focused on topics which are required for understanding following chapters.

Backtracking

As aforementioned, JPF explores all potential execution paths reachable in the verified system. To avoid re-executing the program from the beginning, the backtracking technique is employed. JPF is able to revert state of the verified system to any previous state, where a desicion on further exploration has been made. Execution may continue from this point along another path. Efficient state management is required to support the backtracking technique.

State matching

State matching is another key mechanism to reduce unnecessary work. While JPF executes it checks, whether its state is the same as another previously visited state (states are matched by means of heap and thread-stack snapshots). In this case, it is no reason to continue execution along this path and JPF backtracks to explore another non-explored path.

On-the-fly partial order reduction

In case of concurrent programs, the number of different scheduling combinations is the prevalent factor for complexity of the verification process. The partial order reduction (POR) is the most important mechanism to reduce this complexity. JPF employs on-the-fly POR that does not rely on user error-prone instrumentation or static analysis.

The goal is to skip context switches which cannot have effect across thread boundaries. Sequences of instructions executed by a particular thread are grouped to transactions. Only such transactions of different threads are interleaved.

Host VM execution (MJI)

JPF is able to delegate execution of some code to the host JVM. This method is called Model Java Interface (MJI). The host VM is independent of the state-tracking, backtracking and other JPF techniques. The delegation makes sense for code that is not relevant to verified properties. It is designed for executing I/O operations and other standard libraries, however, this technique may help to decrease size of the model and to reduce the JPF state space.

Search- /VMListeners

It is a basic technique to extend the JPF functionality. Listeners are custom objects that register themselves with JPF and then they are notified when JPF performs certain operations. They can interact or even control the JPF behavior.

Search listeners are related to exploration of the JPF state space. VM listeners monitor VM processing — execution of particular instructions, scheduling threads, synchronization, garbage collection and others.

JPF model API

The JPF model has direct access to a subset of the JPF API[1]. It contains methods for generating non-deterministic data; configurable heuristic choice generators are implemented. Other methods anotate JPF states. It is also possible to cancel exploration of JPF search paths — the JPF search graph can be pruned by the model itself.

2.5.2 Java PathFinder experiences

This section describes practical experiences and recommendation with using the Java PathFinder model checker. JPF, as a runtime environment, acts in a very different way in comparison with standard JVM in several aspects. Therefore, design of the verified software systems should be adapted for this runtime environment.

[1]The model API is centralized in the gov.nasa.jpf.jvm.Verify class.

Most of the following experiences were identified when the most suitable form of the component environment was designed. However they are applicable in general on any piece of software that is verifyed by JPF — including primitive components. On the other hand, any hand-made modification of the verified system may potentially introduce or hide errors.

Speed of the JPF VM

Java PathFinder acts as a Java virtual machine. However, execution of code in the JPF virtual machine is significantly slower than execution in a common VM even if no non-determinism, backtracking, and multiple-execution occurs. The reason is that JPF has to interpret each step by many operations that are performed in another JVM.

Therefore, complex computations, especially when they do not depend on non-determinism, should be pushed out of the JPF model (for example by using native methods) to be executed faster.

Variables inside the JPF model

Several aspects should be considered in connection with usage of variables inside the JPF model.

First, each JPF model variable increases requirements for representing each state of JPF — JPF represents states explicitly; tens of millions of states may be managed by JPF. For example large constant or immutable data (e.g. fixed user database) can be pushed out of the JPF model by abstracting some data structure.

Moreover, volatile variables causes the state maching heuristic less efficient. For example a simple counter, which counts number of a method call, may cause the verified system uncheckable. Every volatile variable should be considered, whether it is relevant for corectness of the verified system. If not, it should be removed or pushed out of the JPF model. For example Verify.incrementCounter() or other (custom) native methods can be used to do this.

On the other hand, volatile variables, that depends on values of other variables in the JPF model, do not make any troubles to JPF. Consider the following example:

Source code listing

```
for (int i = 0; i < 20; i++) {
   int ii = i*i;
   ...
```

Introduction of the ii variable does not increase total number of JPF states — variable i generates 20 states, variable ii generates also 20 states, however, these states are the same. This statement holds if changes of dependant variable values are done inside one transaction — if POR heuristic does not trigger creating new states meanwhile.

Finally, data structures should be always in a consolidated and uniquely determined state. A lot of data structures save time by releasing rules or by some "lazy" algorithms. This results to situations, that are not suitable for the state matching heuristic. A particular content of the data structure may be represented by many different ways. Consider for example number of distinct binary search trees that represents the same set of elements.

Data structures should be chosen with respect to the state matching heuristic. They should be consolidated after each operation and their memory footprints should be uniquely determined by their content. This does not hold, of course, if the object of verification is the particular data structure.

Managing threads

Thread management is critical for performance of multi-threaded systems that are verified by JPF. Number of traces verified by JPF grows exponentially with number of threads. The state matching heuristic helps to reduce most of them. However, to keep the state matching heuristic working, a proper thread management must be applied.

Consider applications which create threads dynamically — http and other servers, it is also the case of asynchronous requests to a primitive component by its environment. Creating threads dynamically is quite inefficient in JPF VM, because each thread gets an unique name, which is relevant for the state matching heuristic.

If maximal number of used threads is known, a thread pool with fixed size may be used. This technique may be optimized by symmetry reduction based on permuting threads (discussed in 5.2.1). Another solution of this issue is presented in [6], which is based on modification of JPF and reusing thread identifications.

Also communication between threads requires a special care. Usage of monitors, synchronized blocks and methods `wait()`, `notify()` and `notifyAll()` is the most suitable thread synchronization mechanism in JPF VM. Alternative methods (e.g. spinlocks) are possible but not efficient enough.

Verified system has to be free of dependencies on thread scheduling. JPF tries any possible schedule of threads. For example, if a thread waits in a loop for activities of another thread, it may never stop. Situations which lead to thread starvation in common VM have to be eliminated.

Using Model Java Interface

The MJI is often the right way to optimize complex software systems and adapt them to JPF requirements. However, usage of it may influence new issues that have to be solved.

First, code of native methods does not have direct access to object type parameters. Only a reference is known, which can be used for some limited access to the object member data. However, more advanced operations are not supported. The same limitations holds for accessing other data structures inside the JPF model.

Moreover, data which were pushed out of the JPF model with some code may be relevant for corecntess of the verified system. In this case, they (or a code/hash of it) have to be propagated back to the model by the native methods. Progress of native methods execution is invisible to the POR heuristic — a lot of changes of the JPF model data may be done in a single transaction.

Usage of MJI may be motivated by determining value of a global property (in any state of the verified system) or by supporting the verification process (some reason for this were presented above). In the second case, backtracks of JPF have to be reflected in data structures in the host VM. To achieve this, data can be restored from the JPF model, if they are relevant to corectness of the verified system. If they are not, search listener mechanism can be used to associate content of data strucutres with the JPF state and to detect backtracks. Consider memory requirements when using this pattern.

Chapter 3

Concept and Analysis

This chapter contains two parts. The first one is focused on choosing the most suitable concept for the task of primitive component behavior verification (G1). A few general observations are presented; above all, several possible concepts are discussed and a comparison of their characteristics is presented. The most suitable approach is selected.

The second part introduces a basic analysis of the selected concept. It is focused especially on the problem decomposition and requirements specifying; business processes are identified and described. Also the goal G2 is elaborated in this part in more detail.

3.1 Concept of the work

To verify behavior of a primitive component against its behavior specification (frame protocol), it is necessary to check behavior of the component in each situation allowed by the protocol.

This task is very close to the model checking task — see section 2.4. However, the behavior specification (the behavior protocol) is at a higher level of abstraction compared to properties that common model checkers are designed to verify. Nevertheless, the most suitable approach for the primitive component verification problem is using some of the existing model checkers. It is the only way to avoid many serious issues that are related to the model checking task in general.

To apply this approach, the following issues have to be solved:

- Software system incompleteness — the verified component itself is not a complete program that can be checked by a typical model checker — at least an entry point is missing.
- Missing environment — the component is not connected to other objects through provided and required interfaces, which are required for communication. The component also supposes to be requested through these interfaces by the environment in any possible correct way.
- Specification of properties to be checked — the high level property (the behavior protocol) has to be transformed into properties that model checkers are able to

verify (assertions in particular).

There is no a straightforward way to solve the last issue. The reason is that such assertions depend on the current state of the verified system. A more suitable way is to involve a behavior protocol checker in the checking process. The behavior protocol checker can manage its state by observing the checking process (in particular events emitted on public interfaces of the component). It can also check whether the behavior of the system is correct according to the behavior specification or not.

There are several concepts that conform to all ideas above; however, they differ in many aspects. Description of them is presented in the following sections. The most suitable solution is chosen and discussed in the rest of this thesis.

A common problem of all solutions is the value specification problem — provided and required methods may require parameter and return values in general. They have to be specified when the component is requested. This problem is caused inherently by the quite high level of abstraction of behavior protocols. Therefore, this issue is not a criterion for comparing presented concepts.

Note

The Java PathFinder model checker version 4.x is chosen for this thesis, some of its features or limitations may be referred in the following text.

3.1.1 Protocol driven environment

According to this concept, an environment is generated. The environment is specific for a particular checked component. The architecture of this concept is outlined in the figure 3.1.

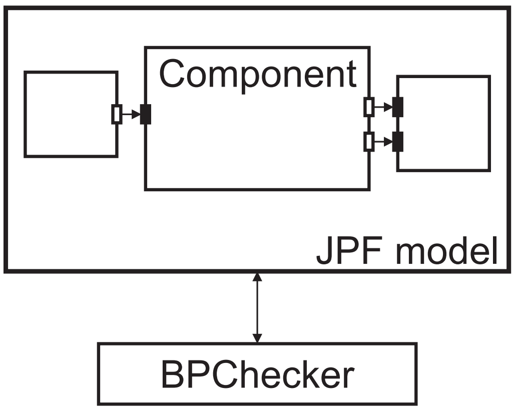

Figure 3.1: Architecture of the protocol driven environment concept.

The environment and the component form together a complete program that can be verified by a model checker. The environment should be connected to the checked component to enable communication on provided and required interfaces of the component. The environment should also be able to emit any correct sequence of events — requests to the checked component.

The environment generation is based (among other aspects) on the behavior protocol; then the environment is driven by it. The idea of generating Java code from a behavior protocol is described by the table 3.1. The generation is focused primarily on active events of the environment. However, interleaving active and passive events in the protocol may introduce some issues.

A ; B	A; B;
A + B	if (Verify.randomBool()) A; else B;
*A**	while (Verify.randomBool()) A;

A | B	new Thread() { public void run() { A; } }.start();
	new Thread() { public void run() { B; } }.start();

Table 3.1: Generating Java code from a behavior protocol.

Another part of the concept, independent from the environment, is the behavior protocol checker. It observes events emitted by the whole system and checks that all sequences occurred are correct according to the behavior protocol. It is possible to exclude this part from the JPF model space to improve performance of such solution; however, some additional issues have to be solved then. For example, behavior protocol checker has to reflect JPF backtracks and the state of the behavior protocol checker has to be respected by the JPF state matching heuristic.

This concept is described in more detail in [5].

Evaluation

The main advantage of this concept is the adaptation to specific model checker needs; the concept does not require any substantial modifications of JPF.

On the other hand, the protocol driven environment may overload the JPF model and cause the performance and memory requirements of the solution to be unacceptable. Some modifications of JPF are required, when the behavior protocol checker is excluded from the JPF model. The usage of such a solution may be restricted to a subset of behavior protocols. After all, an environment generation required for each component may be difficult.

3.1.2 Java PathFinder modification

This concept [6] is based on adapting the JPF model checker instead of the model itself (the verified system) to the task of primitive component behavior verification. No environment is generated; the model consists just of a checked component. The modified model checker should be flexible and reusable for any component.

An overview of the concept is shown in figure 3.2. Only a checked component is inside the JPF model. Requests on public interfaces, which are triggered by the checked component, are caught by JPF modifications; also requests to the component come through them. The dotted round pattern is used to symbolize some of the JPF modifications that are required by this concept.

