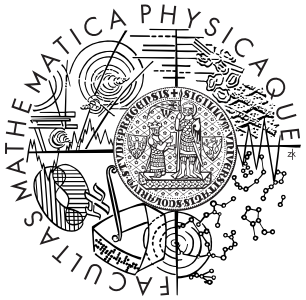
Charles University in Prague
Mathematical and Physical Faculty

MASTER THESIS



Aleš Plšek

Extending Java PathFinder with Behavior Protocols

Department of Software Engineering
Advisor: RNDr. Jiří Adámek
Study Program: Computer Science, Software Systems

Prague, July 2006

First of all I would like to thank my advisor Jiří Adámek. His suggestions, friendly and helpful attitude had a substantial influence on a progress of the work.

Special thanks go to the Distributed Systems Research Group, in particular to Jan Kofroň and Pavel Parizek, for helping with the BPChecker integration and mainly for the assistance during the performance testing.

Last but not least, I would like to thank my friends and family for their support.

I declare that I have elaborated this master thesis on my own and listed all used references. I agree with lending of this master thesis. This master thesis can be reproduced for academic purposes.

Prague, 28th July 2006 Aleš Plšek

Contents

Název práce: Extending Java PathFinder with Behavior Protocols
Autor: Aleš Plšek
Katedra: Katedra softwarového inženýrství
edoucí diplomové práce: RNDr. Jiří Adámek
E-mail vedoucího: adamek@nenya.ms.mff.cuni.cz

Abstrakt:

Komponentově orientované programování, využívající behavior protokoly pro specifikaci chovaní komponent, představuje efektivní přístup k vývoji softwarových aplikací. Jedním z prostředků jak ještě více usnadnit vývoj je i analýza korektnosti systému. Avšak bez ověření, že implementace komponent dodržují dané behavior protokoly, nemůže být analýza korektnosti systému formálně považována za úplnou.

Tato práce předkládá způsob jak ověřit, zda je implementace komponenty v souladu s daným behavior protokolem. Navržené řešení využívá nástroj Java PathFinder Model Checker k ověření implementace komponenty a tímto ambiciózním způsobem úspěšně čelí jednomu z otevřených problémů při vývoji komponentových aplikací.

Klíčová slova: softwarové komponenty, model checking, behavior protokoly

Title: Extending Java PathFinder with Behavior Protocols
Author: Aleš Plšek
Department: Department of Software Engineering
Supervisor: RNDr. Jiří Adámek
Supervisor's e-mail address: adamek@nenya.ms.mff.cuni.cz

Abstract:

The Component-based programming is an aproach to efficient development of software applications, where Behavior protocol is a formalism used for behavior specification of components. To further ease the development of a component application, System Correctness analysis is employed to discover errors at the designe time. However, without verifying that a component implementation complies with its behavior protocol, the analysis of the system could be based on unsatisfied property.

This thesis addresses the problem of verification that a component implementation adheres to its behavior protocol. By employing Java PathFinder Model Checker it offers an ambitious solution which is successfully addressing one of the issues of the software component development.

Keywords: Software Component, Model Checking, Behavior Protocol

Chapter 1

Introduction

It is well known fact that the Component-based programming [1, 2] is the answer to growing requirements on software systems. The basic idea is the composition of the application functionality from individual components. By interconnecting appropriate components the designer can easily obtain the desired functionality. This technology represents a cost-reducing approach facilitating to develop an application that can be easily maintained, updated and further extended.

For the purpose of component behavior specification, behavior protocols [1] were introduced. A behavior protocol specifies a behavior of the component in the system and thus facilitates to the designer to comfortably evaluate whether a specific component fulfils his needs.

To further ease the development of a software component application, it is necessary to address the problem of system correctness. When designing a complex application, it is needed to verify that employed components are used with respect to their behavior protocols. While the checking of compliance between behavior protocols of components has already been addressed [3], verifying that the component's implementation adheres to its behavior protocol is a weakness of the whole concept.

This thesis addresses the problem of verification that a component implementation adheres to its behavior protocol. By employing Java PathFinder Model Checker [4] it offers an ambitious solution which is successfully addressing one of the open issues of the software component development.

1.1 Software Components

In Component-based programming [1, 2] components represent building elements which form a software system. By composition of appropriate components the

designer can easily obtain the desired functionality. This approach offers a comfortable way of developing application that can be easily maintained, updated and further extended.

To sustain this approach, components are viewed as black-box entities providing their services through the strictly defined points - interfaces. Because a component is not defined only by its provisions, but also can have some requirements, there are two types of interfaces. Whereas *provided* interfaces define services provided by the component, *required* interfaces define services required by the component from its environment.

There are many component systems, this work is focused on two of them: SOFA [5] and Fractal [6].

SOFA and Fractal introduce hierarchical components. This approach classifies each component as either *composite*, consisting of lower-level components, or *primitive*, implemented directly in a common programming language. This idea allows to develop complex components providing the most sophisticated services.

1.2 Behavior Protocols

For the purpose of component behavior specification, SOFA employs behavior protocols [1]. The concept of behavior protocols is also an object of the Component Reliability Extensions for Fractal Component Model project [7] which introduces behavior protocols into Fractal.

As already said, a software component is not represented only by the services provided by a component, but also specifies the services which are required. From the component implementation point of view, services are abstractions for method calls on the interfaces.

In order to fully describe a behavior of the component, a behavior protocol defines all sequences of method calls that may appear on interfaces of a component.

1.3 Component Application Correctness

Once behavior protocols of employed components are available, it is possible to reason about compatibility of these components. There are two checks verifying compatibility of integrated components: *composition* and *compliance* test. While *composition* test checks that all components on a particular level communicate without errors, compliance test verifies that behavior of a composite component corresponds to behavior protocols of its subcomponents.

It is important to highlight that these tests are performed only at the level of behavior protocols, with no implementation code involved. Such an approach allows to identify architecture errors in early stages of design, thus decreasing risk of serious errors discovered at later phases of development

However, it is also necessary to address the problem of a primitive component implementation correctness. Without verifying that an implementation of a primitive component adheres to its behavior protocol, verification of the whole component system could be based on unsatisfied assumption.

1.4 Problem Statement

As already indicated, without verifying that a primitive component adheres to its behavior protocol, verification of the whole component system could be based on unsatisfied assumption. To face this issue, the tool presented in [8] was developed. However, the implemented solution comprises several disadvantages:

- Before the verification of a component can be started, nontrivial initialization operations (mainly focused on a frame protocol modifications) have to be accomplished. Moreover, the tool is not able to verify a primitive component directly. Therefore, a component implementation has to be extended with additional code which is wrapping a component and forms a complete program. The additional code is being verified together with a component and thus a performance of the tool is adversary affected.
- Additionally, the tool does not support a full scale of situations that can be specified by behavior protocols, e.g. special cases of the repetition or parallel operator usage. This restriction requires to simplify a frame protocol of a primitive component before the verification can be started. Therefore, a component implementation cannot be verified against the original protocol which may consequently prevents from identification of all protocol violations.

1.5 Goals

Having all the presented issues stated in mind, it is obvious that the question of component application correctness cannot be considered as answered without verifying primitive component implementations. Even though a tool attempting to verify a primitive component was implemented, the problems listed in Section 1.4 have shown that the tool cannot be considered as the final solution. Therefore, this thesis lays out the following goals:

- **(G1)** The first goal is to propose a solution that would bring comprehensive answers to the issues of a primitive component verification. A solution should fully support all behavior protocol operators and has to use rigorous methods of verification in order to guarantee that every protocol violation will be identified. At the same time, the proposal needs to be designed with respect to performance of the future implementation.
- **(G2)** A prototype implementation demonstrating correctness of identified solution should be implemented. The application should be able to verify a Fractal primitive component implementation against arbitrarily constructed behavior protocol without the need for additional reductions. Since the performance is also important aspect, the developed tool has to offer an acceptable time of verification.

1.6 Structure of The Thesis

To reflect the goals, the thesis is structured as follows.

Chapter 2 describes the software component systems checking in greater detail. Necessary background about Java PathFinder is also given here. In the light of information provided by this Chapter, concluding Section 2.4 defines more precisely the goals of this work.

Chapter 3 proposes an approach to a primitive component verification. It analyzes the issues that need to be faced and progressively introduce possible concepts of the solution. These concepts are consequently discussed and evaluated, thus the solution is being precisely refined.

In Chapter 4, the solution proposed Chapter 3 is presented in greater detail. The level of detail provided by this chapter is sufficient for the reader to fully comprehend the solution. Moreover, this Chapter introduces approaches to crucial issues of the software component verification.

Chapter 5 describes development of the prototype implementation based on the proposed solution.

A case study demonstrating features of the prototype implementation is presented in Chapter 6.

Chapter 7 presents the performance statistics of the prototype application and discusses achievement of the goals.

Chapter 8 is listing related work.

This thesis is concluded by Chapter 9.

Chapter 2

Background

The goal of this chapter is to describe the problem of checking the software component behavior in more details.

To achieve this, Section 2.1.1 describes behavior protocols in very details and illustrates their application on a simple component system.

Once the behavior protocols of the components in the system are specified, it is able to reason about their compatibility. To verify that all the components employed in the system communicates correctly several checks are used, they are introduced in Section 2.2.

Finally, the reader is thoroughly familiarized with the Java PathFinder, Section 2.3.

In the light of information provided by this chapter, the concluding Section 2.4 revisits the goals of this work.

2.1 Behavior Protocols and Software Components

As already said, the basic idea of the Component-based programming [1, 2] is a composition of the application functionality from individual components. The application is then created by interconnecting specific components and making them cooperate with each other. Since every component provides some services, by choosing sufficient components, the designer can achieve desired functionality of the whole system.

To sustain this approach, components are viewed as black-box entities providing their services through the strictly defined points - interfaces.

To further extend the idea of component system, a hierarchical model is introduced. Here, each component is either *composite* (i.e. it is created as a composition

of lower-level components) or *primitive* (implemented directly in a common programming language, e.g. Java). By creating a composite component from specified components, one can develop a complex component providing the most sophisticated services.

The interfaces of the component are divided into *required* and *provided*. Whereas through provided interfaces, services of the component are accessible, the required interfaces are connected to other components which are used by the component for delegation of tasks.

By the term *environment* we denote all the components directly connected to the interfaces of the component.

For the purpose of component specification, behavior protocols are introduced. A behavior protocol facilitates to the designer to comfortably evaluate whether a specific component fulfils his needs. To precious definition of the behavior protocol specification, the following Section 2.1.1 is dedicated.

2.1.1 Behavior Protocols

Behavior protocols [1] are a platform for component behavior specification. The component behavior is describing a communication of the component with its environment, this communication is representing an effort of the component to fulfil provided services.

From the perspective of the component's implementation, the communication of the component is represented by invocation of methods on the interfaces of the component.

Therefore, behavior of the component is described as a set of admissible sequences of method calls on the interfaces of the component. More precisely, behavior protocol is a regular-like expression syntactically composed from tokens, operators and parentheses.

?m.i↑	Accepting an invocation
?m.i↓	Accepting a response
!m.i↑	Emitting an invocation
!m.i↓	Emitting a response

Figure 2.1: Event Token Variants

An atomic piece of the behavior protocol called *event token* represents an event on interface of a component. Table 2.1 shows four possible events that can occur for a method `m` on an interface `i`.

Operator	Meaning
;	Sequence: a;b means b is performed after a
+	Alternative: a+b means either a of b is performed
*	Repetition: a* means a is performed zero to finite number of times
|	And-parallel: a|b generates all arbitrary interleaving of the sequences defined by a and b

Figure 2.2: Basic Protocols Operators

Figure 2.2 shows the table defining basic protocol operators.

Additionally, there are defined three syntactic abbreviations of method calls, see Table 2.3.

Abbreviation	Meaning	Stands for
!i.m	Issuing a method call	?i.m↑; !i.m↓
?m.i	Accepting a method call	?i.m↑; !i.m↓
?m.i{**expr**}	Processing of a method	?i.m↑; **expr**; !i.m↓

Figure 2.3: Behavior Protocol Abbreviations

Protocol State Space The behavior protocol can be also viewed as an automaton, the words accepted by this automaton represent correct sequences of events occurred on the interfaces of the component. From this point of view, we can introduce a state space representing a particular behavior protocol. Based on this, we introduce the term *trace* which represents the sequence of events from the beginning state to the current one.

Nondeterminism Behavior protocols are also introducing a factor of nondeterminism. The repetition operator can generate an infinite number of finite sequences. This is reflected inside the state space by a presence of cycles.

2.1.2 Example

To show an utilization of behavior protocols in component oriented systems, this Section describes a simple component application borrowed from [9].

Table 2.4 shows behavior protocols of components constituting the application. The architecture of the application under discussion is illustrated on Figure 2.5.

Component	Behavior Protocol
LogDatabase	?db.start; (?db.get + ?db.put)*; ?db.stop
Database	?db.start!lg.log; (?db.get{!lg.log{ + ?db.put{!lg.log})*; ?db.stop{!lg.log}
Logger	?lg.start; ?lg.log*; lg.stop

Figure 2.4: Logging Database Protocols

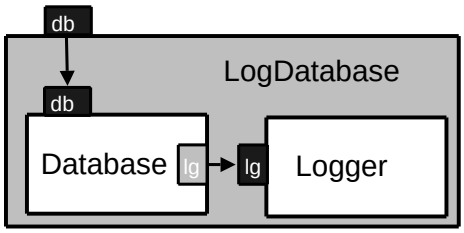


Figure 2.5: Logging Database Architecture

In order to clarify the meaning of behavior protocols, the detail description of the *Database* component follows. The functionality of the component is expressed by its behavior protocol. First, the component accepts only the initialization method call `start`. This leads to calling `start` method on `lg` and then the result of the `start` method call on the *Database* component returns. After that, the component is able to absorb an arbitrary number of `get` or `put` method calls, each results in calling `log` method on the `lg` interface. To finish the execution, the component can be stopped by calling the `stop` method.

2.2 Checking Software Component Behavior

The concept of employing different components brings along the task of checking that applied components are utilized with respect to their behavior protocols.

To verify this, several checks are performed during the application development [9]. The following sections are describing the basic ideas of these verifications.

2.2.1 Compliance and Composition Checks

The key motivation of these checks is to verify that components involved in the system are cooperating with respect not only to their frame protocols, but also to the behavior protocols of other components. To meet this goal, composition and compliance tests are introduced.

Composition test verifies that all components on particular level of hierarchy cooperate with respect to their behavior protocols.

Compliance test is based on comparison of the component behavior with behavior specifications of its subcomponents.

Obviously, these tests verify only behavior specifications of components, component implementations are not taken under consideration. This allows performing the tests at the design stage of the application development, thus providing to the designer an opportunity to identify architecture errors early.

However, the basic assumption of these tests is that implementations of primitive components conform to their frame protocols. In order to guarantee this, an additional test is required.

2.2.2 Primitive Component Check

The motivation for the primitive component check was introduced in the previous section. It is required to verity that a primitive component implementation behaves according to its frame protocol.

To adhere to the black-box concept of communication in component systems, the component cannot make any assumptions about its future environment additional to the interface definition. Therefore, it is needed to verify component implementation against its frame protocol.

Here, an obvious challenge is the need to confront two component specifications, implementation and behavior protocol, which both work on different levels of abstraction.

2.2.3 Behavior Protocol Checker

The Behavior Protocol Checker [10] developed by [11] is a tool used to perform *compliance* and *composition* checks. Informally, behavior protocols are represented inside the checker by their state spaces which are traversed and compared.

For the purpose of the primitive component check was developed a modified version of the protocol checker - Runtime Protocol Checker further referred as *BPChecker*, which contains state space of a frame protocol for a given component. This checker is able to check whether a component violates the frame protocol in a particular run.

2.3 Model Checking

Model Checking [12] is a technique to formal verify a given software or hardware system. This is achieved by checking whether specific properties are hold in the system. However, the state spaces of real-life systems are usually considerably complicated, therefore a simplified model derived from the system is verified.

The goal is to obtain a model which is simplified, but still reasonably reflects the original system and verify that specified properties are satisfied by this model.

To verify a model, it is usually needed to face the state space explosion problem, which represents a situation when even a simplified model cannot be fully traversed in a reasonable time.

2.3.1 Java PathFinder

Java PathFinder [4] is an explicit state software model checker. It verifies given program by traversing its state space and searches for implementation errors (e.g. deadlocks, unhandled exceptions,...) and property violations.

The main feature differentiating Java PathFinder from other model checkers is that it works at a program byte-code level [13]. This means that a real-life application written in a common programming language (Java in this case) is used as a model of a system.

Informally, Java PathFinder (JPF) is a Java Virtual Machine (JPF VM) that executes a given program not only once, like a normal VM, but theoretically in every possible way, thus exploring all execution paths of a program under discussion. During the exploration, JPF is checking for property violations like deadlocks or unhandled exceptions.

JPF provides the extension mechanism that allows to the user to define his or hers own properties that will be checked. The Search-/VMlisteneres techniques can be also employed to gather statistics about the verification process.

Furthermore, implementation of sophisticated heuristics [14, 15] allows to face model checking obstacles, e.g. the state space explosion problem.

For examples of the JPF application see [16, 17, 18].

The following sections introduce some of the key features and heuristics. Besides the JPF documentation [4], introduction to model checking heuristics can be found in [14, 15].

Java PathFinder State Space

The program that is to be verified is represented by its state space inside the JPF. The state space represents all possible execution paths of the program. JPF traverses through this state space and thus verifies the program implementation. To efficiently explore the state space, JPF employs two mechanisms:

Backtracking To explore the state space, JPF uses a backtracking technique to avoid an unnecessary work. Whenever JPF explores an execution path, it backtracks in search for another unexplored execution path.

State Matching Heuristic The state matching heuristic is employed when a new state is reached. JPF checks every new state if it already has seen an equal one. When a state that already has been explored is reached again, JPF backtracks, since all execution paths leading from this state are already checked. Obviously, this contributes to better performance of the verification process.

POR algorithm

Although mechanism for an efficient state space traversing was introduced, JPF still has to face the state space explosion problem.

This problem is inherent to the explicit state model checking, since it is a method that theoretically requires to explore rigorously the whole state space. For example all possible scheduling sequences of thread executions have to be considered.

To face this problem, JPF introduces the POR algorithm [12], which is the algorithm used to reduce the state space in concurrent programs.

The basic idea of this mechanism is to consider context switches only at operations that can have effects across threads. Application of this algorithm causes that transitions between two states can be represented by execution of not only one bytecode instruction, but in fact by several instructions that do not access objects from different threads.

Search Strategies

JPF introduces an option of configuring the most appropriate search strategy for the program that is to be verified. This approach is based on the idea that the whole state space cannot be searched, so the search strategy is directed and defects are found quicker. Of course, this means that the tool is not used to proof the correctness of the program, butit is used as a debugger.

Host VM Execution

Another technique used to keep the state space as small as possible is the Host Virtual Machine execution. Since JPF is a JVM written in Java, there is also other VM on top of which JPF is running. Therefore, parts of the program that are not property relevant can be executed by the Host VM. For example, IO simulation and other standard library functionality is delegated to the Host VM.

2.4 Goals Revisited

In the light of the information provided in this chapter, the goals of this thesis can be specified more precisely:

- **(G1)** A solution has to propose comprehensive answers to the issues of a primitive component verification. The key issues are the full support of all behavior protocol operators and the usage of rigorous methods of the verification in order to guarantee that every protocol violation will be identified. At the same time, the proposal needs to be designed with respect to performance of the future implementation.
  - **(G1.1)** Since the usage of rigorous methods of checking is required, the model checking tools should be considered as a keystone of the future implementation.
  - **(G1.2)** Although a primitive component cannot make any assumptions about its future environment, to verify the implementation it is necessary to simulate every correct activity of the environment; therefore this issue has to be addressed by the solution.
  - **(G1.3)** Furthermore, source code implementation and behavior protocols represent different levels of abstraction, but yet have to be confronted. To achieve this, a proper way of mapping and comparing has to be identified.
  - **(G1.4)** Behavior protocols represent a very strong instrument that is able to express a functionality which cannot be easily defined by the source code, e.g. nondeterminism, but still has to be covered by the solution.
- **(G2)** A prototype application should be implemented in order to demonstrate the functionality of a proposed solution. The application should be able to verify a Fractal primitive component implementation against arbitrarily constructed behavior protocol without the need for additional reductions. Since the performance is also important aspect, the developed tool has to offer an acceptable time of verification.

Chapter 3

Verification of a Primitive Component

The goal of this chapter is to propose an approach to a primitive component verification.

Identification of the most suitable approach is divided into several steps, each targets a specific issue and proposes several concepts of solution. After that, a discussion highlights interesting features and the most fitting concept is chosen.

Going through these steps, the solution is being extended to more and more refined one. And finally, all the crucial issues are addressed.

This Chapter is concluded by a short recapitulation of chosen concepts and then summarizes the proposed solution of the primitive component verification problem.

3.1 Component Verification and Model Checking

To successfully verify that a primitive component adheres to its frame protocol, it is necessary to check every possible behavior of a component against the frame protocol.

For formal verification of a primitive component's behavior, methods of model checking can be used. In fact, the task of a component behavior verification can be easily transformed to a model checking task – verification that a specific property is satisfied by a system under discussion. In this case, the system is represented by a primitive component and the property is defined by a frame protocol of the component.

Based on this observation, two approaches are possible when developing a tool verifying a primitive component implementation.

Brand New Model Checker

One of the possibilities is to develop a new tool that would be able to verify a primitive component implementation. This approach allows focusing on problems of the primitive component verification since the design phase which facilitates to implement the most suitable and straightforward solution. However, the development of a tool that successfully curbs with problem issues of model checking, e.g. the state space explosion problem, is unquestionably out of the scope of this thesis.

Existing Model Checkers Application

When trying to avoid an unnecessary work, it is reasonable to search for a tool that could be applied to this specific type of model checking task. To achieve this, the goal of a primitive component verification can be decomposed into two tasks. First, to explore a primitive component implementation and second, to evaluate events occurring on the interfaces of the component. Whereas for the second task BPChecker (introduced in Section 2.2.3) can be used, for the sake of the first task a model checking tool has to be employed.

Fortunately, JPF model checker appears to be a promising choice. This tool is applying the model checking at the Java byte code level, which facilitates to directly verify component implementations. Moreover, it offers wide scope of features that can be easily extended and modified.

There are also other model checkers, mentioned in Chapter 8, that could be considered. The most interesting is the Bandera tool which is working at the program source code level, but this tool does not appear to be so promising in comparison with the Java PathFinder. Its latest version is not fully stable yet and the support for extensions here is unsatisfactory.

3.1.1 Evaluation

The decision to prefer integration of already developed tools to development of a brand new tool is apparent. JPF and BPChecker represent results of years of development with focus on correctness and efficiency. Therefore, an attempt to employ (with minor modifications) both the checkers is the approach that bring the goal of this thesis to a successful fruition.

3.2 Java PathFinder Application

Although JPF is a tool that applies model checking at the program byte code level, which is the most suitable approach for the software component code verification, there are still issues that have to be faced before beginning with a development of a prototype implementation.

Since JPF was primarily developed to verify low-level properties (see Section 2.3.1) and the behavior protocol can be considered as a quite high-level property, JPF and BPChecker work on different levels of component functionality abstraction.

To face this conflict, it is necessary to define appropriate integration of both the checkers in the verification process. When addressing this issue, several solutions were identified.

Protocol Assertions

The basic idea of this solution is to transform a high level property, represented by the frame protocol, into a low-level one, represented by assertions. Because JPF was designed to verify the assertions written inside the code, this solution does not require any other modifications of JPF. Obviously, the biggest challenge here is to decompose a frame protocol into assertions which can be easily inserted into a component code.

State Spaces Integration

This solution intends to integrate both JPF and BPChecker state spaces into one state space and to adjust JPF to traverse such a state space. Although this idea offers theoretically the most correct approach, which very smoothly brings along solutions of related issues (the environment simulation problem (G1.2), nondeterminism of behavior protocols (G1.4),...), it requires more then significant interventions into JPF implementation.

JPF and BPChecker Cooperation

This approach is based on coordinated exploration of the state spaces. JPF traverses the state space of a primitive component and whenever it detects communication on the interfaces of the component, BPChecker is requested to verify that the detected communication events are conforming to the frame protocol.

Therefore, this approach requires to introduce a cooperation between JPF and BPChecker tools. Moreover, it is necessary to define proper mapping between state spaces of the integrated tools (see also (G1.3)).

Despite these requirements, this solution provides a straightforward way to achieve the goal.

3.2.1 Evaluation

Although the first two approaches seem to be promising, they are based on theoretical ideas and require significant interventions into JPF implementation.

In contrast, the solution introducing cooperation between the tools is a result of pragmatic approach which brings required features. The demand only for minor modifications of JPF can be highlighted as a second contribution of this approach

From this perspective, it is reasonable to choose the *JPF and BPChecker Cooperation* as the most suitable solution.

3.3 JPF and BPChecker Cooperation

Since JPF is able to verify only closed systems, the most crucial problem that has to be faced when using it to verify a primitive component implementation is an incompleteness of a program that is to be verified (G1.2).

From JPF point of view, a primitive component represents only a fragment of a software component application and the checker is not able to identify an explicit starting point of the verification process (e.g. the `main` method).

This problem was also identified in [8] where it is called the *Missing Environment Problem*.

The concept employed when solving this obstacle is the fundamental point determining the approach of the whole solution. When focusing on the solution of this problem, three main options were identified.