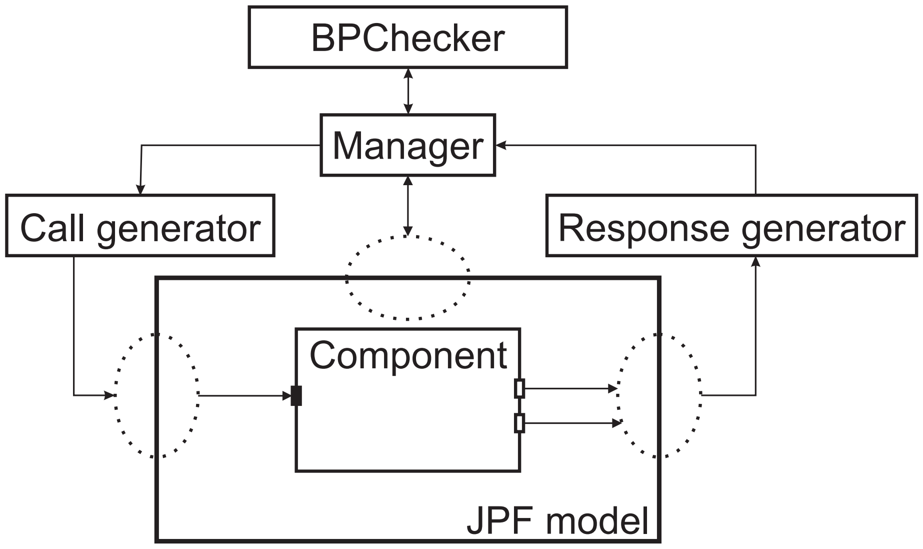

Figure 3.2: Architecture of the JPF modification concept.

The Manager is a control unit, it acts as the component environment. Moreover, it manages the whole process of component verification. The manager communicates with the behavior protocol checker and simulates every correct behavior of the component. The manager and also the BPChecker have to implement the backtracking technique in a way synchronized with the JPF model.

Evaluation

This concept solves the problem of missing environment in a generic way. Even if it does not require any pre-processing steps, the most considerable benefit of this solution is the reduction of state space of JPF — only the code of the component is traversed.

On the other hand, a lot of modifications of JPF are required by the concept. JPF is a very complex system, which is developing continuously. Some modifications of the system may affect not only its functionality but also its performance. A lot of the required modifications can also be passed by using more suitable techniques. Other issues are specific to this concept only — i.e. implementing object types on signatures of provided and required methods may be quite difficult.

However, the most important disadvantage of the solution is that a new model checker is built up, which is independent of the original JPF. So, it may be very difficult to keep up with the development of the original JPF model checker.

Another attribute is common to all generic solutions — the value specification problem cannot be solved by using an application-specific logic without touching the code of the solution itself.

3.1.3 BPChecker driven environment

This concept prioritizes two challenges — avoiding any JPF modification and the performance, of course. The architecture of the concept is shown in figure 3.3.

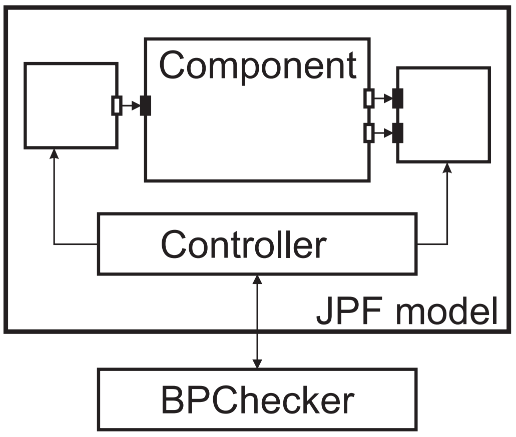

Figure 3.3: Architecture of the BPChecker driven environment concept.

The concept uses a very light-weighted environment that resides in the JPF model. The responsibility of the environment is just making requests to the checked component and gathering events occurred. All other activities lie on the behavior protocol checker that is pushed out of the JPF model. The environment and the behavior protocol checker communicates together synchronously using MJI (see section 2.5).

The main difference to the protocol driven environment is that most operations of the environment is done out of the JPF model by the BPChecker. This idea is close to the idea of the JPF modification concept; however, the way of realization is completely different.

The concept can be used for generated as well as generic environment. Due to performance reasons, generated environment is preferred. So far, the JPF model checker does not support dynamic proxy objects, which are required to implement a generic environment.

Evaluation

This concept avoids any JPF modifications; some problems related to modifying JPF are summarized in the previous concept (3.1.2). This concept introduces an environment that does not generate any unnecessary states in JPF; thus, the performance of the solution should be satisfactory. The concept is clear and easy to understand as well as to implement.

An environment generation is used for now, which may be annoying. However, it is not required by the concept itself. In addition, the necessity of an environment generation is compensated by the potential of possible optimizations. All other potential disadvantages, which were discussed in context of other approachs, are eliminated by this concept.

3.1.4 Concept selection

There were presented some reasonable approachs to the primitive component behavior verification problem in previous sections.

Considering carefully the advantages and disadvantages of each solution, the BPChecker driven environment concept seems to be the most suitable one. It introduces a very efficient solution without any unnecessary effort. It is also applicable to any incoming release of the JPF model checker with no or trivial changes. The main drawback of it — the necessity of environment generation, may be eliminated in future without any key changes of the concept.

3.2 Analysis of the concept

An analysis of the selected concept is presented in this section (G1). It is focused on splitting the problem into smaller and independent parts and especially on specifying requirements for them. Also some conformance tests are introduced in this section to satisfy the goal G2.

The architecture of the concept is shown in figure 3.3. There are several parts of the concept, which are described in the following sections.

3.2.1 Environment of a primitive component

The Java PathFinder model checker requires a complete program to perform the checking. To satisfy this, it is enough to put a main method (for example an empty main method) to the JPF model. However, it is not enough for verifying the behavior of a primitive component.

Above all, an instance of the primitive component class has to be created and then combined with its environment. The environment consists of driver and stub objects at least. The driver objects are bound to the provided interfaces, while stub objects to required interfaces of the component.

Note

'Required interfaces' and 'provided interfaces' point to interface names, not types. It is allowed to use one interface type several times in a single component.

To verify behavior of a component, the environment has to:

- Call any provided method asynchronously whenever the call is expected.
- Return a required method call whenever the return is expected.
- Do the final-state check when no more activities are done.

To know what events are expected and to perform the final-state check, the environment uses the behavior protocol checker. It is described in the following section.

3.2.2 Behavior protocol checker

Behavior protocol checker (BPC, BPChecker) is introduced in section 2.3.1. Its main objective is to check whether the sequence of events occurred is allowed by a behavior protocol.

BPChecker is driven by the behavior protocol of a checked component. It holds its internal state that corresponds to the sequence of events absorbed by it (event trace). These events are provided and required method calls and returns. An event is absorbed by the BPChecker just when it is expected. If it is not, a bad activity of the component is reported (see 2.3.2 for possible behavior errors). The BPC is also able to differentiate whether its current state is one of the final states or not.

BPC and environment cooperation

The environment of a component communicates with the behavior protocol checker synchronously. The BPChecker has to be notified about events occurred on public interfaces of the component to be able to manage its state. The checker should also publish all events that are expected in the next step to the environment.

3.2.3 Environment generator

It is obvious that the environment described in the section 3.2.1 depends on the checked component, on provided and required interface sets, on interface type definitions, and on other inputs (e.g. value sets for function arguments and return values). The environment can be

- generic — reusable for any primitive component; a lot of proxy objects and reflection have to be engaged to implement this, or
- specific for given inputs — it may be generated in an automatized way; also some important optimizations can be applied during the environment generation phase when all inputs are already known.

Due to performance reasons, the second approach was chosen. Consequently, an environment generation tool with these characteristics has to be provided.

3.2.4 Value specification problem

In general, the environment of a component has to specify values of provided method parameters and return values in case of required methods. The problem lies in missing specification of these values. Behavior protocols are not interested in it; the level of abstraction of the behavior protocols is quite high.

The problem can be solved by introducing a new specification that is expected as one of inputs of the task. The specification contains a set of possible values for each method parameter or a return value of a function. Every element of the appropriate set is tested by JPF on every request to the component during the checking process.

However, there is no opportunity for using any application-specific rules for determining parameter or return values. To avoid introducing a new, complex language for describing such rules, pieces of Java code can be placed into the generated driver and stub classes to implement application-specific rules, if it is required.

3.2.5 Conformance tests

This section introduces some conformance tests, which may help to describe the quality of the checking process and also the conformance to expectations of the test designer. These tests were introduced to satisfy the goal G2.

Note

The term *test designer* points to a person who is responsible for the verification of the comopnent behavior.

Code coverage test

The motivation for this test is the value specification problem, which is described in section 3.2.4 — parameter and return value sets have to be specified by the test designer. These sets have to be small enough to avoid overloading JPF; however, there is no feedback, whether these sets were sufficient for exposing the component to all possible situations or not. If there is a gap, it may be signalized by the code of the component which was never executed during the checking process. The goal of this test is to identify such a code and report it to the user. Consider the following example.

Behavior protocol fragment: *?a.userLogged{!b.getUserAge + NULL}*
Fragment of the component implementation code:

Source code listing

```
public void userLogged(User user) {
  if (user.isMale())
    userDatabase.getUserAge(user);
   ...
```

Supposed the third line is never executed during the checking process, the test designer should upgrade the set of values for the `User` parameter by some object of male type.

The ability of reporting code coverage does not depend on type of checked application (e.g. a primitive component in our case). It can be of a general purpose and a reusable piece of code or a plug-in for JPF. However, the primitive component checking has some specificity. A primitive component can/has to implement some interfaces defined by the component framework but they are not used by the environment. This case should be detected and reported in a different way.

The reporting tool has to spy code that is executed by JPF. The set of analyzed classes should be configurable to distinguish the component implementation code (business logic) and the other supporting code (the environment, Java and other libraries, etc.). The missed code should be highlighted in the source code to be illustrative.

BPChecker states/transitions coverage test

This test proves the environment and also the component, whether all situations allowed by the behavior protocol were visited by them.

From the point of view of the environment, the test vindicates the environment itself and parameters of the verification.

From the point of view of the checked component, the test demonstrates the conformance of the component behavior with its specification. The specification may be more general than the behavior; or in other words, the component may behave stricter than the protocol allows.

This divergence is not wrong in general — in terms of software engineering, any component may be replaced with any other implementation that respects the contract, including the behavior specification. However, a tool, which is able to identify these differences, may be very helpful — for behavior specification designers for example.

The tool has to cooperate closely with the behavior protocol checker, because the checker gets all required information to process the report — states and transitions of the protocol that were used during the checking process.

Iteration limit test

This test is focused on repetition operators in behavior protocols. Their usage cause the set of tested traces to be possibly unbounded. The solution of this problem introduces a limit of usages of every iteration operator on every particular trace. The solution is described in section 4.6.3 in more detail.

This test is designed to detect situations, when the iteration limit was reached and pruning the current search path is triggered. The total number of pruned paths is reported at the end of the checking process.

3.2.6 Dataflow/processes

This section presents diagrams of dataflow and business processes that operate with the data.

There are two figures. The first one describes the environment generation process. The next one describes the component behavior verification process. Inputs and outputs of the processes can be easily identified. Particular parts of the program code (classes) are also displayed here, because some of them are generated or linked at runtime.

Figure 3.4 shows the environment generation process. It has several inputs — ADL with the component definition, behavior specification, and value sets specification. It introspects also the checked component and its public interfaces. The

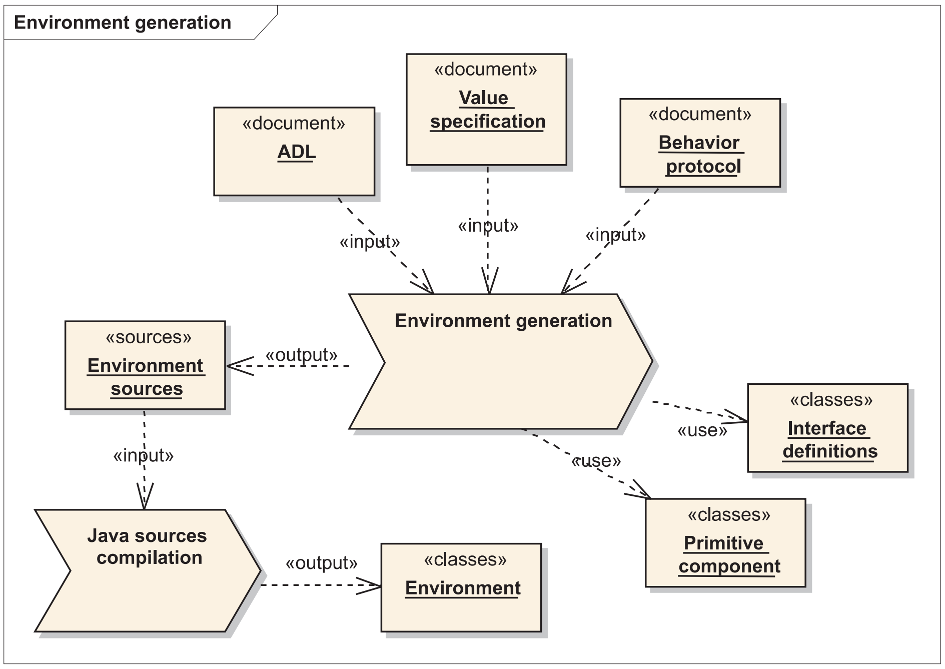

Figure 3.4: Process model — environment generation

only outputs of this process are source files of generated environment. These source files are to be compiled.