Virtual Environment

The key idea of this concept is to generate an environment of the component, thus creating a closed system that can be verified by JPF. The most challenging part is the environment generation, since such environment has to guarantee that every correct sequence of events absorbed by the component can be issued from the environment's implementation code. After that, the process of verification requires only to evaluate how the component responds to the absorbed events.

Generic Class

This solution tries to get along without the environment generation. It introduces a generic structure managed by the frame protocol that is able to wrap the component with intention to create a closed system. Such a generic structure has to be able to both fully supersede the environment and remain flexible and reusable for every component implementation.

JPF Modification

Similarly to the previous one, this concept is not generating the environment. The fundamental point of this solution is to adapt JPF to the software component verification. The goal is to modify JPF to be able to verify only the component itself and to locate procedures managing the communication between the component and its environment outside the JPF state space.

Figure 3.1 shows schemas of the presented concepts.

Since each concept involves different challenges, it is important to discuss their advantages and to evaluate their potentials. In Sections 3.3.1, 3.3.2 and 3.3.3 these concepts will be described in greater detail.

In conclusion 3.3.4 of this Section these concepts are confronted and the most suitable one is chosen for the implementation.

3.3.1 Virtual Environment

As said, the key idea is to generate an environment of the component, thus creating a closed system that can be verified by JPF. The most challenging part is the environment generation, since such environment has to guarantee that every correct sequence of events absorbed by the component can be issued from the environment's implementation code. After that, the process of verification requires only to evaluate how the component responds to the absorbed events.

In fact, the generated environment is not the standard component environment as defined in Section 2.1, the generated environment only contains a code, which is generated from a given frame protocol and is wrapping the component in order to create a closed system. Therefore, the term *virtual environment* is more suitable here.

The schema illustrating architecture of this solution is shown in Figure 3.2. A component under discussion is represented by the blue square, the green squares represent the virtual environment. The whole system is verified by JPF, highlighted by the red color. The cooperation with BPChecker is expressed by the black arrow connecting JPF with the Protocol Checker implementation.

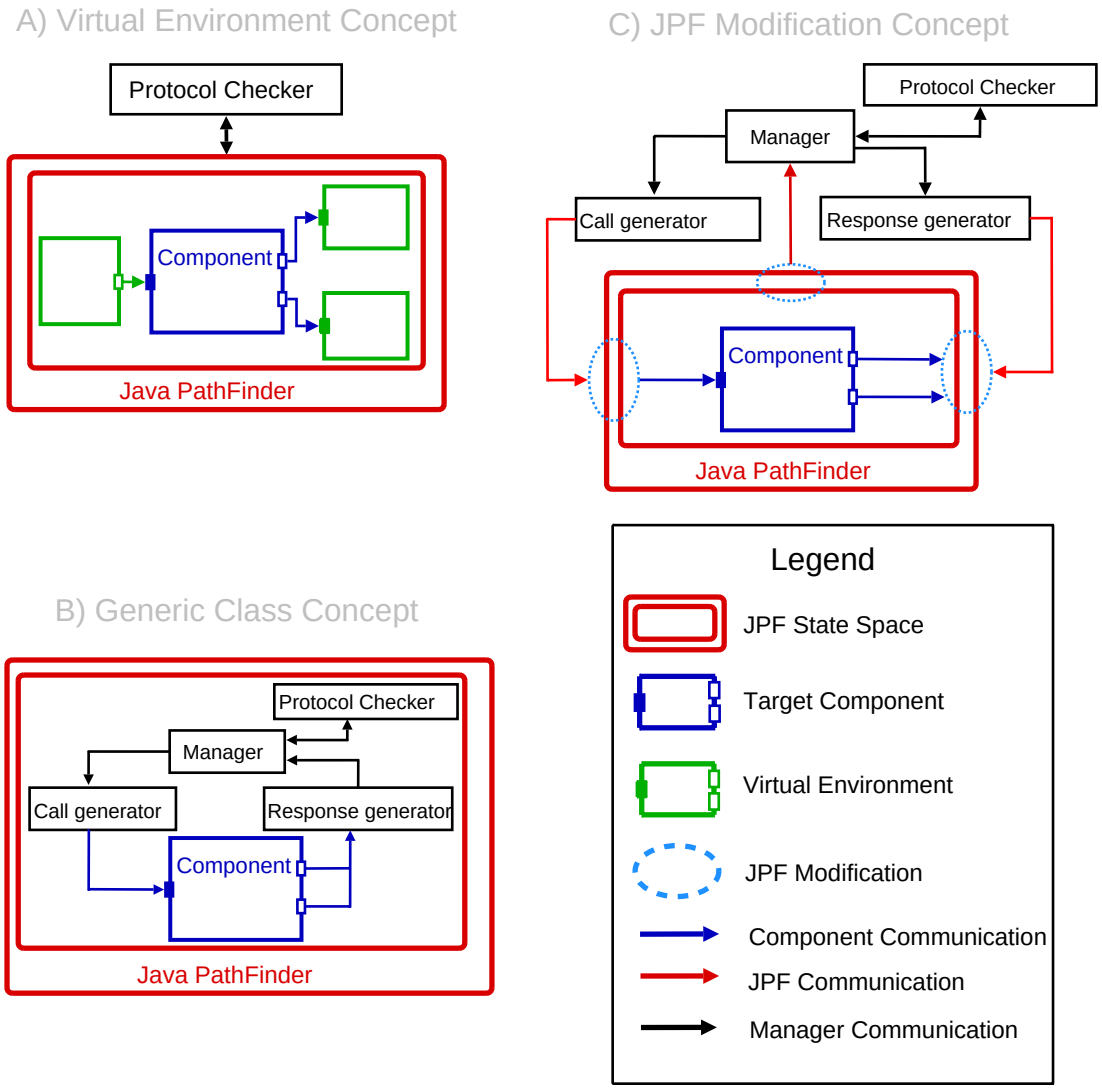


Figure 3.1: JPF and BPChecker Cooperation, Proposed Concepts

The Process of Verification

The Environment Generation Phase As already said in 2.3.1, JPF is not able to model check an isolated component. Therefore, it is necessary to generate an environment that would, along with the component, form a closed system which can be processed by JPF.

A generated environment has to fulfil several requirements. First, the composition of the component and the generated environment has to form a complete program with an explicit starting point (e.g. the `main` method). Second, the implementation code of the generated environment has to be able to communicate with the

component - issue events on the component's interfaces and absorb the events emitted by the component. And finally, the implementation of the environment has to be able to generate every sequence of events that is in compliance with the frame protocol of the primitive component. This guarantees that JPF will be able to traverse the code of the primitive component exhaustively.

A suitable environment can be generated from a frame behavior protocol, however, the data values of the method parameters have to be also specified. This brings the need to carefully define sets of possible values for method's parameters, otherwise JPF could not be able to check all of the control flow paths in the component.

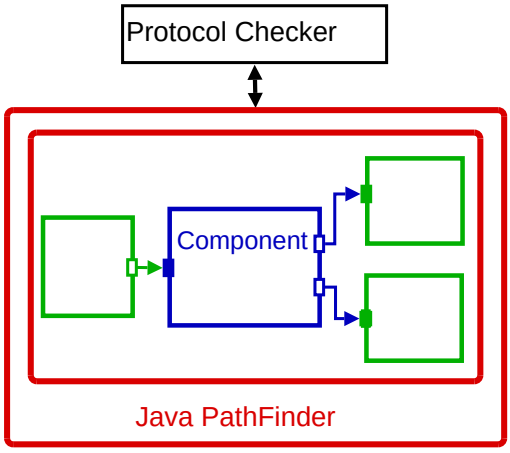


Figure 3.2: Virtual Environment Schema

The Verification Phase Once the virtual environment is generated, JPF is able to check the whole program composed of an environment and the component.

The backtracking techniques implemented by JPF allow to explore every execution path of the code. Naturally, the environment implementation has to be adapted to such an advance of the verification process.

By employing the Search-/VMListener techniques BPChecker can be notified about the communication between the component and its environment. The integration of BPChecker provides a sufficient way to detect violations of the behavior protocol of the component.

Advantages

The biggest advantage of this solution resides in no JPF modifications. The whole program, formed by the component and the virtual environment, can be easily verified by the tool without any additional effort.

Disadvantages

An obvious drawback of this solution is the necessity to generate an environment. While this concept allows to relatively simplify the process of verification, problems related to the verification of software components are not solved, they are only moved to the environment generation phase, where they have to be faced again.

Since the verified program contains not only the component implementation, but also the environment implementation, the state space traversed by JPF is much larger, which adversely affects the performance of the solution.

Another factor contributing to lesser performance is a need for generation of a new environment for every component - frame protocol pair.

Although the key motivation for this concept is to avoid Java PathFinder modification, the complexity of the software component verification still requires some modifications of the tool.

3.3.2 Generic Class

This solution tries to get along without the environment generation. It introduces the generic structure managed by a frame protocol that is able to wrap a component with intention to create a closed system. Such a generic structure has to be able to both fully supersede an environment and remain flexible and reusable for every component implementation.

The Verification

The key idea of this solution remains similar to the previous one - to obtain a closed system that can be verified by JPF. However, this concept brings along several improvements in meeting this goal.

The main enhancement resides in the implementation of the structure which is wrapping the component. The motivation here is to avoid creation of an environment whenever a frame protocol of the component is changed. Therefore, the generic structure called Wrapper is introduced.

Wrapper As already said, Wrapper, structure that encapsulates the component, has to fulfil two basic requirements: full replacement of the missing environment and reusability for every component. These ambitious demands bring along several challenges.

An effort to replace an environment requires from the Wrapper the ability to establish a communication with the component. Obviously, the mechanism for

communication with the component needs to be generic enough to communicate only on the basis of the frame protocol specification. Fortunately, this requirement can be met by employing the Java Reflection API [19].

The Wrapper also needs to be implemented with respect to the Java PathFinder search strategy in order to enable exploration of every possible execution path of the component implementation. This demand leads to necessity of BPChecker integration inside the Wrapper. The cooperation with BPChecker allows to Wrapper to simulate every possible behavior of a component's environment.

The Verification Process The principle of the verification process is based on the idea that Wrapper is able to fully replace an environment and that the JPF tool is forcing Wrapper to communicate with the component, thus executing code of the component.

The implementation of Wrapper is designed with respect to the backtracking techniques employed inside JPF, therefore the tool is able to explore the whole state space representing the component implementation.

The Search-/VMListener techniques are again employed to gather statistics about the verification process and the designer is able to evaluate the progress of verification and to detect violations of the frame protocol.

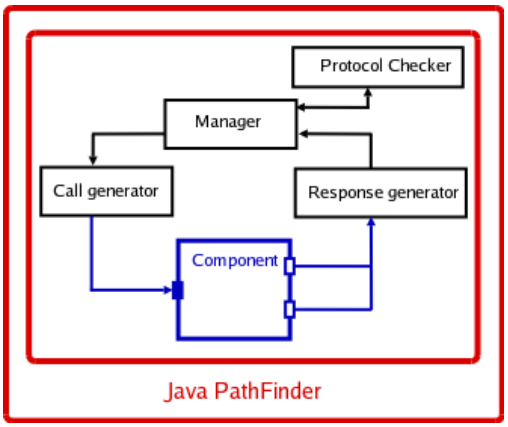


Figure 3.3: Generic Class Schema

Figure 3.3 illustrates the schema of this solution. Again, the component under discussion is represented by the blue square which is wrapped by a generic class structure. An obtained system is then verified by JPF.

Advantages

The motivation for introducing this solution is to overcome the obstacle of an environment generation and at the same time to avoid JPF modification.

The desire to cope with the necessity of environment generation resulted in introduction of Wrapper, which is suitable to fully replace an environment. Additionally, Wrapper provides the generic approach and therefore can be used for every component implementation without the need for other modifications.

Also the demand for no JPF modifications is respected by this solution.

And finally, the reusability of Wrapper is a highly potential aspect, which should be featured in every solution that attempts to come to a successful fruition.

Disadvantages

Similarly to the Virtual Environment concept, JPF modifications are required.

Although the Generic Class concept avoids an environment generation, this goal cannot be achieved entirely. The Wrapper still has to generate the input/output parameters of methods invoked on interfaces of the component. This problem is further referred as the *Value Generation problem*. To overcome this drawback, the user has to define a set of possible values which will be considered as input/output parameters of methods. Obviously, this brings the need to carefully define sets of possible values, otherwise the Java PathFinder tool could not be able to check all the control flow paths in a component.

When pursuing abilities of this solution, one has to carefully investigate Wrapper implementation. Since Wrapper is the control unit of the whole program that is going to be verified, its implementation has to sufficiently replace an environment and still allow an efficient verification of the component code. These requirements are highly contrasting with the description of Wrapper implementation in Section 3.3.2 which demands the BPChecker integration.

Integrating BPChecker consequently means that the state space of JPF has to comprise the state space of BPChecker. Since JPF is inherently traversing the whole state space, by including the Wrapper implementation the performance of this solution is adversely affected.

Although this solution seemed to be very promising, the necessity of the BPChecker integration inside the Wrapper brings the goal of an efficient implementation beyond the scope of successful feasibility.

3.3.3 JPF Modification

Similarly to the previous one, this concept is not generating the environment. The fundamental point of this solution is to adapt the JPF to the software component verification. The goal is to modify JPF to be able to verify only the component itself and to locate procedures managing the communication between the component and its environment outside the JPF state space.

This concept is inspired by the solution introduced in 3.3.2, keystones remain the same: to avoid an environment generation and to design a controlling unit, which will be flexible and reusable for different components.

The significant characteristic of the previous concepts was an effort to avoid JPF modification. This effort brought along several approaches that led to lesser performance and revealed issues, which in most cases could not be solved satisfactorily. Furthermore, despite the motivation to preserve original JPF, modifications of the tool were still inevitable.

Under these circumstances it is appropriate to break the basic assumption of no JPF modifications. In addition, the key desire is to introduce such a modification of the JPF core that it will allow reaching better performance. It is also anticipated that some problems identified as open issues in the previous concepts will be solved.

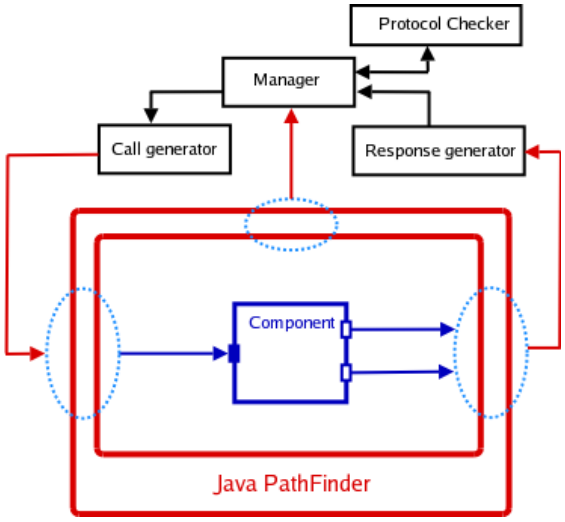


Figure 3.4: JPF Modification Schema

Figure 3.4 shows the design schema of this solution. It can be seen that only the component that is to be verified resides inside the JPF state space, the remaining units are placed outside the JPF.

Manager

The idea of the component encapsulated by a control unit, which is simulating an environment, was further extended in this solution. The control unit is called Manager and its task is not only to replace an environment, but also to manage the process of component verification.

In order to overcome obstacles mentioned in the Generic Class concept, it was necessary to deploy the control unit outside the JPF state space. This is the key advantage of the solution which brings along the challenge of the JPF modification.

JPF Modification

Key purposes of these modifications are to adapt JPF to a communication with Manager and to adjust the JPF exploration strategy to a state space representing a component implementation.

To control the verification process, Manager has to be thoroughly informed about the progress of verification. Although the notification techniques can be easily integrated into Manager, they are not sufficient, especially for the purpose of a direct modification of the JPF state, and need to be extended.

Another major modification concerns the exploration strategy of a state space which represents the component code. Since a component does not form a closed system, the exploration strategy has to be adjusted and whenever JPF wants to reach a code that is unavailable, a situation is reported to Manager.

The Verification

During the verification, Manager is being notified by JPF about the current state of the verification process and by evaluating this state it determines next steps.

The integration of BPChecker guarantees that Manager is able to simulate every possible behavior of a component environment. Additionally, it is necessary to force the Java PathFinder to explore all execution paths of a component code. To achieve this goal, the concept of backtracking employed inside the JPF has to be implemented also by Manager.

Advantages

Introduction of Manager in association with JPF Modifications allows to efficiently avoid the Missing Environment Problem. This concept provides a generic approach, which can be applied to every correct implementation of a software

component. Moreover, no additional preprocessing step is needed, a verification process can be started immediately.

When discussing the possible performance of this solution, it is necessary to highlight the reduction of a state space which is introduced by this solution. JPF is traversing only the state space representing the code of a component which is the result that inherently cannot be achieved by any of the previous concepts.

From other benefits the possibility of directly controlled progress of verification stands out. Manager can interrupt this process at any time and force JPF to explore another execution path, thus affecting the progress of a verification. This gives us an ability to control the verification and to smoothly integrate additional heuristics.

Disadvantages

The necessity of JPF modification is an obvious challenge. The JPF tool itself is an ambitious project dealing with numerous problems. Therefore, every intervention into such a complex structure needs to be carefully considered with respect to possible side effects of such a modification.

The problem of the Value Generation needs to be also addressed by this concept. Unfortunately, there is no silver bullet and the introduction of Value Specifications seems to be the only solution.

3.3.4 Evaluation

Previous three sections introduced different concepts of cooperation between JPF and BPChecker. Each of them employs different approach and reveals specific problems concerning implementation of the solution.

The goal of this section is to summarize the proposed solutions, highlight interesting and valuable issues and finally determine the most suitable and promising advance.

Although the presented proposals intend to provide transparent approaches, the complexity of a software component verification brings along various problems that prevent from nomination of the perspicuous solution. The drawbacks mentioned in Sections 3.3.1, 3.3.2, and 3.3.3 make the goal of identifying the best solution very complicated.

Discussion

As already indicated, the most crucial problem when using JPF to verify a software component is the lack of environment. The proposed solutions introduce different

issues facing this obstacle, but without solution of the Value Generation problem this goal can not be met satisfactorily. Therefore, the Value Generation problem is common to all proposals.

Despite the motivation to preserve the original JPF implementation, a detailed analysis of possible approaches showed that some modifications of the JPF are inevitable.

Furthermore, the Generic Class concept introduces an interesting issue - reusability of a control unit, which is a very promising feature. Unfortunately, the successful implementation is not possible due to technical problems concerning integration of BPChecker inside the Wrapper.

Under these conditions, only two solutions can be taken into consideration. While the Virtual Environment concept still keeps the motivation to avoid JPF modifications, the JPF Modification concept chooses the generic approach and proposes modifications of the JPF core.

And The Winner Is

With possible options introduced, the ultimate choice has to be made. In the light of the outlined advantages, the JPF Modification concept was chosen to implement. The possibility to verify only the state space of a component together with a generic approach are very attractive features which deserve our attention.

Although JPF modification can be highly potential when concerning the direct control of the verification process, one has to carefully consider which modifications can be introduced. The Java PathFinder Model Checker represents a complex application that can be easily affected by insensitive interventions. Obviously, the detail knowledge of the JPF implementation is essential for this solution.

3.4 Summary

The goal of this Chapter was to identify possible approaches of a primitive component verification.

At first, the task of a primitive component verification was elaborately considered and the analysis revealed that model checking provides techniques which can be employed by a solution. Additionally, JPF and BPChecker tools were identified as keystones of the future prototype application.

After that, it was necessary to precisely define an utilization of both the checking tools in the verification process. This lead to the introduction of the *JPF and BPChecker Cooperation* strategy defining the basic principles of the collaboration between both the checkers.

The challenge of integrating JPF and BPChecker in the verification process was further described in Section 3.3 which introduced different concepts of the *JPF and BPChecker Cooperation*. These concepts were later confronted and the most suitable one was chosen to be applied.

As a result, this Chapter proposes an approach to a primitive component verification applying the techniques of model checking. The concept of *JPF and BPChecker Cooperation* is proposed to become a keystone of a new model checking tool - *Software Component Model Checker*.

Chapter 4

Software Component Model Checker

This Chapter presents the Software Component Model Checker – the solution to a primitive component verification.

In order to introduce all aspects of the solution in detail, the Chapter is structured as follows.

Section 4.1 is identifying the basic ideas of the solution and provides to the reader essential, but still simplified orientation in issues employed by the solution.

In Section 4.2 the basic ideas are further extended and exhaustively described. This Section provides all information necessary to fully comprehend the approach engaged in the software component verification.

Section 4.3 presents flexibility and efficiency of the solution when facing the crucial issues of the software component verification.

In the conclusion (Section 4.4), the results of this chapter are discussed.

4.1 Basic Ideas

The solution employing JPF and BPChecker was chosen as the basic idea of the software component verification. JPF is traversing the state space of the component implementation and the behavior of the component is evaluated with an assistance of BPChecker. The key feature is to arrange a proper way of cooperation between both the checkers. This is further addressed in Section 4.1.1.

Once the basic strategy of the component verification is specified, one has to face the biggest challenge of the primitive component verification process - the simulation of the component's environment (G1.2). As already indicated in Section

3.3, to solve this problem, JPF Modification concept will be employed. Section 4.1.2 describes this approach.

4.1.1 Cooperation

The key motivation of this solution is to force the employed tools to cooperate with each other on the evaluation and control of the verification process. JPF is exploring the state space defined by the component implementation and whenever an event is detected on the component's interface, BPChecker is asked to verify correctness of the event.

This requires to keep both the checkers synchronized. Since each tool is working on a different level abstraction (JPF with instructions and BPChecker with events), it is necessary to define a proper mapping between their state spaces (G1.3). This would be possible if the states representing the invocation of methods on the component's interfaces could be identified. Whereas this is inherently satisfied inside the BPChecker state space, the JPF state space is representing only the component itself and therefore it is not complete.

To face this problem, it is necessary to extend the JPF state space in order to include also the states representing a communication between the component and its environment. The Solution is described in Section 4.2.1. The approach of the *State Space Mapping* is elaborately addressed in Section 4.2.

4.1.2 Environment Simulation

The JPF Modification concept introduces a controlling unit called Manager which simulates activity of a component's environment. In order to avoid drawbacks presented in Section 3.3.2, the controlling unit is deployed outside the JPF state space. Although this brings along a feature of representing only the component itself inside the JPF state space, the price paid for this advantage is a necessity to modify the core of the JPF.

Because the environment comprehends the component as a black-box entity, it can issue events on the interfaces of the component potentially at any time, of course with respect to a given frame protocol. Therefore, the key challenge when replacing the environment is to ensure that Manager will be able to simulate the occurrence of an event in any moment of the verification process.

Since the POR algorithm (see Section 2.3.1) allows to divide the verification process into steps, the execution of the component code is represented as a sequence of JPF states. Based on this observation, the task of issuing event on interfaces of the component at any time can be reformulated. It is correct to demand on

Manager to be able to simulate occurrences of events only in every state of JPF VM.

4.1.3 Getting Whole Picture

The central unit of the whole verification process is Manager. It communicates with both the checkers and determines the future step of the verification. Also the cooperation between JPF and BPChecker is arbitrated by Manager.

The significant role of Manager is also taken in the environment simulation process. Manager evaluates states of both the checkers and decides which events will be simulated on the interfaces of the component. Whereas to issue events absorbed by the component, Manager cooperates with BPChecker, to detect events emitted by the component, the cooperation with JPF is essential.

Informally, we can say that JPF is representing the component, BPChecker is representing the environment and Manager is the connecting layer between them.

The following subsections are highlighting the key features of each element employed in the prototype application. For details concerning the prototype implementation, see Chapter 5.

JPF

JPF contains the state space representing a component under discussion. This state space is being traversed in the search for behavior violations.

As a cornerstone of this part of the implementation was used Java PathFinder Model Checker [4] which was further extended and adjusted to the verification of software components.

Manager

Manager is simulating environment of a component under discussion. To achieve this, it is thoroughly informed by both the checkers about their current states.

To reflect the exploration strategy of JPF, Manager is integrating the Manager Stack, a structure that is introduced in Section 5.2 in detail.

BPChecker

For the purpose of the prototype implementation, a Runtime Protocol Checker, referred as BPChecker, developed in [11] was used.

Similarly to JPF, BPChecker also contains the state space. This state space is defined by a frame protocol of the component. During the verification process BPChecker is used for two tasks. First, to evaluate whether an event emitted by the component is conforming the frame protocol of the component. And second, to provide for every state of BPChecker a list of correct events that can occur on the interfaces of the component.

To fulfil these requirements, BPChecker has to be able to traverse its state space.

Addressing challenges of software component verification has lead to extensions of the BPChecker functionality, see Sections 4.3.5 and 5.3 for greater detail.

4.1.4 Verification

To fully comprehend a complexity of the verification process, this section introduces this process in greater detail.

The verification proceeder can be divided into two parts.

Initialization

Before the verification can be started, two tasks have to be accomplished. First, it is necessary to create a representation of the component inside the state space of JPF. And second, a starting point of the verification process needs to be defined.

Just to remind, the software component is defined as a class. Therefore, obtaining a representation of the component inside JPF means to create an instance of this class inside JPF VM.

For the sake of the second task, the cooperation between Manager and BPChecker is essential.

The component represents services which are provided and required. The usage of a required service is inherently connected to the usage of some provided service. Therefore, JPF should start with execution of the component's code whenever a required method is invoked.

From this perspective, potentially every provided method can be a starting point of the verification process. It is therefore needed to choose those provided methods which are also starting in the meaning of a given frame protocol. Manager is then responsible that every correct starting point will be considered by JPF.