Figure 3.5 shows the component behavior verification process. A checked component with its environment is loaded into the JPF model space. On the host VM level, the remaining parts of the application operate. An error report is generated, when any behavior violation or other error is found. Also some additional reports may be generated (see section 3.2.5). The source code of the checked component is used to generate a code coverage report.

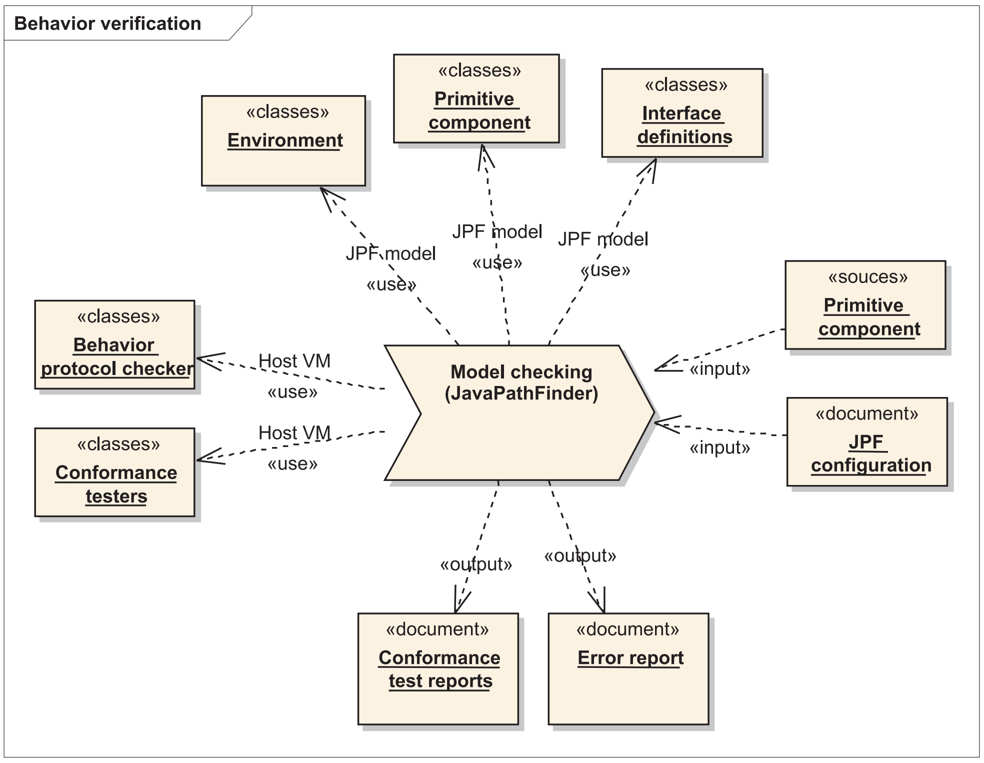

Figure 3.5: Process model — behavior verification

Chapter 4

Design

This chapter designs the solution that was introduced and analysed in the previous chapter. It describes platform dependant aspects and also some implementation details. It is focused on the model checking process and on the environment.

The most critical characteristics of the design is performance. The design had been confronted with the partial implementation very often, because the Java Path-Finder, as runtime environment, acts very different from the standard one in many aspects.

4.1 Design overview

The most important aspect of the design is what code is executed by the JPF virtual machine (that means it is part of the JPF model) and what code is executed in another way (host VM in this case).

The section 2.5.2 provides some reasons to let the JPF model as small as possible. However, putting code out of the model may cause some additional issues.

Of course, the checked component has to be inside the JPF model, as well as the driver and stub objects that are bounded to it. The component is the object of verification. In addition to this, the JPF model contains a controller object that communicates with the JPF model environment, with the BPChecker, and it makes all work together. Nothing more has to be placed into the JPF model. The BPChecker and some reporting tools remain outside of the JPF model space. Figure 4.1 shows this architecture.

The following sections describe each part of the schema — environment controller, driver/stub objects, behavior protocol checker. Their cooperation and also some more advanced topics are also discussed.

4.2 Environment and BPChecker cooperation

Cooperation between the environment and the behavior protocol checker is critical for this concept. The environment is driven by the checker — it has to know what

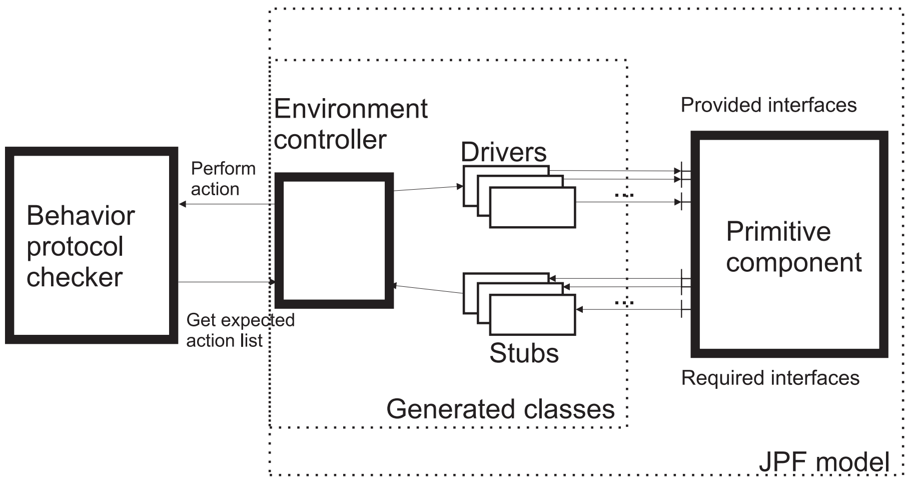

Figure 4.1: Top-level view of the checking process.

events are allowed in the next step of the verification process to be able to emit any correct sequence of events; and also, the behavior protocol checker depends on information about events occurred that come from the environment to manage its state.

Every change of the BPChecker state, which is caused by a notification about an event occurred and which runs in the host VM, implies a change of events allowed in the next step, which are stored in the JPF model because they are interesting for the environment. This change has to be done synchronously with some JPF model thread to be sure that JPF tests all possible traces — the JPF is not designed to expect asynchronous changes of the model data.

These two interactions between the JPF model and the BPChecker may be joined into one synchronous action. This action is triggered by the environment whenever it detects an event occurred. The BPChecker is notified by a MJI call and the JPF model is updated during this call. However, the way to implement the event detection and the MJI call has to be examined carefully. This section is focused on this issue too.

Note

The environment and the BPChecker use numeric codes to identify events on public interfaces. During the checking process, there is no reason to use symbolic names instead. The assignment of codes to events can be done in the environment generation phase and the BPChecker can spare time and memory by using numeric codes in the runtime.

4.2.1 Notifying BPChecker

An important decision is which part of the code is responsible for notifying the BPChecker about events occurred. There are several options, i.e., driver/stub objects, the checked component or the runtime (JPF or an extension of it).

It is correct to notify the BPChecker by the runtime according to the definition of behavior protocols semantics — the checker is notified just when an event is proceeding. However, it is hard or even not possible to implement this using the selected concept — consequences of the notification would have to be propagated from the runtime to the JPF model synchronously. It is preferred to notify the BPChecker in code of the driver and stub objects to avoid this as well as to avoid changing the code of the checked component.

In general, there is always a source and a destination of each event. The event is either a method call or a return on some of public interfaces. The BPChecker can be notified by the event source (1), by the event destination (2) or by the runtime (3) — see figure 4.2.

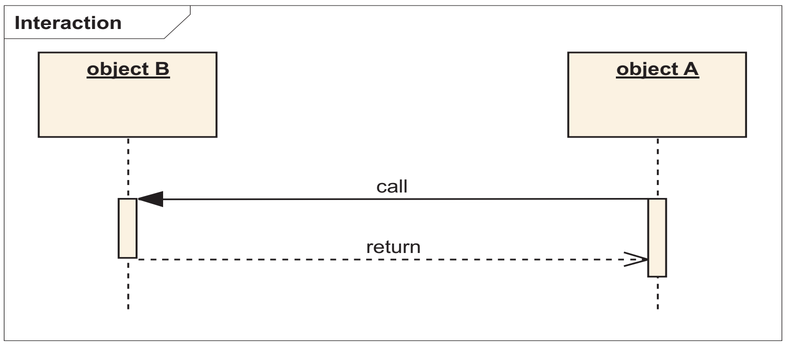

Figure 4.2: A simple call — return scheme.

In case of (1) or (3), the critical thing is the time between the event and the checker notification. Other running threads can execute code (including emitting events) in this time interval and cause therefore a false bad activity violation; because the BPChecker gets event notifications in an order different from the one they have happened.

Unsynchronized methods

Supposed methods of provided/required interfaces are not mutually excluded (in terms of Java, they are not synchronized), the choice (1) or (3) instead of (2) is correct — between the event and the checker notification is no synchronization; the order of events and notifications is therefore discretionary, it depends just on the scheduler. In that case, reported behavior violation is not a false violation.

Synchronized methods

In case of mutually excluded (synchronized) methods, it is necessary to analyse the following two cases:

- Required method calls and returns from a required method — the checker notification should be placed into the stub object to assure that the monitor lock is held between the event and the checker notification.

- Provided method calls and returns from a provided method — the monitor lock is acquired after the method is called by the driver object and released before the return to the driver object. Thus, the checker notification should be placed in the component's code. However, to avoid changing code of the component, it is possible to use explicit synchronization on the component object to get its monitor lock before the provided method is called and notified; and release it after the return and the checker notification.

Note that events originated in the environment are emitted just after they are expected according to the state of the behavior protocol checker. Therefore, they cannot cause bad activity in connection with other active events of the environment. However, these events can interfere with events emitted by the component — which is the reason for explicit synchronization in the case of provided method. Consider several examples of protocols that demonstrate this requirement below:

- *(?A.x^ ; !A.x$) + (!B.y^ ; ?B.y$)* in case of a provided method call.
  Supposed the method A.x is synchronized and the method B.y is called only if the component monitor lock is held by it to mutually exclude these two events. It is obvious that the notification of the *?A.x^* event has to be done inside the critical section, which requests the component, to avoid notifying the *?A.x^* event after the monitor lock is acquired by the component and the B.y method is possibly going to be called.
- *?A.x^ ; !A.x$; !B.y^ ; ?B.y$* in case of return from a provided method.
  Supposed the method A.x is synchronized and and wakes up another thread that calls the B.y method after it acquires the monitor lock of the component. It is obvious that the notification of the *!A.x$* event has to be done inside the critical section, which called the A.x method, to ensure that the *!A.x$* event notification is done before the other thread is allowed to call the B.y method.

Note

Required methods are never synchronized by default in the current implementation because Java does not allow declaring synchronized methods in interface types; stubs are generated from them. However, it is possible to change signatures of stub methods by hand to analyse the behavior of such component system.

Summary

It is possible to implement checker notifications in driver and stub objects only. Therefore, neither modifying the code of the checked component nor asking the runtime for information about method calls and returns and propagating them to the JPF model is required.

4.3 Environment controller

The environment controller is a generated class. It contains the main method and it is responsible for initialization of the component, the environment and the BPChecker. However, the primary purpose of this class is the control of active events of the environment, which means determining events to be executed and their order. Active events are provided method calls and some of required method returns. Details on required methods are presented in section 4.5.