The process of the initialization is further described in Section 5.1.1.

Verification

JPF explores the state space representing the component, thus verifies its implementation.

In order to guarantee that Manager is able to simulate the occurrence of an event, JPF notifies Manager whenever it reaches a state. The advanced state is evaluated and Manager decides with cooperation of BPChecker whether an event will be issued. The process of inserting an event into the state space of JPF is described in Section 4.2.3.

The events emitted by the component have to be also considered by the solution. An event is emitted when JPF executes instructions representing an invocation of a method on a required interface. At this point, the execution has to be interrupted and Manager is informed about the invoked method. BPChecker has to verify that the event is in a compliance with the frame protocol. The thread invoking the required method is interrupted until the moment when Manager decides that an event representing the response to the invoked method can be issued. This fundamental modification of JPF is more precisely addressed in Section 4.2.3.

During the verification, Manager evaluates the states of JPF and BPChecker and decides whether there is an event that can be simulated. Of course, for one state of JPF there can be several different events, which can occur on the interfaces of the component. The key desire is to be able to issue every correct sequence of events on the interfaces of the component. To achieve this, it is necessary to implement backtracking techniques also inside the Manager. Section 4.7 introduces all steps involved in the backtracking process.

4.2 Advanced Topics

This section is further describing already presented issues.

Before explaining the approach to the environment simulation, it is needed to introduce in detail the process of mapping between JPF and BPChecker state spaces. To this purpose, Sections 4.2.1 and 4.2 are dedicated.

When the principle of cooperation between JPF and BPChecker is thoroughly described, the approach of the environment simulation can be presented. See Section 4.1.2.

It is also important to focus on the backtracking process which is fundamental for exploration of every possible execution path, see Section 4.7.

4.2.1 State Space Extension

When verifying a software component, we have to deal with a system which is not closed. This drawback is inside JPF represented by an incompleteness of the state space. To enable that the component can be verified, we have to implement extensions of the state space in order to make it traversable.

As described in 2.3.1, transitions in the JPF state space represent instructions executed by JPF VM. To express also communication with an environment inside the state space, a concept of transitions representing only instruction execution needs to be extended. Therefore, we introduce a second type of a transition - the transition representing an event occurred on an interface of the component.

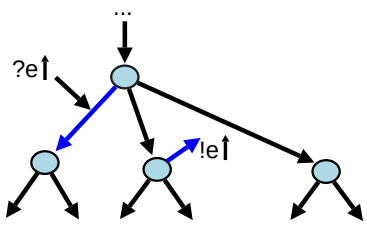


Figure 4.1: JPF State Space Extension

Figure 4.1 shows the extended state space. Transitions inserted into the state space by Manager are denoted as the blue arrows. The event absorbed by the component is represented by the blue arrow connecting two states. The second blue arrow is denoting the event emitted by the component, it represents a sequence of transitions leading to the invocation of a required method.

4.2.2 State Spaces Mapping

The task of the state space mapping is one of the essential requirements of the solution (G1.3). The approach to this, based on the cooperation of JPF and BPChecker, requires that it is possible to map the state of JPF into the state space of BPChecker.

This requirement can be easily met once the state space of the Java PathFinder tool is extended. The extended state space is representing not only instructions executed, but also the emitted/absorbed events. Thanks to this, it is possible to create mapping between transitions from the JPF state space and transitions from the BPChecker state space.

The State Space Mapping is illustrated on Figure 4.2, which shows JPF and BPChecker state spaces.

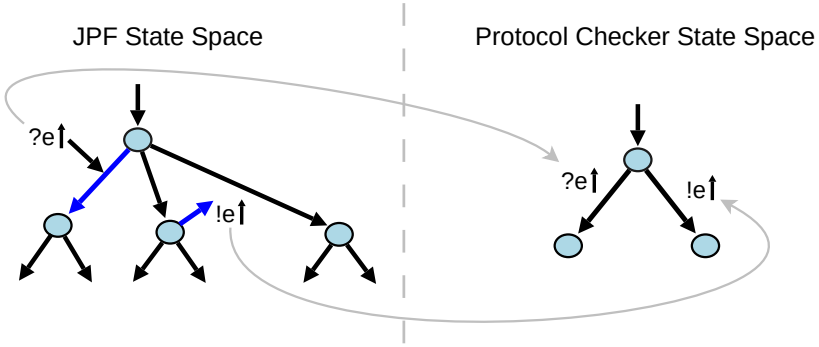


Figure 4.2: State Space Mapping

4.2.3 Environment Simulation

Once the proper mapping between state spaces is defined, the biggest challenge of the software component verification, the environment simulation, can be faced.

The goal of this section is to provide detailed look at the environment simulation process. Following text describes steps necessary to simulate events occurred on the interfaces of the component.

When simulating the environment, one has to strongly distinguish between the events emitted by the component and the events absorbed by the component. While the emitted events are direct results of the code executed by JPF, the events absorbed by the component are generated by Manager. Therefore, two tasks of the environment simulation problem can be identified:

- Inserting Event - Simulating an event occurred on an interface of the component
- Detecting Event - Detecting event emitted by the component

Inserting Event

This section describes the process of inserting an event. When Manager, in cooperation with BPChecker, decides that the event can be absorbed by the component, it simulates an occurrence of this event on the component's interface - inserts the event into the JPF state space.

Procedure The first step of the process of inserting an event is depicted in Figure 4.3. JPF has advanced a state and notifies it to Manager. The task of Manager is to simulate the environment. Therefore, it requests BPChecker for the

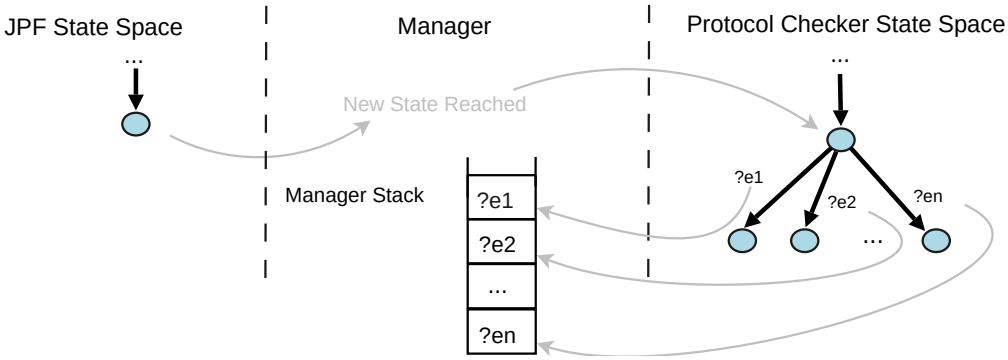


Figure 4.3: Inserting Event, 1st step

list of events which can occur on the interfaces of the component in this state. If there are such events, they are stored in the Manager Stack in order to be inserted into the JPF state space.

Now, Manager disposes list of events which can be inserted into JPF. As a second step, the first event from the Manager Stack is inserted into the JPF state space, thus creating a new state. BPChecker has to be also kept in synchronization, therefore, it is notified that the event was inserted and proceeds to the corresponding state. Situation describing the second step is shown on Figure 4.4.

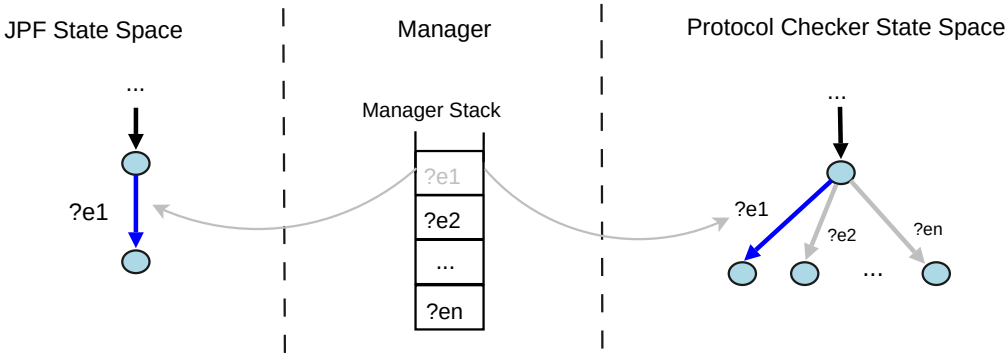


Figure 4.4: Inserting Event, 2nd step

Value Generation Absorbing an event is an abstraction for a communication process which transforms a data to the component. These data has to be also inserted into JPF. This problem is targeted by Section 4.3.1.

Backtracking As can be seen on Figure 4.4, there is more than one event to be inserted at the time. Fortunately, JPF implements backtracking techniques

which facilitates to simulate every possible occurrence of events on the interfaces. Also Manager is implementing backtracking techniques with intention to simulate every correct possibility.

Influence of backtracking techniques on the JPF state space can be observed on Figure 4.5. Picture shows the JPF state space when all events were inserted, denoted as the blue arrows. The black arrows represent the execution paths when no events was inserted in this state, instead, JPF has executed instructions. It can be easily seen that this approach is generating every possible execution path, thus allows to explore the state space of the component exhaustively.

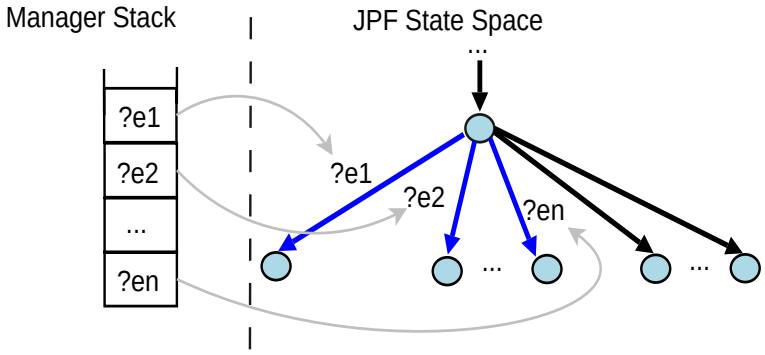


Figure 4.5: Events Inserted

Request and Response The behavior protocol formalism differentiates two types of absorbed events. An absorbed *request* means that the method on a provided interface is being invoked. An absorbed *response* represents a returning value of a required method invocation. Obviously, requests and responses modify a JPF state differently.

Inserting Request A request represents a thread which is going to execute a method on a provided interface of the component. It is therefore necessary to create a new thread inside the JPF VM which will execute the desired method. To achieve this, a process of thread creation, implemented by the original JPF, has to be significantly modified, see Section 5.1.4.

Inserting Response A response represents a return value which has to be assigned to the corresponding thread that invoked the required method. Therefore, it is assumed that some thread inside JPF VM has invoked a required method and now is waiting for the response. This problem is also related to the parallelism problem described in Section 4.3.3.

Detecting Event

As already said, it is necessary to pay attention to the events emitted by the component; therefore, this section introduces the process that detects emitted events.

An event is emitted when JPF executes instructions representing an invocation of a required method. Since the code implementing the required method is not available, the invoked method cannot be executed and this situation has to be solved differently.

In order to delegate the solution of this situation to Manager, the process of instruction execution has to be interrupted. This consequently means that a new state, representing a required method invocation, is created. This is also related to the POR algorithm, see Section 5.1.2.

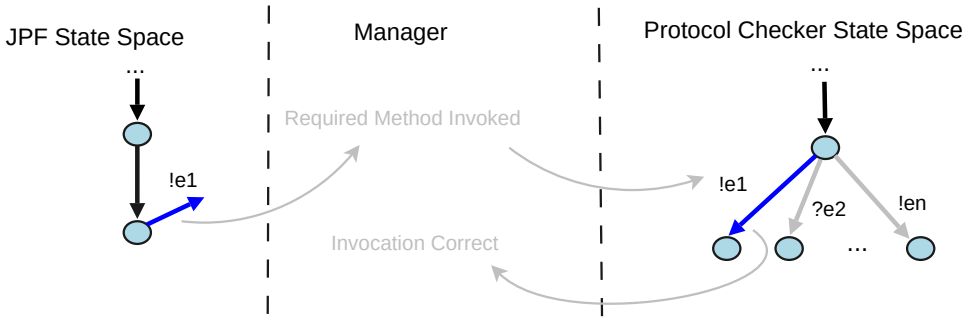


Figure 4.6: Detecting Event

Once Manager is notified about an emitted event, see Figure 4.6 depicting this situation, it asks BPChecker, whether this event obeys the frame protocol. If no, a protocol violation is reported, the current state of JPF is marked and JPF starts with backtracking in order to explore another execution path. If an emitted event is correct, the state of BPChecker is updated and JPF can proceed with the verification.

Again, when an event is emitted, it requires some administration, which differs according to the type of an event.

Response Detected Emitting a response represents a returning call to the method invoked on a provided interface of the component. Execution of a provided method has finished, result is being returned and a thread, which was executing this method, is going to be terminated.