The controller has to ensure that any active event may be emitted by the environment whenever it is allowed to be emitted by the BPChecker. To satisfy this, the controller contains a loop that does the following:

- It chooses non-deterministicly one of the active events allowed by the BPChecker and lets the competent driver/stub object execute it.
- It is assumed that execution of every event is done asynchronously — the execution of the controller may proceed before previous calls are finished. However, the state of the BPChecker has to be changed synchronously before the next iteration of the controller is executed to avoid an infinite loop here.
- If no active events are expected, the controller thread falls asleep.
- If no events are expected at all, the controller initiates the final-state check and then the current checking branch is terminated.
- The controller thread is woken up after any event is notified to the BPChecker.
- Choosing the event to execute and notifying the BPChecker are mutually excluded.

Note

The environment controller allocates one thread for its activities in the JPF model.

4.4 Driver objects

Driver objects are objects of the generated environment that are responsible for:

- notifying the BPChecker about provided method calls and returns, and
- calling provided methods asynchronously. If provided methods have arguments, their values have to be specified.

Calling synchronized and unsynchronized provided methods are discussed separately. These two cases are differentated during the environment generation phase by introspecting implementation of the checked component. The mechanism of asynchronous calls is described in the section 4.4.3.

4.4.1 Calling unsynchronized provided methods

Calling an unsynchronized provided method is easy. The BPChecker is notified about the provided method call synchronously (still by the environment controller thread) and the rest is done asynchronously — calling the component and notifying the return. Calling synchronized provided methods is discussed in the following section.

4.4.2 Calling synchronized provided methods

In case of mutually excluded (synchronized) provided methods, the environment controller cannot determine the point when a synchronized provided method is called. The reason is that performing a synchronized provided method call is done simultaneously with acquiring the component monitor lock. That may take some time and also other events may be emitted meanwhile. Therefore, it must be ensured that the provided method call is still expected by the BPChecker after the monitor lock is acquired.

Two ways were designed to call synchronized provided methods.

Unsafe mode

This method assumes that any time a synchronized provided method is allowed by the BPChecker to be called, it is possible to acquire monitor lock of the verified component without performing any other active events by the environment meanwhile. If this assumption is not satisfied, a deadlock caused by the environment may occur. In this case, the environment generation can be switched into the safe mode. On the other hand, the unsafe method is significantly faster than the alternative one.

Calling a synchronized provided method in unsafe mode is similar to calling unsynchronized provided methods. The main distinction is that the monitor lock of the checked component is acquired before all other activities are done. Note that the monitor lock is acquired not by the environment controller thread but by the thread, which is interested in calling the provided method. The controller thread waits until the monitor lock is acquired.

Safe mode

In this mode, the environment controller thread does not wait for acquiring monitor lock of the component. Its request just indicates that the provided method is planned to be called from now on. Consequently, the BPChecker cannot be notified synchronously by the environment controller thread, the notification has to be done later — after the monitor lock is acquired.

The environment controller is not allowed to plan any synchronized method to be called multiple times at a moment. It has no sense for synchronized methods; moreover, this condition eliminates infinite loop in the environment controller thread.

The asynchronous code acquires the monitor lock of the checked component. After that, it is ensured that the call is still allowed by the BPChecker; the BPChecker is notified and the provided method is called.

This model of calling synchronized provided methods is correct because the scenario laid out by the environment controller is one of possible checking paths and wrong scenarios are eliminated by the test inside the critical section.

4.4.3 Thread management

Driver objects depend on the ability of executing code asynchronously. This section describes how this feature is designed.

Additional threads are required to execute some code asynchronously. Section 2.5.2 discusses some reasons why creating a new thread on every request to an asynchronous call is not reasonable. A more efficient solution is implemented by the event manager.

The event manager is based on a thread pool with a fixed size. Fixing the size of the pool is the most suitable way with respect to the state matching technique of the JPF. The manager contains a list of events that should be executed. Events are allowed to appear in the list more than once. The threads in the pool contain an infinite loop that acquires events from the list and executes the code associated with them. If there is no event in the list for a particular thread, the thread waits. A waiting thread is woken up after some event is added to the list.

Consider the situation, there is no thread in the pool ready to execute a scheduled event — all threads in the pool are busy. This should never happen; otherwise the checking process would be incomplete. To avoid this, the thread pool has to be sized sufficiently; however, making the pool overlarge may cause the checking process to be significantly longer — amount of work of JPF typically grows exponentially with the number of threads inside the JPF model.

Section 4.7 discusses various ways to determine the optimal thread pool size.

Note

There are good reasons for performing primitive component behavior verification using less than sufficient count of threads in the pool, for example:

- When the sufficient thread count for a particular protocol cannot be determined.
- To estimate time requirements for a verification that use sufficient number of threads.
- The checking process may find some violations of the component behavior faster.
- It may be the only way to get over the checking process on too complex protocols or component implementations.

However, some errors may be omitted in that case — sufficient thread count is required for exhaustive verification. Using insufficient thread pool size may also lead to a deadlock during the checking process.

4.5 Stub objects

Stub objects are objects of the generated environment that are in general responsible for:

- notifying the BPChecker about required method calls and returns,
- delaying the method return until it is expected by the BPChecker, and
- returning a value, if the required method is not void type.

At first, consider these two behavior protocols:

(1) *!A.x^ ; ?A.x$*

(2) *!A.x^ ; ?B.y^ ; !B.y$; ?A.x$*

In the first case (1), the required method return follows the call immediately. In the second case (2), events *!A.x^* and *?A.x$* are separated by other events, therefore, the required method return has to be delayed until it is expected by the BPChecker. If there are no assumptions on events that are between the required method call and return, these inner events have to be executed by threads that are not interested in handling of this required method call — consider protocol *!B.y^ ; !A.x^ ; ?B.y$; ?A.x$*.

This required method return hold-up is implemented by a synchronization mechanism which makes the checking process slower. The slowdown is not critical; however, it is suitable to distinguish these two cases during the environment generation phase and significantly optimize the first case (1), which is very common.

4.5.1 Simple stub methods

A stub method that is referred to as *A.x* is treated as simple, if all occurrences of the event *!A.x^* are immediately followed by the event *?A.x$* in the behavior protocol. All the other stub methods are treated as stub methods with a conditional return (for differences see the following section).

A simple stub method has only to notify the BPChecker about the method call and return and to return a non-deterministically chosen value if it is required.

Note

The environment controller does not have to care about the returns from required method in case of simple stub methods. They are handled by simple stub methods.

4.5.2 Stub methods with a conditional return

Stub methods of this type have to be identified during the environment generation phase and handled in a different way. The environment controller has to treat the returns from them as active events (see section 4.3) and also the generated stub code is different from simple stub methods. Instead of notifying the BPChecker about a required method return, the required method call is blocked until a specific request from the environment controller is received. The omitted BPChecker notification is done by this request (by the environment controller thread).

The waiting has to be realized by using a helper object because of two reasons:

- It is important to determine the required method of the stub object that should be notified by the environment controller thread.
- If the required method is synchronized, the monitor lock of the stub object should not be released when the thread gets blocked (see semantics of the Object.wait() method in Java).

Note

Consider the protocol *!A.x{?B.y} | !A.x{?C.z}* and the trace *!A.x^, !A.x^, ?B.y$, !B.y^*. The expressiveness of behavior protocols is not sufficient enough to recognize which call of the component should be returned at this moment. Therefore, all available choices are tested by the JPF in the current implementation. This issue is called the parallelism problem.

4.6 Behavior protocol checker details

In the first part of this section, we introduce the design of a new behavior protocol checker implementation, which is required for the task. The second part describes some issues related to engaging the BPChecker implementation in the primitive component verification process. The last part discusses the problem of unbounded state space of the JPF.

4.6.1 Behavior protocol checker design

A short description of the behavior protocol checker is presented in section 3.2.2. Several existing implementations of the behavior protocol checker can be adapted and used (see [11]); however, a new checker was built up. The main reason is that the new checker is designed for the particular purpose and it is more efficient.

The behavior protocol checker can be implemented in many ways; however, the state space explosion problem has to be defeated in every approach.

The first approach is based on nondeterministic finite state automata (FSA) enhanced by parallel nodes, which correspond to and-parallel operators. This idea is shown on figure 4.3. Time and memory requirements and also the number of states of such automata are comparable and similar to parse tree automata [12]; however, it is not possible to perform significant optimizations on both these structures. The most important advantage of the enhanced FSA structure is that it is very similar to behavior protocol parse trees and thus, it is easy to translate any automaton state back to the protocol state. Also the creation of the structure is quite fast.

The second designed approach is based on the previous one. The main moti-

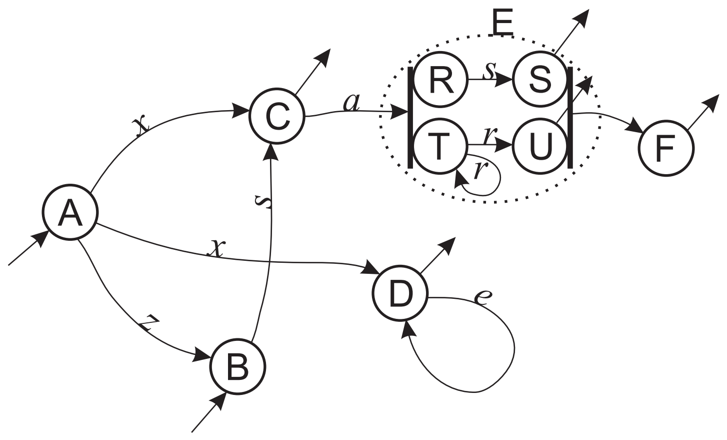

Figure 4.3: Nondeterministic FSA enhanced by parallel nodes.

vation is eliminating the hierarchy from the structure definition. In other words, it is driven by standard nondeterministic finite-state automata. The expansion of parallel operators is required, but then the state explosion problem gets in play. An automaton optimization procedure that looks for and eliminates automaton equivalent states is engaged to solve this challenge.

The final approach inheres just in a small modification of the previous one. It makes the automaton deterministic before the optimization procedure to ensure that the resulting automaton is a minimal one. Making the automaton deterministic can (theoretically) produce exponential growth of automaton state set; however, this is not the case of common protocols — typically, the growth is negligible.

Although the selected solution is quite elementary, it has many advantages:

- fast execution of all operations,
- states are uniquely identified by a single number,
- return to a particular state can be done easily without additional time or memory requirements,
- memory requirements are reasonable (thousands of states for complex behavior protocols),
- an explicit representation is suitable for other supplementary operations (e.g. reporting coverage of states/transitions of the BPChecker),
- set of states of the BPChecker is the smallest possible,
- a static representation is suitable for the runtime.

On the other hand, the solution requires some non-trivial pre-processing steps to build up the automaton data structure. Also, the size of the explicit automaton representation is limited.

Note

The state space of JPF is typically much larger than the state space of the behavior protocol; JPF uses explicit representation of each state as well as the BPChecker. It is obvious that JPF is not able to verify a system, whose behavior description is too complex to be represented by an automaton in the operational memory explicitly.

4.6.2 Model Java Interface usage

This section describes the way the BPChecker implementation (described in previous section) is plugged into the checking process. According to the section 4.1, the BPChecker communicates with the environment controller synchronously. However, there are several reasons to place the BPChecker out of the JPF model.

The JPF allows synchronous communication between the model and the host JVM objects using native methods (Model Java Interface, MJI). Some general characteristics and issues of using this technique are discussed in section 2.5.2.

These issues have to be solved before the BPChecker is taken out of the JPF model:

- the JPF state matching heuristics has to respect the state of the BPChecker to examine all possible behaviors (this is often called the 'Spearhead problem', described in [5]),
- the BPChecker has to restore its state simultaneously with JPF backtracks.

Both issues can be solved easily by propagating the current state of the BPChecker (a single number) to the JPF model. Backtracks of JPF may be detected by using the JPF search listener mechanism.

Note

The advantages of the designed BPChecker implementation (listed in section 4.6) fit very well to this architecture (MJI) and also the pre-processing does not cause any problems when it runs in the host JVM fast enough.

4.6.3 Unbounded JPF state space

This section is focused on guaranteeing of finiteness of the verification process. There are several aspects that may make the JPF state space unbounded:

(1) an infinite loop inside the component implementation code that does not contain any interactions with the environment but contains changes of the JPF model data; for example, the following code may cause such behavior;

Source code listing

```
while (true) i++;
```

(2) an infinite loop inside the component implementation code or inside the environment code, which contains activities on public interfaces and some data of the JPF model are changed by the component code inside the loop.