Request Detected Component emits a request when instructions representing required method are executed. Because at this point is not known when a response will be absorbed, a thread, which executed a method invocation, has to be stopped. This thread is started again when the response to this call is absorbed.

State Space Extension For the purpose of verification, it is necessary to force JPF to respect a required method invocation and to create a new state for every required call. This is done by a modification of the POR algorithm, see Section 5.1.2.

4.2.4 Backtracking

JPF employs the backtracking technique in order to explore every execution path. Since Manager is cooperating with JPF, it is necessary to adapt Manager to this approach.

Additionally, Manager has to keep BPChecker synchronized. Therefore, BPChecker has to be able to traverse its state space in both directions and to restore the previous state, when it is required.

Backtracking JPF State When JPF is backtracking from a state which was generated by JPF itself there is not necessary to notify Manager. The transition being backtracked is representing an execution of instructions which do not express a communication with the environment. Therefore, there is no corresponding event which could be backtracked inside the BPChecker state space.

Furthermore, it is not necessary to ask BPChecker whether there are any events that can occur on the interfaces in the restored state, because, according to the description of the inserting event process in Section 4.2.3, all execution paths in which an event occurred in this state were already explored.

Backtracking Inserted State The situation, when JPF is backtracking state that represents an event which occurred on a component's interface, requires a special handling.

Figure 4.7 illustrates the backtracking of an inserted event. When JPF is backtracking this state, it reports to Manager that state which represents an event is being backtracked. To keep BPChecker synchronized, Manager instructs BPChecker to backtrack the event and to restore the previous state. Since the event was previously suggested by BPChecker to be inserted, the action of backtracking an event is always correct.

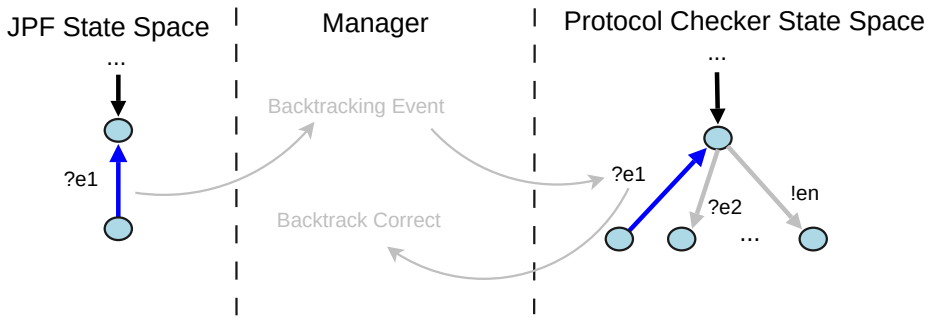


Figure 4.7: Backtracking Event

Advancing End State Special situation occurs when JPF reaches a state which represents an ending point of an execution path. Before JPF can backtrack from this state, it has to notify Manager that the end state was reached. Manager asks BPChecker whether its current state is also the ending point. If yes, JPF can backtrack, otherwise the protocol violation is reported.

4.3 Addressing The Issues

Since all issues describing the approach employed by this solution were introduced, it is possible to discuss the abilities of the solution and confront them with questions identified as crucial issues of the software component verification.

By presenting approaches to the most delicate problems of the software component verification we want to demonstrate the features and flexibility of this solution.

4.3.1 Value Generation

The value generation problem was identified when facing the challenge of the environment simulation, to which it is related.

Problem Statement

Although JPF Modification concept, introduced in Section 3.3.3, avoids the environment generation, this goal cannot be achieved entirely. There still remains demand for the value generation of the input/output parameters of methods invoked on the interfaces of the component.

In other words, Manager has to be able to generate values of input parameters for every provided method and generate return value for every required method.

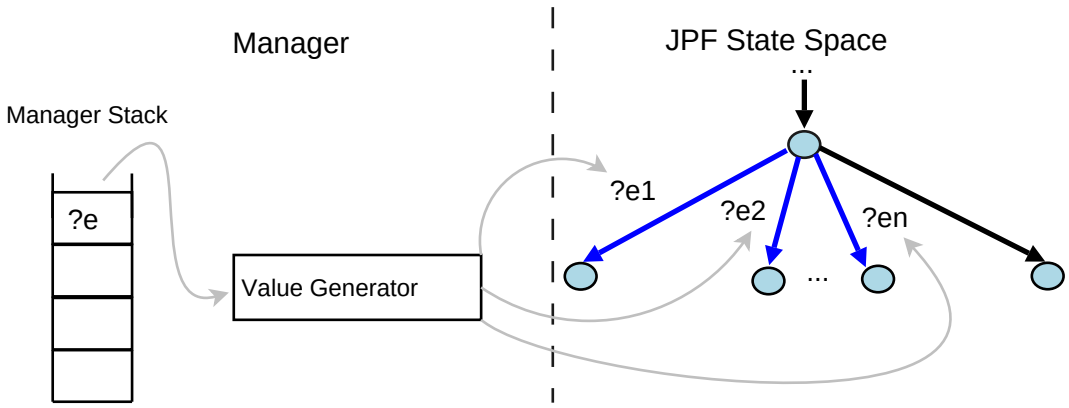


Figure 4.8: Value Generator Application

With the motivation to simulate thoroughly an activity of the environment, this goal is becoming unfeasible.

Problem Solution

Since there is no silver bullet, at least partial solution should be identified.

In order to overcome this obstacle, every event contains the value generator that is able to generate a set of possible values which are to be considered as parameters.

However, such a solution requires a value specification to be defined. For the purpose of the value specification, the component description file was introduced. The user can define for every parameter of a required/provided method the set of values. The value generator guarantees that every possible combination of parameter values for each method will be considered.

The usage of the value generator slightly modifies the process of inserting an event (described in Section 4.2.3). One event is inserted into JPF not only once but once for every possible set of parameter values. Obviously, the range of possible values for parameters directly affects the state space size, since each selected set of values triggers a different branch in the state space. Such a situation is illustrated on Figure 4.8.

4.3.2 Spearhead Problem

Spearhead problem was identified in [8] for the first time. Although authors are employing a different approach to the software component verification, this problem needs to be addressed by this solution as well.

Source Code	Behavior Protocol

`if(...) {`
`  a();`
`  b();`
`}`
`else {`
`  c();`
`  d();`
`}`
`e();`
	`(a ; b ; e )`
	`+`
	`(c ; d ; x )`

Figure 4.9: Spearhead Problem: Source Code and behavior protocol Fragments

Problem Statement

The problem is related to the state space mapping and appears when relation between the state spaces of JPF and BPChecker is not unique. Manager is unable to decide whether the verification should continue or backtrack.

Figure 4.9 shows an example. Here can be seen a source code and a corresponding protocol which define situation leading to the spearhead problem.

Looking at the source code, the important thing to note is the `else` branch. After invocation of required methods `c()` and `d()` is invoked the method `e()`, while the method `x()` is expected by BPChecker. This situation represents a classical example of the behavior protocol violation.

The spearhead problem appears when JPF tries to verify the source code introduced on Figure 4.9. To illustrate the progress of verification, Figure 4.10 is showing state spaces defined by the source code and the behavior protocol.

JPF verifies the source code and executes methods `a()` and `b()` from the `if` branch at first. Consequently it executes the method `e()`, reaches the end state and backtracks to the beginning state. When executing this source code again, to verify the `else` branch, after invocation of methods `c()` and `d()` JPF reaches the state that was already visited in the first execution path. JPF evaluates the situation and decides that it is not necessary to further explore this execution path, since it was already explored. Therefore, the method `e()` is not invoked and no protocol violation is reported.

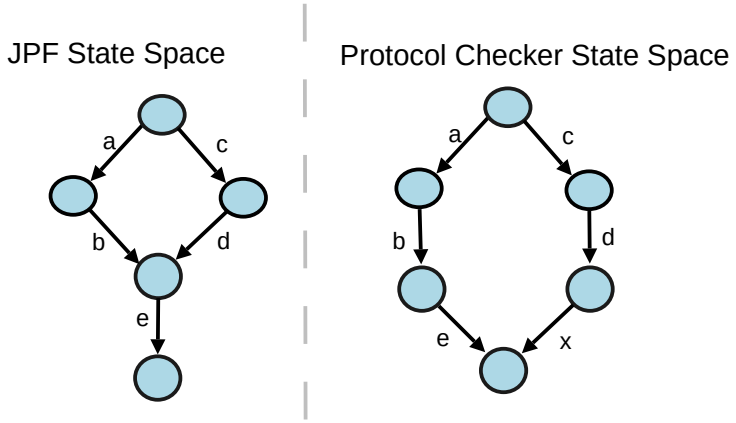


Figure 4.10: Spearhead State Space

Problem Solution

As shown in the problem statement, the spearhead problem appears when JPF reaches an already visited state, but BPChecker resides in a new state, compare both the state spaces depicted on Figure 4.10.

The solution is to modify the JPF search strategy and allow JPF to backtrack only if BPChecker state was already visited.

This solution requires modification of the BPChecker implementation, in order to evaluate whether a current state is a new one or it was already visited, see Section 5.3.

4.3.3 Parallelism

This problem is related to the usage of the parallel operator inside the behavior protocol. As defined in Section 2.1.1, the parallel operator applied on two sequences of events generates an arbitrary interleaving of the events from these two sequences.

From the component implementation point of view, it means that there are two threads running concurrently which invoke events on the interfaces of the component.

Problem Statement

The problem occurs when two threads are concurrently invoking a method on the required interface. The Figure 4.11 is illustrating this situation. Both the threads

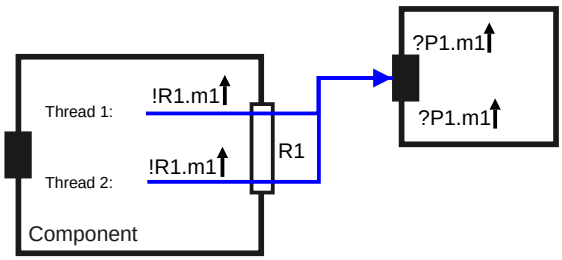


Figure 4.11: Parallel Request

are invoking a required method at the same time. JPF interrupts execution of these threads and notifies Manager. As described in Section 4.6, threads are starting to wait for the response.

Following Figure 4.12 depicts the situation when the response is going to be inserted into the JPF state space. JPF contains two threads which are waiting for this response, but Manager is not able to decide to which thread the response should be assigned.

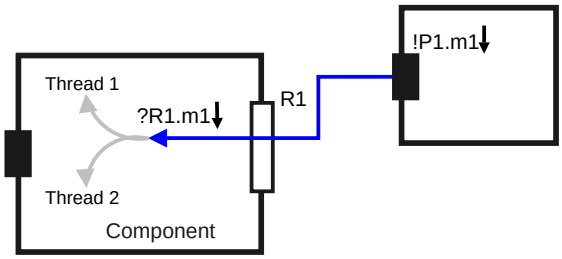


Figure 4.12: Parallel Response

Solution

The level of abstraction employed by a behavior protocol does not allow to distinguish between two identical responses, although at the component implementation level these two responses can be identified unambiguously.

When addressing this problem, it was necessary to extend the behavior protocol specification with thread suffixes. The suffixes are identifying every event, thus they enable to assign responses of calls to corresponding requests.

In other words, a thread suffix is represented be a unique identifier which is appended to every event. Those events which will be executed by one particular thread share the same thread suffix.

The Figure 4.13 shows application of thread suffixes. The parallel operators are dividing the behavior protocol into abstract parts, each of these parts will be executed by a different thread. The thread suffixes have to be appended in order to guarantee that a corresponding response can be identified for every request. Detail instructions describing the application of thread suffixes can be found in Appendix A.

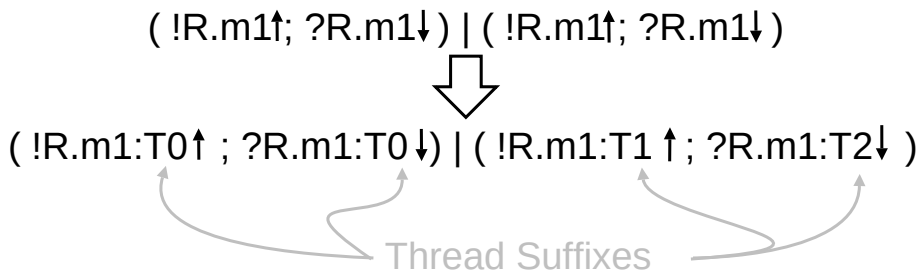


Figure 4.13: Behavior Protocol with Thread Suffixes

This straightforward way to solve the problem brings a downside of the suffix generation. Since it is not always obvious which response is corresponding to which request, the designer has to manually appended thread suffixes to events of the protocol.

4.3.4 The Alternative Operator

The alternative operator problem demonstrates differences between abstractions employed by a behavior protocol and a source code implementation. Behavior protocol syntax allows defining protocols that cannot be easily reflected by an implementation of a component. This situation appears in particular cases of the alternative operator application.

Problem Statement

While the behavior protocol `( ?a + !b )` is very simple, it defines component's behavior which cannot be easily verified.

When trying to simulate activity of an environment that would correspond to this protocol, Manager faces the dilemma of inserting the event `?a` or waiting for occurrence of the event `!b`. Both choices can lead to a protocol violation and there is no way of determining which decision is correct.

Manager cannot insert the event `?a` because there can be a running thread inside JPF VM that is going to emit the event `!b`. Similarly, waiting for the event `!b` to

occur represents menace that no such event will be emitted, which again leads to a protocol violation.

The problem is that Manager does not know whether the event `!b` will be in the future emitted by any of the component's running threads. In fact, there is no way of finding it out without executing these threads.

Problem Solution

As already indicated, the problem cannot be solved without executing the running threads. Only then Manager is able to decide whether the event `?a` can be inserted, according to an occurrence of the event `!b`.

This requirement could not be met, if the JPF Modification concept have not been employed when implementing the environment simulation solution. Thanks to this, the JPF Search strategy can be smoothly modified and Manager is able to postpone issuing of the event `?a` until the whole information is available.

Adjusting Search Strategy The key idea of the solution is to postpone the event `?a` and explore the remaining execution paths. After exploring paths that corresponds to execution of the threads inside the component, Manager already knows whether the event `!b` was emitted.

The situation is illustrated on Figure 4.14. The BPChecker state space shows possible events that can occur on the interfaces at this time. The red arrows represent events that introduce the alternative operator problem.

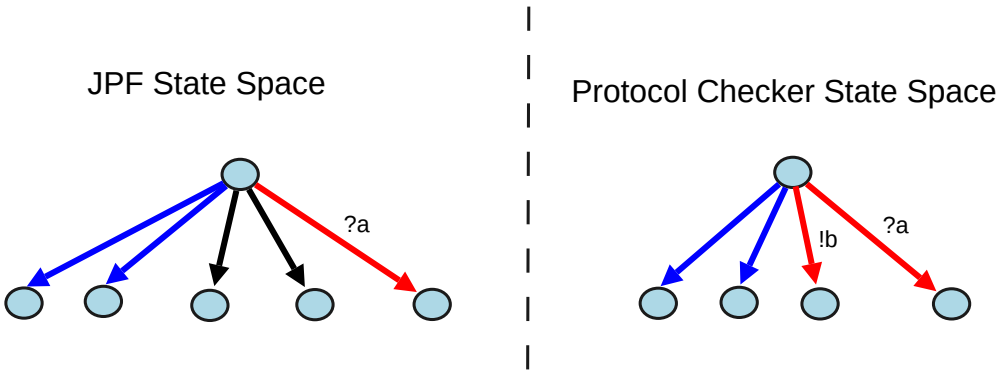


Figure 4.14: Alternative operator State Spaces

In this Figure, the JPF state space shows the adjusted search strategy. First, the sub-trees representing occurrences of non-conflicting events, denoted as the blue arrows, are explored. Then, the subtrees representing execution of the threads

inside the component are explored, the black arrows. After that it is able to evaluate whether the event `!b` was emitted. If it was not, the postponed event `?a`, denoted as the red arrow, is inserted.

Obviously, when the alternative operator problem occurs and there are no runnable threads inside JPF VM, it is not necessary to postpone any events because there is no thread to emit a conflicting event.

This idea is further employed when exploring those paths which represent instruction execution, the black arrows depicted on Figure 4.14. If JPF reaches state when no threads are runnable and the event `!b` still was not emitted, the postponed event can be inserted in every state along this path, see Figure 4.15 illustrating this example.

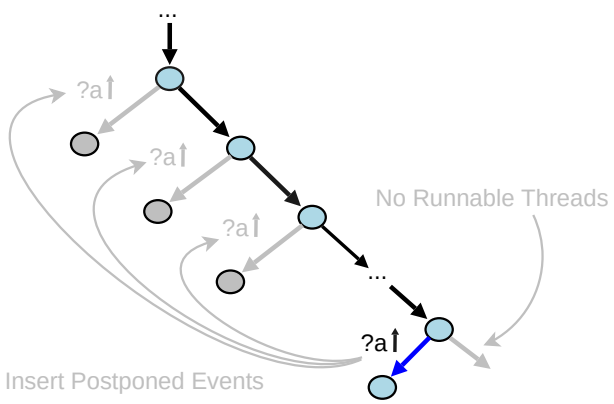


Figure 4.15: Inserting Postponed Events

Real-Life Experience Unfortunately, implementations of components in the real life are more complicated and contain several running threads. Therefore, one has to be careful when postponing events and exploring execution paths.

Consider the protocol `( ?a + !b ) ; !c`. When postponing the event `?a`, JPF will not reach a state with no running threads, instead, a state where the event `!c` is emitted will be reached. Such a behavior of course is not in compliance with the given protocol. Although the implementation is correct, a protocol violation will be reported because Manager has postponed the event for too long.

To face this obstacle, such a situation needs to be detected by Manager. It is necessary to interrupt the protocol violation report, backtrack from the current state and insert the event `?a` instead. Inserting the event `?a` is correct because the event `!c` indicated that one of the events `?a` or `!b` should already occur.

Not so obvious drawback of this solution is that it presumes correctness of the component's implementation. This presumption is reflected when the event `!c`

is emitted and Manager concludes that the event `!b` will not be emitted by any of the remaining threads. If there is a thread going to emit the event `!b`, it is a behavior error that will be reported, but the trace leading to the error will be: `?a;!b` instead of the: `!c;!b`, due to the intervention of Manager.

Lesson learned when facing this problem is that although the solution has been found and it identifies correctly protocol violations, in special cases traces leading to errors are reported inaccurately. Therefore, the user has to carefully examine those error reports which are related to the usage of the alternative operator.

4.3.5 The Repetition Operator

This Section is facing the (G1.4) goal - nondeterminism of the repetition operator, which was introduced in Section 2.4.

Problem Statement

The repetition operator defines an infinite number of finite sequences. In order to simulate all possible execution paths generated by the repetition operator, Manager should be able to simulate an infinite number of finite sequences too. Such a straightforward approach of course cannot be applied.

Figure 4.16 is showing a fragment of the BPChecker state space generated by the repetition operator. It can be seen that the operator is creating a cycle inside the state space.

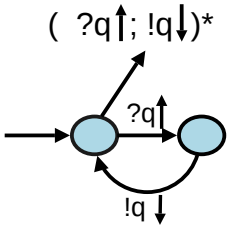


Figure 4.16: The Repetition Operator, BPChecker State Space

When facing the problem of a correct simulation of the repetition operator, it proved to be useful to analyze the cycles generated by this operator.

Cycles From the component implementation point of view, there can be distinguished two types of cycles.

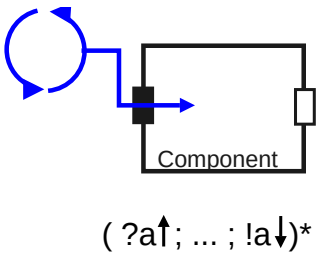


Figure 4.17: Outer Cycle

The *outer cycle* is generated by a component's environment and represents an infinite number of finite sequences of a provided method request. Schema representing this type of the cycle, along with the appropriate behavior protocol, can be seen in Figure 4.17.

The *inner cycle* is representing an infinite number of finite sequences of a required method request. Figure 4.18 shows a schema representing the *inner cycle* and the appropriate behavior protocol.

While the *outer cycle* is generated by an environment and has to be simulated by Manager, the *inner cycle* represents an invocation of required methods which are issued directly by the component's implementation. Therefore, no special administration of this type of a cycle is needed.

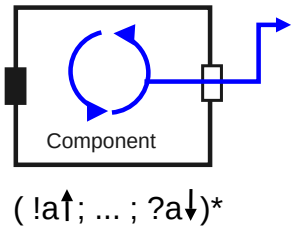


Figure 4.18: Inner Cycle

State Spaces The important aspect that needs to be also taken into account is the ability of JPF to evaluate reached states and identify already visited ones. Since the repetition operator is generating a cycle inside the BPChecker state space, it is reasonable to focus on identifying cycles inside the JPF state space too.

When addressing this idea, two variants of state spaces generated by the repetition operator can be identified.

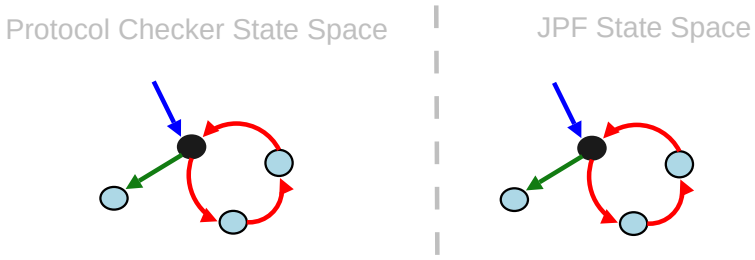


Figure 4.19: The Repetition Operator State Space, Variant 1

Figure 4.19 illustrates the first case where a cycle can be identified also inside the JPF state space. The JPF state space cycle corresponds to a situation when JPF explores an execution path representing one loop of the repetition operator cycle and during this exploration no component's attributes are modified. The state matching heuristic (Section 2.3.1) is applied and the ending state of the cycle is evaluated as an equal to the beginning one.

The second case is illustrated on Figure 4.20. Again, the execution path representing the cycle generated by the repetition operator is explored, but during the execution a component's attribute is modified. This prevents from applying the state matching heuristic and the ending state of this path is different from the beginning one.

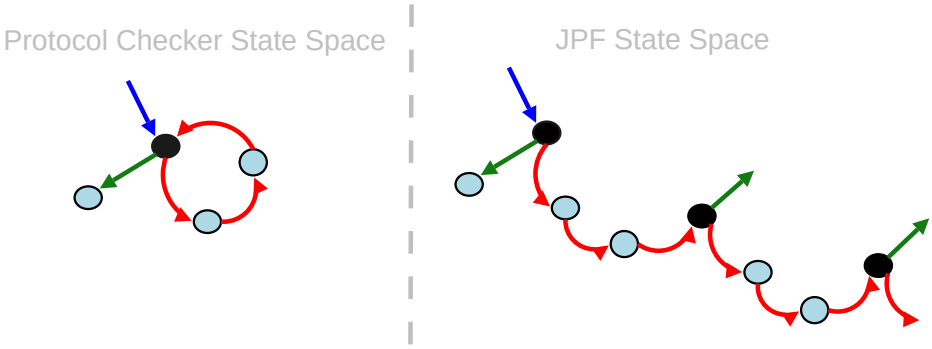


Figure 4.20: The Repetition Operator State Space, Variant 2

While exploration of the cycle in the first case is not enlarging the state space, thus does not need any special handling, the cycles modifying attributes of the component represent a threat, since they are able to generate potentially an infinite state space.

Problem Solution

Although the repetition operator represents a threat of a potentially infinite state space, which cannot be explored exhaustively, the analysis has revealed some possibilities.

Primarily, only specific types of the cycles generated by the repetition operator require special manipulation. For instance, *inner cycles* do not need to be trailed, since they are generated by the source code. In addition, also the cycles which allow to apply the state matching heuristic do not require any additional effort.

Therefore, only the cycles classified as *outer* with the ending states different from the beginning ones require increased attention because of the state space explosion problem.

Since the nondeterminism contained in the definition of the repetition operator cannot be reflected by any reasonable approach, at least partial solution needs to be identified.

An approach accepted as a solution constrains the maximum number of cycle loops that can be explored in one execution path. Even though this solution prevents from traversing through the whole state space, and thus potentially identification of every protocol violation is not possible, it is still the most suitable one.

Moreover, the user can set the maximum number of cycle loops as one of the input parameters which allows to adjust this constrain to the specific source code and thus increases the chance of a successful verification.

4.4 Summary

The goal of this chapter, to familiarize the reader with the solution, has been met.

The detail level provided in Sections 4.1 and 4.2 is sufficient for the reader to understand the solution applied to the software component verification.

Additionally, in Section 4.3 approaches employed when facing the crucial issues of the software component verification were demonstrated.

Chapter 5

Prototype Implementation

As a part of the thesis, a prototype application implementing the proposed solution was developed, see the (G2).

The prototype implementation is called *Carmen: Software Component Model Checker* and its distribution can be found on the cd attached to this document. For the User Documentation see Appendix A.

This chapter describes the main implementation techniques and presents the crucial modifications of the tools employed by the prototype implementation.

Section 5.1 is the fundamental part of this chapter, describes the most important modifications of the Java Pathfinder tool.

The main design issues concerning the implementation of Manager are introduced in Section 5.2.

When developing the prototype implementation, it was also necessary to modify implementation of the BPChecker tool. Section 5.3 addresses these extensions.

For further details of the prototype implementation see also [20].

5.1 Java PathFinder Modifications

This section presents the crucial modifications of the Java PathFinder tool [4].