There are no other ways to generate unbounded state space of JPF — the state matching technique eliminates situations when no JPF model data are changed; the environment itself does not generate infinite number of unmatched JPF states, supposed there is no blunder in the generated environment code.

There is no way to detect the first situation (1), because it is JPF competence only, but the JPF model checker is not designed to detect this type of errors.

On the other hand, the second situation (2) may be detected by the behavior protocol checker. The situation implies that the behavior specification allows infinite set of traces; infinite set of traces may be allowed by the protocol only due to the usage of iteration operators. The behavior protocol checker has to provide a way to limit allowed trace count to a finite number.

Two ways are designed to do this reduction of allowed traces. Both of them limit the number of usages of iteration operators on every particular trace by a constant.

Note

The iteration limit usage makes the verification process possibly incomplete and some errors may be omitted. However, in case of unbounded JPF state spaces, there is no other alternative.

Behavior protocol rewriting

This technique affects the way, how the protocol automaton is generated from the behavior protocol. Instead of creating a loop, several alternatives are generated for iteration operators. The effect is the same as rewriting the input behavior protocol — see the example in table 4.1.

*(!A.x)**	*NULL + !A.x + (!A.x ; !A.x)*

Table 4.1: Iteration operator replacement for limit 2.

This solution is clear; however, the size of the automaton may grow significantly and also the information about the usage of particular transitions becomes a part of the JPF model. On the other hand, this does not make the JPF model larger in case of components that change their state anytime.

Search path prunning

This approach is based on pruning JPF search paths that exceed the maximal allowed number of iterations in a particular loop. It does not affect the automaton of the behavior protocol checker. Information about the usages of particular transitions are stored in another data structure outside the JPF model (management of the structure has to respect backtracks of JPF).

The current search path is pruned, if the iteration limit is reached. Therefore, the JPF model is not affected by introducing this limits at all, just the coverage of the state space is reduced. Also memory requirements of this solution are quite reasonable, because the length of any trace is limited. This solution seems to be very efficient; then, it is preferred to the behavior protocol rewriting based iteration limits.

Note

Although there is a solution of the unbounded state space problem, the component implementations, which affect the JPF model data too often, cause the verification process to be much more time- and memory-consuming. The test designer may optimize the component implementation by using tricks described in section 2.5.2.

4.7 Determining the optimal thread count

From the environment point of view, a behavior protocol allows the component environment to call provided methods concurrently. The maximal level of this concurrence defined by the behavior protocol is the most important property for the environment thread manager. This number also expresses the minimal sufficient (and also optimal) count of threads for exhaustive checking. In other words, at least the optimal thread count has to be active to verify the component behavior according to the protocol completely. Details about the thread manager are presented in the section 4.4.3.

Note

Just provided method calls cause the need of allocating threads on side of the environment. Required methods are unimportant for the purpose of this section, because they are executed by threads managed by the checked component.

Note

The maximal concurrence level does not exist when the number of concurrent calls may grow infinitely. This time, the checking cannot be complete according to the protocol at all.

Two ways to determine the optimal thread count are described in following sections.

4.7.1 Protocol analysis

The optimal thread count can be determined by a recursive analysis of the behavior protocol.

Let us define two integer functions $a(subprotocol)$ and $b(subprotocol)$ according to the table 4.2.

	Condition
$a(NULL) := 0; b(NULL) := 0;$	
$a(elem) := 1; b(elem) := 1;$ $a(elem) := -1; b(elem) := 0;$ $a(elem) := 0; b(elem) := 0;$	elem is a provided method call elem is a provided method return elem is another elementary event
$a(p1; p2) := a(p1) + a(p2);$ $b(p1; p2) := max(b(p1), a(p1) + b(p2));$	$a(p1) >= 0$
$a(p*) := a(p); b(p*) := b(p);$	$a(p) = 0$
$a(p1 + p2) := a(p1);$ $b(p1 + p2) := max(b(p1), b(p2));$	$a(p1) = a(p2)$
$a(p1\| p2) := a(p1) + a(p2);$ $b(p1\| p2) := b(p1) + b(p2);$	

Table 4.2: Definitions of $a(subprotocol)$ and $b(subprotocol)$ functions.

Also the condition $a(protocol) = 0$ for the whole protocol has to be satisfied. If any of these conditions is not satisfied, the protocol may allow traces that

- unlimitedly allocate or deallocate threads (in case of the iteration operator),
- are not simulateable (e.g. schedule a return that has no respective call), or
- are not correct (e.g. some calls are never returned).

Note

These conditions should match semantic rules for writing behavior protocols, which do not exist yet. These conditions are reasonable just for determining optimal thread count; however, they are not sufficient for checking semantic correctness of the protocol.

The $a(subprotocol)$ function expresses how many threads are allocated by the environment to perform calls described by the sub-protocol. A negative value indicates that some (previously allocated) threads are released (returned).

The $b(subprotocol)$ function provides the optimal thread count for the sub-protocol when $a(subprotocol) = 0$.

4.7.2 Simulation

Another way to determine the optimal thread count is a simulation of the checking process. Number of allocated threads is investigated while all possible traces are traversed. The highest value that was reached during the simulation is treated as the optimal thread count.

The JPF state matching technique helps to avoid infinite simulation in case of infinite trace sets. However, it may still happen that the simulation (theoretically) never stops. This is expected when the optimal thread count is not finite. Consider the protocol *(?A.x^)* ; (!A.x$)**. Note that this protocol allows a lot of incorrect traces.

The simulation works as a part of the generated environment controller class.

4.8 Reporting tools

The verification process is monitored by several tools, which gather statistics and other information for reports. These reports are generated at the end of the checking process. They may help to find out possible problems or to prove the quality of the verification. Most of these tools were described in section 3.2.5.

Error reporting

When JPF detects an error, it generates an error report according to its configuration. This report is a general-purpose report that contains information about threads, executed code etc. However, when a bad activity of the checked component is detected, a more specific report is generated by the behavior protocol checker as well. This report is focused on behavior of the component and on describing the detected behavior violation in a more readable way.

The report contains the trace that has caused the behavior violation. The behavior protocol checker can also find out the shortest trace leading to the last valid checker state and mark this state in the original behavior protocol. More options may exist because of non-determinism of behavior protocols.

Code coverage reporting

This tool operates on the host JVM level to be able to detect instructions executed by the JPF VM. The JPF listener extension mechanism makes it possible to find out what is happening inside the JPF VM without touching the JPF itself. Executed instructions of analyzed classes are marked during the checking. A summary and reports for every analyzed class are generated at the end of the verification process. These reports highlights missed methods, source lines and byte code instructions in the source code by using different colours.

BPChecker states/transitions coverage reporting

The BPChecker marks used states and transitions of the protocol automaton during the checking process. At the end of the checking, the number of missed states and transitions is reported. In addition, a shortest event trace is printed for every missed state or transition. Such traces are allowed by the protocol, but they were not tested during the checking. Every particular trace may be analysed, whether it is caused by the environment or by the component.

Restriction of iteration reporting

Section 4.6.3 introduces a way to limit iterations efficiently. However, if the used iteration limit is reached, the verification is not exhaustive anymore. This tool informs, how many times the BPChecker had triggered exceeding of the iteration limit during the checking process.

4.9 Summary

This chapter presented a detailed description of the solution's design — highlighting design of the BPChecker and the environment, the cooperation of these two parts and last but not least, reporting tools, which give a lot of important additional information to the test designer.

According to these information, it is easy to implement the solution.

Chapter 5

Performance testing

Previous chapters proposed and designed a solution of the primitive component behavior verification task. This solution was successfully implemented; the implementation is called JANE. It is not just a prototype implementation; it has all required features for a real-life use.

This chapter provides a complex description on the performance of the JANE implementation. Its performance is also confronted with two other reference implementations — DSRG and Carmen checkers. Comparison criteria, testing samples and configurations are also described in this chapter.

5.1 Comparison

The verification process performed by Java PathFinder may be compared with respect to several aspects. Some of them were chosen with focus on objectivity of such rating. Their enumeration and more detailed explanation follow:

- checking time (referenced as 'time') — total checking time displayed in format hours:minutes:seconds. Environment generation phase, report generation and other pre-verification or post-verification steps are not included in this time.
- unique JPF states (referenced as 'unique') — total number of unique JPF states that were discovered during the verification.
- revisited JPF states (referenced as 'revisited') — total number of visits of states which were already discovered during the verification process. A particular state may be revisited several times.
- states per second (referenced as 'states/s') — average number of visited states per second. Visited states are both — unique and revisited together. I.e. the formula $(unique+revisited)/time$ is used; millisecond precision of the time value is used in this formula.
- used memory (referenced as 'memory') — maximal size of the host JVM memory heap in MB. The memory heap is always larger than amount of used memory. The corresponding system process consumes a few megabytes of memory more.

5.2 Recommended JPF configuration

Java PathFinder is a very open and highly configurable system. Thanks to this, the JPF model checker is applicable to a wide range of software system types. However, configuration of JPF should always be tuned for a specific type of application.

The solution of the primitive component behavior verification task, as described in previous chapters, has several attributes that should be reflected in the JPF configuration. Description of them follows. We distinguish only between default and recommended configurations.

5.2.1 Thread symmetry reduction

The proposed and designed solution is characterized by using a fixed-size thread pool to avoid creating a lot of threads during the checking process. Detailed description of this aspect is presented in section 4.4.3. This section is focused on choosing the most suitable configuration of JPF with respect to this aspect.

The JPF state matching heuristic can be more efficient by introducing symmetry reduction based on permuting threads. Thread identifications are relevant to this heuristic by default. Consequently, a particular state of the system may be represented by as many as $N!$ JPF states where N is number of threads in the pool.

This issue is caused by the default serializer of JPF. Serializer is an abstraction of JPF, which is responsible for translating of the current state of the verified system into a sequence of bytes used by the state matching heuristic. The default one (FilteringSerializer) runs quite fast and addresses the heap symmetry reduction but not the thread symmetry reduction.

A much more complex serializer is the AbstractingSerializer, which is also a part of JPF. Among others, it addresses the thread symmetry reduction using a heuristic (the problem is as hard as the graph isomorphism problem); however, its execution is slower.

A new serializer was designed for the specific purpose of this project (OptimizedFilteringSerializer). It is based on the FilteringSerializer; it affects just the order of threads in the processing. Its efficiency depends on some restrictions in the JPF model space; however, they are fulfilled by the environment and the event manager.

Following tables present a simple comparison of performance of the aforementioned serializers.

Table 5.1 presents a situation, which cannot be optimized by introducing thread symmetry reduction (just 1 thread in the pool); however, the speed of these serializers can be compared.

Serializer	time	unique	revisited	memory
FilteringSerializer	0:00:43	89,078	73,203	10.2
AbstractingSerializer	0:01:14	89,078	73,203	10.0
OptimizedFilteringSerializer	0:00:44	89,078	73,203	9.1

Table 5.1: Serializers - AccountDatabase example with only 1 thread in the pool.

Table 5.2 shows that the AbstractingSerializer may give similar time results as the OptimizedFilteringSerializer due to state space size reduction.

Serializer	time	unique	revisited	memory
FilteringSerializer	00:16:48	775,941	2,824,956	36.8
AbstractingSerializer	00:00:54	22,769	84,726	9.1
OptimizedFilteringSerializer	00:00:48	35,961	121,654	8.3

Table 5.2: Serializers - Firewall example with 5 threads in the pool.

Table 5.3 demonstrates that the OptimizedFilteringSerializer may be significantly faster than others.

Serializer	time	unique	revisited	memory
FilteringSerializer	1:58:01	9,558,378	15,135,612	355.0
AbstractingSerializer	1:20:48	3,851,556	6,063,560	130.8
OptimizedFilteringSerializer	0:53:27	4,501,683	6,558,802	140.3

Table 5.3: Serializers - AccountDatabase example with 3 threads in the pool.

Although this comparison has not an obvious winner, the OptimizedFilteringSerializer was chosen to drive all tests presented in the following sections because of its performance. However, if memory limitations are primary, the AbstractingSerializer may be used instead of it.

Note

Introducing the OptimizedFilteringSerializer is not a hack into Java PathFinder. It is a general-purpose extension of JPF, not a change of its code. Serializers are switched using the vm.serializer.class property of the JPF runtime configuration.

5.2.2 Component environment

JPF model space contains two parts — a checked component and its environment. Although the environment is specific for every component, it is always generated the same way. It uses the same patterns which are already well tested. Checking time may be decreased by reducing the effort of JPF on proving correctness of the environment.

JPF parameters vm.por.field_boundaries and vm.por.sync_detection instruct JPF to detect some types of synchronization issues in the JPF model; however, the JPF state space is much larger then. It is useful just for specific purposes; it can be disabled globally or selectively just for classes of the environment by setting the vm.por.exclude_fields property. It is disabled in the tests below.

5.3 JANE input parameters

Values of two orthogonal parameters have to be specified when running JANE — number of threads that are managed by the environment and iteration limit. This section describes the meaning of these parameters and their influence to the checking process; it may also help to test designers with choosing suitable values.

5.3.1 Number of used threads

This parameter specifies the number of threads which are managed by the component environment. One of these threads is always used by the environment controller; the other threads are reserved for performing provided method calls. Thus, the parameter value should always be at least two.

Although several optimization were introduced, this parameter is still of major importance for complexity of the verification process especially in connection with a complex implementation of the checked component.

There is no reason to use value higher than the optimal value (see section 4.7). However, the optimal value may be also too high in case of complex behavior protocol and component implementation. In this case, a value between two and optimal thread count can be used. However, if the used value is smaller than the optimal one, the verification is not complete and some error may be omitted. Some important traces which were not tested are reported (see the section 4.8 for details). Influence of the thread count parameter on completeness of the verification process is demonstrated in table 5.4.

Thread count	Missed BPC states	Missed BPC transitions
2	1,008	3,514
3	786	2,848
4	350	1,360
5	0	0

Table 5.4: Influence of the thread count parameter on completeness of the verification process (ValidityChecker example with five optimal threads).

The figure 5.1 shows the relation between checking time vs. number of threads. The AccountDatabase example was used to make this chart. Measured values are marked with diamond points; the curve shows a computed, exponential type regression. Estimated time for five threads in this example is about 8 hours, for 6 threads almost 3 days.

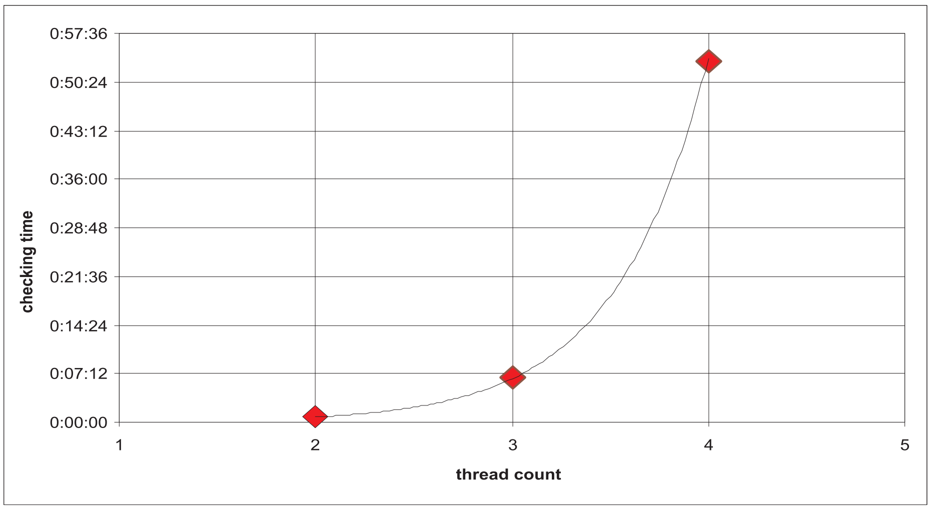

Figure 5.1: Relation between checking time vs. number of threads.

Note

An alternative implementation of the environment controller was created; it uses threads in the thread pool and avoids allocating one additional thread. However, results of such modified solution were worse than the original one with one more thread in the JPF model — see the table 5.5. The reason is that the amount of work of JPF is the same in both cases, but the modified attend requires additional synchronization.

Thread count	time	unique	revisited
0 + 3	00:02:33	251,417	463,372
1 + 3	00:01:37	172,789	311,275

Table 5.5: Controller thread elimination on the ValidityChecker_simple example.

5.3.2 Iteration limit

Iteration limits were introduced in section 4.6.3. The usage of iteration limits can possibly be a cause of incompleteness of the verification process. Two ways were proposed and designed to implement iteration limits. This section presents some real-life experiences using them.

Protocol rewriting based iteration limit

The table 5.6 shows the influence of using protocol rewriting based iteration limit using the AccountDatabase example with two threads. In this case, number of states of the BPChecker is affected by the iteration limit value significantly — cycle limit of 3 is too high to construct the BPChecker automaton in 600 MB memory pool.

Limit	BPC	time	unique	revisited	memory
1	315	00:00:07	12,087	11,032	6.8
2	2,275	00:01:57	211,815	199,199	15.6
3	——	00:10:40	——	——	600

Table 5.6: Protocol rewriting based iteration limit influence table.

Search path pruning based iteration limit

The table 5.7 shows the influence of the search path pruning based iteration limit on the same examples as above. Although the first row gives results worse compared to the previous method, this one is more suitable for all the other cases. In this method, the number of BPChecker states is fixed.

Limit	time	unique	revisited	memory
1	00:00:13	27,139	20,883	7.3
2	00:00:30	58,958	47,372	8.4
3	00:00:44	89,078	73,203	10.0
4	00:00:56	117,799	98,781	11.9
5	00:01:14	153,862	130,133	12.0
10	00:02:33	315,863	273,063	17.7

Table 5.7: Search path pruning based iteration limit influence table.

The number of JPF states and checking time grow together with the number of traces allowed by the iteration limit value. In this example, the trace count grows linear — see the figure 5.2. Diamond points show checking times from the table 5.7, triangle points show times from the previous table (5.6). The black line shows the computed linear regression of diamond points.

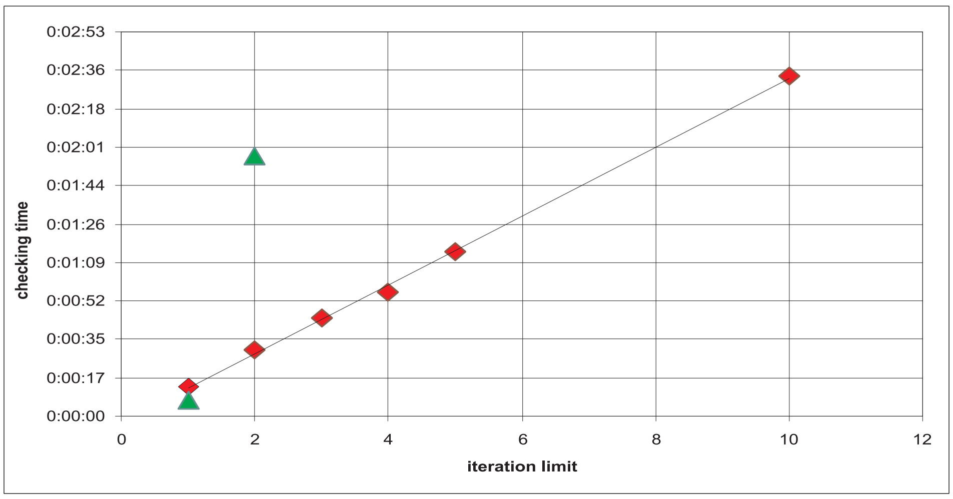

Figure 5.2: Iteration limit influence chart.

5.4 Introduction of reference implementations

In following sections, the performance of JANE implementation is confronted with two other solutions that face the same problem — primitive component behavior verification. These reference implementations are introduced in this section.

5.4.1 Carmen checker

Carmen is a prototype implementation of the JPF modification concept, which was discussed in section 3.1.2. Beyond general attributes of the concept, Carmen introduces several other aspects, which have to be considered when comparing its performance to other tools.

The usage of parameter types on public interfaces is limited just to primitive types (boolean, int) in Carmen. Thus, all examples had to be simplified and adapted to this restriction.

Moreover, Carmen introduced so-called thread-suffixes — an extension of behavior protocols, which affects both — syntax and semantics of behavior protocols. This change implies a two-way thread-parallel operator correspondence — a thread is created for each parallel operator in the behavior protocol and vice versa. That involves a significant reduction of non-determinism and the JPF state space. However, the meaning of such verification is not clear with respect to the original behavior specification.

Another restriction of the Carmen checker is the way the Spearhead problem is solved. Anytime, when the JPF state matching heuristic triggers backtrack of the JPF state, the BPChecker is queried, whether it is in a new state or not. A backtrack is not allowed in a newly discovered BPChecker state. This rule solves the Spearhead problem; however, such cooperation of the BPChecker and JPF is not correct with respect to several situations — see example on figure 5.3.

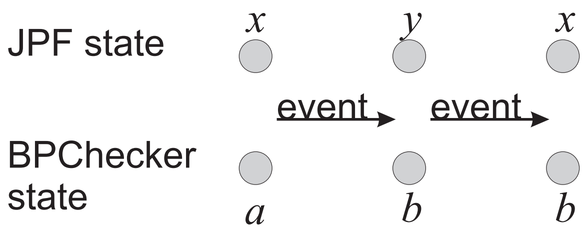

Figure 5.3: Example of unsatisfactory cooperation between JPF and the BPChecker.

A backtrack of the system is allowed in a revisited state of JPF (last occurrence of the state x), when the appropriate state of the BPChecker (b) was already seen. However, it is not checked, if these states (x and b) were visited together. If they were not, the backtrack is not correct. The behavior of the checked component is never verified in such situations, the verification is therefore not complete and some errors may be omitted.

Also the evaluation of Carmen is more complicated compared to other implementations. Carmen prints some verification statistics, which are different from statistics printed by standard JPF logging tools. Since author of the Carmen implementation did not provide any satisfactorily explanation of this distinction, presented results are gathered by standard tools, which are common for all compared implementations. However, some operations done inside the modified JPF but not visible to the JPF model, may be hidden to these tools.

5.4.2 DSRG checker

Software Component Model Checker was proposed and developed in the Distributed Software Research Group at Charles University [7]. This tool is referred to as DSRG or DSRG checker in the following text. It is inspired by the protocol driven environment concept, which was introduced in section 3.1.1. Thus, all benefits and disadvantages of this concept also hold for the DSRG checker.

The way the environment is generated limits the tool usage to special types of behavior protocols. The solution introduces a one-way thread-parallel operator correspondence — a thread is created for each parallel operator in the behavior protocol. For example, nested active events of the environment are not handled correctly. Another issue that cannot be addressed by this concept is the parallelism problem (see the note box in section 4.5.2) or the alternative problem (motivated by protocol *?A.x + !B.y*). Repetition operators are always rewritten to several alternatives.

Moreover, generated environment does not use more than two threads at a time. The DSRG checker usually requires simplification of complex protocols to drive the environment; therefore the verified component cannot be exposed to all situation allowed by the original protocol. Also the solution of the Spearhead problem is the same as in the Carmen implementation; so it is incomplete and possibly incorrect.

5.5 Results

This section presents confrontation of performance of the Carmen and DSRG checkers with JANE.

5.5.1 Test examples

All three implementations are tested on the Demo project. It presents a prototype implementation of a payment system for public Internet access on airports. Its complex description is presented in [10].

Due to the restrictions of the Carmen checker, components from the Demo cannot be verified directly. Distribution of Carmen contains some components from the Demo, which were adapted for Carmen. This adaptation caused also simplification of these examples. This simplification is demonstrated by the distinction of results between Carmen (simplified, 5.5.3) and DSRG (original, 5.5.4) examples verified by the JANE checker.

5.5.2 Test specifications

All tests were run on Pentium 4 (core Prescott) 3.2 GHz with 1 GB of memory, Windows XP Professional operating system. JANE and Carmen checkers were executed using JDK version 1.6.0, DSRG checker is restricted to JDK 1.4 — version 1.4.2_09 was used. JPF configuration for JANE was described in the section 5.2, other implementations were tested with JPF and its configurations that were provided with these tools without any changes.

5.5.3 Carmen checker confrontation

In this section, performance of the Carmen and JANE checker is confronted. All following examples were adapted to usability restrictions of the Carmen checker and were supplied by it. JANE checker performs an exhaustive verification; this assumption is not guaranteed in case of Carmen checker (5.4.1).

FlyTicketClassifier

The FlyTicketClassifier test presents an example of a simple protocol along with a simple component implementation. Results of both checkers are reconcilable — see the table 5.8.

Checker	time	unique	revisited	states/s	memory
Carmen	00:00:02	109	649	292	6.8
JANE	00:00:01	168	440	811	5.1

Table 5.8: Results of the FlyTicketClassifier test (Carmen version).

ValidityChecker, Arbitrator

The results of the ValidityChecker and Arbitrator tests are presented in tables 5.9 and 5.10. Both tests employ complex protocols with several parallel operators; however, the implementation code of these components is relatively simple.

JANE checker shows not only very good states per second performance but also very small model state space size. The nice state space size is caused, above all, by minimalization of the behavior protocol checker automaton (4.6) and by thread symmetry reduction (5.2.1).

Checker	time	unique	revisited	states/s	memory
Carmen	00:32:47	149,081	448,769	304	18.4
JANE	00:02:29	177,952	546,894	4,872	27.1

Table 5.9: Results of the ValidityChecker test (Carmen version).

Checker	time	unique	revisited	states/s	memory
Carmen	00:33:32	1,017,329	2,169,676	1,584	49.0
JANE	00:00:19	14,651	41,417	2,871	6.5

Table 5.10: Results of the Arbitrator test (Carmen version).

Note

Recall the most important drawback of the Carmen concept — the tool is independent of the official Java PathFinder source code now. For example, Carmen cannot be easily reconfigured to use thread symmetry reduction because JPF did not support this feature when the Carmen fork was done.

5.5.4 DSRG checker confrontation

This section presents confrontation of performance of the DSRG and JANE checkers.

The DSRG checker is able to verify just simple behavior protocols without modifications required (see 5.4.2, also other DSRG limitations are mentioned here). In case of more complex protocols, the DSRG checker uses simplified version of behavior protocol to drive the environment.

This simplification is not used by the JANE tests (with one exception), JANE uses original protocols to drive the environment and also to verify the system behavior. However, the test is executed several times with different configurations. Configuration (1) is similar to the DSRG test, it uses 3 threads. Restrictions of iterations are mostly not significant in following examples, and then they are applied just when they are necessary. Second configuration (2) uses optimal thread count, thus it demonstrates the ability of the JANE checker to perform a much more detailed verification than the DSRG checker.

FlyTicketClassifier

The FlyTicketClassifier test represents a simple protocol without any parallel operators along with a simple component implementation. The JANE checker performs an exhaustive verification. The results of this test are shown in table 5.11.

Checker	time	unique	revisited	states/s	memory
DSRG	00:00:17	707	143	50	76.1
JANE(2)	00:00:01	201	129	415	7.3

Table 5.11: Results of the FlyTicketClassifier test (DSRG version).

ValidityChecker

The ValidityChecker test represents a more complex behavior protocol along with a simple component implementation. The results of this test are shown in table 5.12.

The first configuration (1) is similar to restrictions of the DSRG test; (2) presents the results of an exhaustive verification with 5 threads.

Checker	time	unique	revisited	states/s	memory
DSRG	00:00:50	3,699	4,348	161	94.6
JANE (1)	00:00:13	16,034	26,792	3,224	11.5
JANE (2)	00:26:20	2,269,103	5,348,437	4,820	92.9

Table 5.12: Results of the ValidityChecker test (DSRG version).

Arbitrator

The results of the Arbitrator test are shown in the table 5.13. Arbitrator presents a complex protocol along with a medium-complex primitive component implementation. The first configuration (1) is similar to the DSRG test settings (3 threads); the second one (2) presents the results of an exhaustive verification with 6 used threads.

Checker	time	unique	revisited	states/s	memory
DSRG	00:59:52	321,806	214,609	149	134.9
JANE (1)	00:03:55	278,304	457,487	3,127	35.6
JANE (2)	01:16:54	3,367,237	10,944,947	3,102	251.4

Table 5.13: Results of the Arbitrator test (DSRG version).

IpAddressManager

The IpAddressManager test represents a complex behavior protocol along with a component, which uses a lot of synchronization methods and blocks.

The original component implementation does not obey its behavior protocol — the JANE checker detected a behavior violation omitted by the DSRG checker. The environment generated by the DSRG checker is driven by a simplified behavior protocol, which caused that the behavior violation was not detected. The implementation of the component was modified to adhere the original behavior specification for the JANE checker. Most methods of the modified component implementation are mutually excluded now. Consequently, a lot of traces allowed by the behavior protocol are restricted by such component implementation.

The results of the IpAddressManager test are shown in the table 5.14. In this test, the iteration limit was required to make the JPF state space finite. The iteration limit value of 2 was used, which is equal to the limit of the DSRG checker.

AccountDatabase

The AccountDatabase test employs a behavior protocol with 6 parallel branches along with a quite complex component implementation. In this case, the JANE checker was executed with the simplified protocol too, because the difference between

Checker	time	unique	revisited	states/s	memory
DSRG	01:58:46	705841	1011960	241	268.8
JANE (1)	00:00:32	20,567	24,606	1,419	11.0
JANE (2)	00:01:16	42,048	72,042	1,495	11.0

Table 5.14: Results of the IpAddressManager test (DSRG version).

the simplified and original protocol with the same input parameters is significant. Results of the verification with the simplified protocol is in row JANE (0).

The verification of the component behavior with the optimal thread count is too difficult for JPF because of an unsuitable implementation of the component. The component implementation should be adapted for the JPF environment first (by removing counters for example). However, it is possible to pass the verification with less than optimal thread count — the last test result row (JANE (3)) presents a result of the verification with 4 threads. Such verification gives much more sureness of the component behavior compared to classical testing or incomplete verification with optimal thread count.

The results of the AccountDatabase test are shown in the table 5.15. All examples in this table use the iteration limit value of 2 to make the JPF state space finite.

Checker	time	unique	revisited	states/s	memory
DSRG	02:57:19	1,266,355	1,828,675	291	151.7
JANE (0)	00:17:18	650,933	854,474	1,450	36.7
JANE (1)	01:15:26	2,880,846	3,894,118	1,497	127.2
JANE (3)	08:23:02	15,005,001	30,660,261	1,513	646.2

Table 5.15: Results of the AccountDatabase test (DSRG version).

5.6 Summary

The JANE checker seems to be a quite promising tool for verification of primitive component behavior. Its performance and memory requirements allow performing verification of complex behavior protocols without necessity of their simplification. It also provides parameterization, which efficiently reduce complexity (and completeness) of the verification process. That may be quite helpful for component implementations and behavior specifications too complex to be verified.

However, performance is not the only advantage of the JANE checker. This tool is able to perform exhaustive verification of the component behavior. It is applicable on any component implementation and almost any behavior protocol. Moreover, it provides several reporting tools, which analyse the verification process. Not least, it is free of modifications of Java PathFinder.

Chapter 6

Related work

6.1 Software model checkers

6.1.1 SLAM model checker and SDV

The SLAM model checker [16] is developed by Microsoft Research. Its focus is static verification of C programs. The verified properties are described in the SLIC language (Specification Language for Interface Checking). The most important challenge addressed by the SLAM checker is an automatized iterative abstraction of the model.

The SLAM model checker is used to drive the Static Driver Verifier tool (SDV). It is a tool for verifying some types of Windows drivers against a pre-defined set of rules. Similar to a software component, a Windows driver is not a complete program that can be verified by a model checker directly. However, it fits the Windows kernel environment and the SDV tool can simulate environment of several types of drivers (including function drivers, filter drivers, bus drivers and other). The pre-defined property categories fit most usual driver APIs, for example APIs that deal with I/O request packets, Plug and Play technology, interrupt request levels and other.

The SDV tool shows that the predefined environment is a suitable way to make incomplete programs checkable by model checkers, if application of such environment is frequent and important enough.

6.1.2 Bandera tools

Bandera [17] is a set of tools providing support for extraction of safe, compact, finite-state models that are suitable for verification from Java source code. These tools support slicing and abstracting of Java programs and include translators for several model checker backends. For example Java PathFinder, SPIN and Bogor model checkers are supported.

The Bandera environment generator (BEG) [18] is a tool that deals with incomplete programs. It is applicable to software components or programs sliced to several parts that are no more complete (modular model checking). BEG is able to generate an environment for these incomplete programs to make it complete. The environment is the most general one possible or it may be driven by a formal specification (a LTL formula or a regular expression).

The most important drawback of Bandera tools is that only alpha versions are available; however, they are still under development.

6.2 Primitive component checking tools

These tools were developed to deal with the primitive component behavior verification problem as well. Also several component models are discussed in the following section.

6.2.1 Component models

Many different component models were developed. Some of them are focused primarily on user interface components (OCX by Microsoft, VCL by Borland); others more or less respect paradigms of the component-based software engineering (e.g. Fractal by ObjectWeb, SOFA by DSRG, DCOM by Microsoft, CORBA component model by OMG).

Although this work is focused on the Fractal component model, it is easy to suit it to other similar component models.

6.2.2 DSRG checker

Software Component Model Checker was proposed and developed at the Distributed Software Research Group at Charles University [7]. It is inspired by the Protocol driven environment concept, which was introduced in section 3.1.1; several characteristics specific for this implementation were mentioned in section 5.4.2.

The tool is well proven now and it is applicable on any component implementation. On the other hand, time and memory requirements of this solution are high, even though the level of paralelism is strictly limited as well as iteration operator use. Moreover, some behavior protocols cannot be checked by this tool. A hand-made simplification of inputs is often required.

Although verification made by this tool is typically not exhaustive, the tool is still valuable. Its performance is good when simplified protocols are employed. It is also easy to modify this tool to support current version of JPF and Java.

6.2.3 Carmen checker

The Carmen checker [6] was developed as a prototype implementation of the JPF modification concept, which was discussed in section 3.1.2. Several characteristics of this tool were presented in section 5.4.1.

The most important advantage of the Carmen checker is that the JPF model is not loaded with any additional code; and it does not require any pre-processing steps such as environment generation.

On the other hand, the tool supports only primitive data types in provided/required interface definitions. Therefore the tool cannot be applied to real-life component systems, they have to be addapted by hand first. Moreover, the usage of thread suffixes, which is required, changes the meaning of the verification with respect to the original behavior specification. Not least, Carmen contains several design as well as implementation bugs, which cause the verification to be not exhaustive.

It is also obvious that the Carmen checker cannot be ported to newer version of the JPF tool easily.

Finally, this thesis has shown, that the basic assumption of the JPF modification concept[1], which says that some JPF modifications are required, was wrong and similar results may be achieved by an effortless way.

[1]stated in the Carmen work

Chapter 7

Conclusion

The goals of the thesis, which were stated in section 1.4, have been reached.

The most important contribution of this thesis is the solution of the primitive component behavior verification problem, which was proposed and designed by chapters 3 and 4; and then sucesfully implmemented. Chapter 5 demonstrated that the solution is applicable to real-life component systems with resonable time and memory requirements without any restrictions.

As a side-product, a collection of tips for Java PathFinder was written up and presented in section 2.5.2.

Last but not least, provided reporting tools turned out to be very helpful. Their employment is discussed in section A.3.

Although objectives of the thesis were ambitious and many problems had to be faced, results of this thesis exceeded expectations in many aspects.

7.1 Future work

The concept, the design as well as the implementation of the solution did not introduce any obvious drawbacks. However, several suggestions for the solution improvement follow.

7.1.1 Iterative verification

The solution is able to report states and transitions of the behavior protocol automaton which were not visited during the verification process. An incoming analysis may make a classification of these data — decide, what states/transitions were missed because of the environment and what because of the component implementation. An interesting idea is to execute a next iteration of the verification process with different parameters just on the missed state space of the behavior protocol.

7.1.2 Code coverage

The code coverage reporting tool is very helpful when standard tools for reporting code coverage are not applicable to the JPF virtual machine. Features of the tool may be extended by additional statistical information; also additional output report types (such as pdf, xml) may be useful. Another way to satisfy these requests is exporting code coverage data in a standard format that can be processed by existing code coverage tools and IDEs.

Bibliography

[1] F. Plášil, S. Višnovský: Behavior Protocols for Software Components. IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002

[2] Java PathFinder, http://javapathfinder.sourceforge.net/

[3] The Fractal Project, http://fractal.objectweb.org/

[4] Bernd Finkbeiner, Henny Sipma: Checking Finite Traces using Alternating Automata http://www-step.stanford.edu/papers/rv01.html

[5] Parízek, P., Plášil, F., Kofroň, J.: Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker

[6] Plšek Aleš: Extending Java PathFinder with Behavior Protocols, Master Thesis, advisor: Jiří Adámek, September 2006

[7] Distributed System Research Group http://dsrg.mff.cuni.cz/

[8] Adámek, J., Plášil, F.: Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance and Evolution: Research and Practice 17(5), pp. 363-377, DOI: 10.1002/smr.321, Online ISSN: 1532-0618, Print ISSN: 1532-060X, Sep 2005

[9] NASA Ames Research Center, http://www.nasa.gov/centers/ames

[10] http://kraken.cs.cas.cz/ft/doc/demo/Demo-Description.pdf

[11] M. Mach, F. Plášil, and J. Kofroň: Behavior Protocol Verification: Fighting State Explosion, IJCIS Vol.6, Number 1, ACIS, ISSN 1525-9293, pp. 22-30, Mar 2005

[12] Mach, M., Plášil, F.: Addressing State Explosion in Behavior Protocol Verification, Accepted for publication in proceedings of SNPD'04, Beijing, China, Jun 2004

[13] Adámek, J., Plášil, F.. Partial Bindings of Components - any Harm?. Presented at the SACT 2004 Workshop, Busan, Korea (held in conjunction with the APSEC 2004 conference), and published in the Proceedings of APSEC 2004, IEEE Computer Society, ISBN 0-7695-2245-9, pp. 632-639, Nov 2004.

[14] http://en.wikipedia.org/wiki/Component-based_software_engineering

[15] http://en.wikipedia.org/wiki/Model_checking

[16] http://research.microsoft.com/slam

[17] http://bandera.projects.cis.ksu.edu

[18] http://beg.projects.cis.ksu.edu

Appendix A

User documentation

A.1 Running JANE

A.1.1 HW/SW requirements

The JANE checker is written in Java as well as all libraries used by it. Therefore it runs on any platform supported by the Java Development Kit — version 1.5 or later is required. The JDK must be installed and the Ant tool is optional. All other supporting software is bundled with the JANE distribution.

HW/SW requirements for the JANE checker are:

- JDK version 1.5 or later
- Ant tool (optional)
- 100 MB memory for the Java process or more
- tested under Windows XP and Linux

A.1.2 Installation

No special instalation procedure is required. It is enough to copy the content of the JANE distribution CD to a read/write device. To uninstall JANE delete the JANE root directory.

A.1.3 Distribution content

The structure of the JANE distribution CD is shown in the figure A.1.

A.1.4 Building and module dependencies

The source codes of the JANE checker, DSRG examples, Carmen examples and generated environment are stored and compiled separately. However, there are some dependencies between these modules — see the table A.1; "libs" refers to libraries placed in the lib directory. These dependences should be reflected in the classpath settings while compiling or running.

Module	Depends on
JANE	libs
Carmen examples	—
DSRG examples	—
Generated environment	JANE, Carmen & DSRG examples

Table A.1: Source code dependencies.

All source codes can be recompiled by the simple command `ant` executed in the distribution root directory (supposed the ant tool is properly installed). Moreover, all modules may be compiled separately — see the table A.2.

Module	Build command
JANE	ant compile.module.jane
Carmen examples	ant compile.module.demo_carmen
DSRG examples	ant compile.module.demo_dsrg.production
Generated environment	ant compile.module.environment

Table A.2: Source code dependencies.

Generated environment module should be compiled always after environment of a component was (re)generated.

A.1.5 Running JANE

The environment generation phase precedes the component behavior verification process. The generated environment has to be compiled indeed. Inputs and outputs of these processes are described in section 3.2.6.

Generating environment for a component

The environment generation process is launched by command
`java cz.jane.envGenerator.EnvGenerator adl compName outPath package`
Meaning of all parameters is explained in table A.3.

Parameter	Meaning
adl	filename of the ADL (component application description)
compName	name of a primitive component to generate environment for (adl may contain several component definitions)
outPath	output base path where the generated files will be placed (do not include package directories)
package	package name where the generated files will be placed

Table A.3: Meaning of the environment generator parameters.

`run/envgen.cmd` and `run_unx/envgen` scripts may help with classpath and other JVM settings. Moreover, generated sources are always compiled by these scripts.

Additional envgen_* scripts generate environment for a particular component with respect to the structure of the JANE distribution.

Component behavior verification

After environment for a component is generated, it is possible to verify the component behavior. The verification process is started by commmand

```
java gov.nasa.jpf.JPF envControllerClass threadCount [iterLimit]
```

Meaning of its parameters is explained in table A.4.

Parameter	Meaning
envControllerClass	full class name of the generated environment controller class (package names have to be included)
threadCount	number of threads managed by the environment (see section 5.3.1), should be always at least 2
iterLimit	iteration limit value (see section 5.3.2); it is an optional parameter not defined value or zero — no iteration limit positive value — protocol rewriting based iteration limit negative value — search path pruning based iteration limit

Table A.4: Meaning of the behavior verification parameters.

run/verify.cmd and run_unx/verify scripts may help with classpath and other JVM settings. Additional verify_* scripts run verification of a particular component behavior.

A.2 File format specifications

A.2.1 ADL file

Format of the ADL XML file is specified by the Fractal ADL project. However, the format was extended to support additional features. JANE expects following elements in the component definition:

```
<protocol file="demos/dsrg/protocols/ValidityChecker.bp"/>
<environment>
<valuesets file="demos/dsrg/specifications/ValidityCheckerImpl.xml"/>
</environment>
```

Specifications of the protocol and valueset file formats follow.

A.2.2 Behavior protocol

JANE accepts behavior protocol syntax defined in [1]. Supported behavior operators are: *; + ** and |. No thread suffixes are expected.

A.2.3 Value specification XML file

Structure of the value specification XML file is shown in the figure A.2. It is enough to mention only parameters, whoose values have to be specified — input parameters for provided interface methods and output parameters for required interface methods.

The type element has to refer to a valid Java type — the import element may be used to shorten the Java type name. The valueRestriction element has to contain zero or more coma separated expressions with the type specified in the type element. The optional initialization element may be used to place static initialization Java code. The "___" string inside this code will be replaced by name of the generated variable that holds values for the parameter.

A.3 Output reports analyse

A.3.1 Code coverage report analyse

The code coverage report visualizes unexecuted code of JPF model classes in their source code. Missed methods, missed source lines and missed byte code instructions are highlighted with different colours.

Some reasons, why some code is missed, are explained by the following example:

Source code listing

```
public boolean CreateAccount(int AccountId, int Password){
   if (AccountId == 0 || Password == 0) return false;        (1) partially missed
   if (database.containsKey(new Integer(AccountId)))         (2)
      return false;                                          (3) missed
```

The line (3) was never executed. It is obvious, that the condition on the line (2) was never satisfied. In this case, either the value set for the AccountId parameter is not sufficient or the database object is not initialized properly.

The line (1) was missed partially — not all byte code instructions generated by this line were executed. In this case, the condition or its part was not satisfied. Short-circuit evaluation of boolean expressions is employed in Java.

A.3.2 Automaton coverage report analyse

The automaton coverage report contains three parts:

- A simple statistics
  Missed automaton nodes: 5
  Missed automaton transitions: 9
- Events identification translation table
  3 → *!IAfFlyTicketDb.GetFlyTicketValidity^*

6 → *?IAfFlyTicketDb.GetFlyTicketValidity$*
...

- A list of missed states and transitions
  = missed state path : 44, 3
  – missed transition path: 44, 3, 6
  ...

Transitions that start in a missed state are also always missed, such sequence of missed state and transitions is grouped:

```
= missed state path : 44, 3, 6
– missed transition path: 44, 3, 6, 45
– missed transition path: 44, 3, 6, 13
```

Several reasons are possible why a state/transition may be missed. They are demonstrated by examples in following sections.

Insufficient parameter value sets

Source code listing

```
public IToken CreateToken(String FlyTicketId, boolean RestrictValidity) {
   if (FlyTicketId == null) return null;
   ...
```

Missed state path: 44, 3, 6
3 → *!IAfFlyTicketDb.GetFlyTicketValidity^*
6 → *?IAfFlyTicketDb.GetFlyTicketValidity$*
44 → *?IFlyTicketAuth.CreateToken^*
Set of values for the FlyTicketId parameter should be extended by the null value.

Insufficient number of threads managed by the environment

Missed state path: 34, 23, 54, 54, 54, 54
23 → *!ICardCenter.Withdraw^*
34 → *?IAccount.RechargeAccount^*
54 → *?IAccount.AdjustAccountPrepaidTime^*
It is obvious that at least five threads are required in the event manager thread pool to visit such state of the BPChecker.

Difference between the component behavior and its specification

Behavior protocol:
?ILogin.LoginWithFlyTicketId {

```
    !IFlyTicketAuth.CreateToken ;
    (!IFirewall.DisablePortBlock + NULL)
  }
```

Source code listing

```
public boolean LoginWithFlyTicketId(InetAddress IpAddress, String FlyTicketId) {
    IToken token = iFlyTicketAuth.CreateToken(FlyTicketId, true);
    return false;
}
```

It is obvious that the behavior specification is more general compared to the component behavior. The listed source code is not present in the original examples — it is a modification that demonstrate such possibility.

A.3.3 Error report analyse

A specific report is generated when a bad activity is detected. It contains:

- The trace occurred, which is not allowed by the behavior protocol. The bad activity is caused by the last trace element.
  Example:
  ```
  Error trace: 44, 45, 44, 3, 6, 3
  Error trace length : 6
  ```
- List of events, that were acceptable instead of the last event occurred.
  Example:
  ```
  Event occurred: !IAfFlyTicketDb.GetFlyTicketValidity^(code 3)
  acceptable events are: {13, 45}
  ```
- The last valid state of the BPChecker marked in the behavior protocol.
  Example:
  ```
  ?IFlyTicketAuth.CreateToken {
      (
          !IAfFlyTicketDb.GetFlyTicketValidity ;
          ( → HERE_1← !IAfFlyTicketDb.IsEconomyFlyTicket + NULL)
      ) + (
          !ICsaFlyTicketDb.GetFlyTicketValidity ;
          (!ICsaFlyTicketDb.IsEconomyFlyTicket + NULL)
      ) + NULL
      → HERE_2 ←
  }
  ...
  ```
  A hierarchical numbering is used for the HERE tokens if parallel operators are used in the behavior protocol. The ordering may be several levels deep if parallel operators are nested.

build\			compiled JANE classes
src\			JANE source codes
lib\			external libraries, required for running JANE
documentation\			JANE documentation
demos\			sample components
	carmen\		sample components (Carmen version)
		build\	compiled classes
		protocols\	behavior protocols
		specifications\	parameter value specification
		src\	source codes
		demo.fractal	fractal component application definition
	dsrg\		sample components (DSRG version)
		build\	compiled classes
		protocols\	behavior protocols
		specifications\	parameter value specification
		src\	source codes
		demo.fractal	fractal component application definition
environment\			generated environment
		build\	environment compiled classes
		src\	environment source codes
reports\			output reports
run\			Windows scripts for running JANE
run_unx\			Unix scripts for running JANE
defauilt.properties, jpf.properties			JPF configuration files
build.xml			ant build file

Figure A.1: Structure of the CD content.

```
<component>
    <componentName>AccountDatabaseImpl</componentName>

    <providedInterfaces>
        <interface>
                <name>IAccount</name>
                <import>java.util.Date</import>
                <import>dsrg.EnvValues</import>
                ...
                <method>
                    <name>CreateAccount</name>
                    <inputArgument>
                        <name>AccountId</name>
                        <type>String</type>
                        <valueRestriction></valueRestriction>
                        <initialization>___ = EnvValues.AccountDbStrings;</initialization>
                    <inputArgument>
                    ...
                    <outputArgument>
                        ...
                    </outputArgument>
                </method>
                ...
        </interface>
        ...
    </providedInterfaces>

    <requiredInterfaces>
        <interface>
                ...
        </interface>
        ...
    </requiredInterfaces>

</component>
```

Figure A.2: Structure of the value specification XML file.