5.1.1 JPF Initialization

The original implementation of Java PathFinder is creating a main thread and determines the explicit starting point (e.g. the `main` method) of the program to be verified at the initialization time. Unfortunately, when verifying a component implementation it is not possible to determine the explicit starting point, see also

Section 4.1.4. In order to overcome this obstacle, the initialization process has been completely reimplemented.

The first step of the Initialization is to prepare JPF for creating a component instance. Consequently a thread, which will execute a constructor method of the component, thus creating its instance inside JPF VM, is created. The instantiation process is finished by creating a starting state of the verification. From this state (denoted as a zero state) all execution paths of the component implementation are beginning.

When the starting state is created, Manager starts the verification process by determining starting points of the verification. It cooperates with Manager, obtains list of provided methods which are also starting methods in the meaning of the given frame protocol. After that, the following approach is similar to the one applied when inserting events, Section 4.2.3.

5.1.2 POR Modification

To successfully intercept an emitted event, it is necessary to trace all executions of the invoke and return byte code instructions that are corresponding to the methods on the provided and required interfaces of the component. The purpose for the event interception is to associate one state from the state space to every emitted event; therefore, it is required to modify the process executing byte code instructions.

As mentioned in Section 2.3.1, the POR algorithm is the most important mechanism to reduce the state space. The application of this algorithm allows to execute more than one instruction without context switching, this modifies the state space - the transitions between states represent execution of several byte code instructions.

However, the application of the POR algorithm can be problematical. When employing the POR algorithm, the sequence of instructions that leads to emitting an event can be evaluated as not context-switching and the execution continues. Therefore, it is necessary to modify the POR algorithm in order to respect the method calls on the interfaces of the component and to create the states associated with these calls.

By introducing the Modified POR-algorithm the application is able to detect execution of an invoke or return instruction that corresponds to an event emitted by the component. At this point the application is also able to create a state representing the call. The difference between using original POR-algorithm and its modified version is depicted on Figure 5.1.

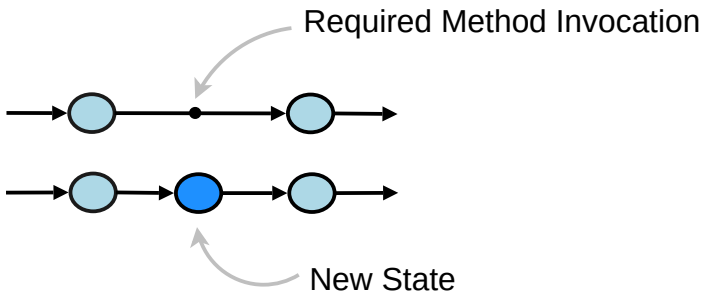


Figure 5.1: Modified POR Algorithm

5.1.3 Search Strategy

Although the original Java PathFinder offers several search strategies directing the state space exploration, none of these strategies was suitable for the exploration of the state space generated by the component implementation.

To face this obstacle, a new strategy that is appropriate for the purpose of the component verification was implemented. Even though, this solution removes the possibility of choosing the optimal search strategy from the user, it was inherently necessary.

The new search strategy is based on the Depth First Search strategy implemented in the original JPF and therefore the name remained the same. Obvious extensions to the DFSearch strategy concern the conditions determining the next step of JPF, since the decision has to be made in a cooperation with Manager and BPChecker. The DFSearch strategy has to be also adjusted to the state space extension (Section 4.2.1).

5.1.4 Thread Management

As described in Section 4.2.3, a method invoked on a provided interface results in creating a new thread inside JPF. The created threads are kept in a data structure called *ThreadList*, which is maintained by JPF VM.

JPF VM is maintaining the ThreadList, which keeps threads created during the verification.

The original implementation of JPF was appending a new thread at the end of the thread list. There were not employed any garbage-collection techniques for terminated threads. Therefore, the terminated threads remained in the ThreadList until their removal by a backtracking step. This proved to be a drawback when developing the prototype implementation.

The absence of the garbage-collection techniques for ThreadList results in creating a new thread for every absorbed request. Thus, every thread is used only once, after its termination the new one with different id number is used. Such thread management completely eliminates the state matching heuristic (introduced in Section 2.3.1), which considers states with different sequences of terminated threads as non equal. E.g. two sequences of exactly the same events executed by threads with different ids are evaluated as non equal.

The need for efficient state matching heuristic, the key mechanism that keeps the state space as small as possible, brought along an implementation of the ThreadList with the garbage collection.

5.2 Manager Implementation

One of the biggest challenges of the Manager implementation is to simulate an environment of the component. Additionally, the concept of backtracking employed inside JPF has to be implemented also by Manager, in order to simulate not only one but every possible sequence of events occurred on the interfaces of the component.

Whenever JPF advances or backtracks a state, it notifies Manager. Through these notifications, Manager is able to control the progress of verification. In order to determine which execution paths have already been explored and which are going to be explored, Manager contains Manager Stack.

5.2.1 Manager Stack

Manager is using this data structure to control the exploration of the state space. ManagerStack is keeping the currently explored path and thelist of events that are going to be inserted. For every state of this path it is able to determine the list of not yet visited branches.

5.3 Protocol Checker Implementation

When developing the prototype implementation, BPChecker - a Runtime Protocol Checker developed at [11] was employed. This protocol checker implementation is based on Static Protocol Checker [10].

5.3.1 Protocol Checker Modifications

Facing the alternative operator problem (Section 4.3.4) and the repetition operator problem (Section 4.3.5) brings along the need for further extensions of the BPChecker implementation. This section is introducing these modifications. For more details see also [20].

Although BPChecker provides a list of events which can occur on interfaces of component in a particular time, when dealing with the alternative or repetition operators, such information is not sufficient. Manager needs also to know whether a specific event is e.g. a starting event of some cycle generated by the repetition operator or an event involved in the alternative operator problem.

To provide to Manager this type of information, every event contains also the information describing its position inside the frame protocol. Thanks to this description, Manager recognizes every event that needs special attention.

Chapter 6

Case Study

This Chapter presents a case study demonstrating features of the implemented solution on the real-life application developed as a part of the Component Reliability Extensions for Fractal Component Model Project [7].

In Section 6.1 the application is introduced under discussion in more details.

Section 6.2 describes the verification process, introduces the way to interpret results of the verification process and finally presents the examples of the error identification.

The statistics summarizing the performance of the verification process are presented in Chapter 7.

6.1 Demo Application

The Demo Application represents a nontrivial component-based system, which was developed as a part of the project funded by France Telecom aiming at integration of behavior protocols into the Fractal Component Model [6].

In order to familiarize the reader with the application under discussion, the following paragraphs are concisely describing the architecture of the system.

The system represents a full-fledged and complex application designed and implemented with respect to principles of the software component systems development.

The basic motivation for development of this application was to obtain a system that would both extend the airport services for wireless internet connection and provide a necessary level of a security. The connection is granted to those who own the first class or business class tickets. Also the owners of the Frequent Flayers Card can use this service if they have a valid ticket. To other passengers this service is available only if they prepay the connection time by their credit card.

The application consists of more then twenty components including virtual components used to model the synchronization. The complexity of the whole system is also reflected by employing the hierarchical approach to design components. Both primitive and composite components of different complexities are present. In Figure 6.1 (borrowed from the project's documentation) the overview of whole component's architecture can be found.

In addition, a short presentation of the main components follows. The *Firewall* component represents a desire for a security, it blocks unauthorized internet connections and redirects them to the login page. To provide an access to the database of the airlines, the *FrequentFlyerDatabase* and *FlyTicketDatabase* components are integrated. Another component providing an access to a database is *AccountDatabase* component, which encapsulates the database of accounts with prepaid internet connection. The *CardCenter* compoentn is used to communicate with the bank credit card services. Every logged user is represented by a dynamically created entity defined by the *Token* component. The *DhcpServer* component is representing a DHCP server for a dynamic IP address allocation. This component further contains primitive components supporting the use of the permanent IP address database. The whole system is controlled by the *Arbitrator* component.

As specified in the goals of this thesis, the verification of primitive components is the main task. Therefore, the *FlyTicketClassifier*, *IpAddressManager*, *ValidityChecker*, *AccountDatabase* and *Arbitrator* components were chosen to demonstrate the verification abilities of the prototype implementation.

6.2 The Verification Process

This Section presents the verification process of the Demo application. All steps necessary to successfully verify the component's implementation are described here.

First, the preparation procedure is introduced, Section 6.2.1. Because only the source code itself is not sufficient for the tool to start the verification, additional specification data are required. In special cases there are also required modifications of the source code, this is also addressed here.

After that, the verification of a component can be started, see Section 6.2.2.

And finally, to complete the verification process, it is necessary to correctly analyze and interpret results of the verification, Section 6.2.3.

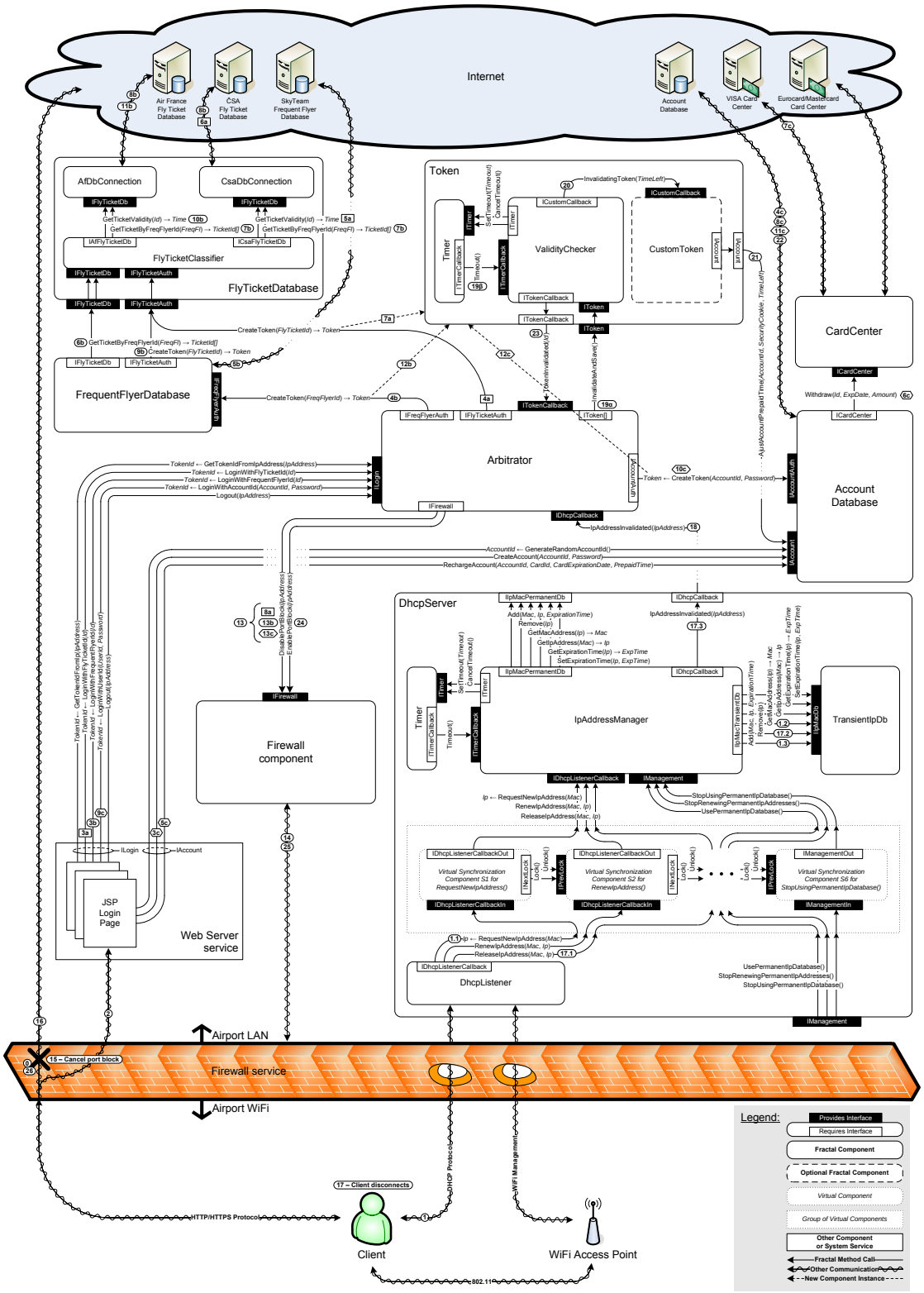


Figure 6.1: Overview of the Airport Internet Providing Demo Application

6.2.1 Preparation

Before the verification can be started, some preparations need to be done in order to provide to the tool all the information necessary to verify a component.

Component Specification Files Because the source code itself does not provide the whole information necessary to verify a component, the component specification files are introduced. The Fractal ADL file defines the component interfaces and the value specification file describes set of values that can be used as input/output parameters of methods.

Value Specification As described in Section 4.3.1, it is required to provide value specifications for input/output parameters of the methods on the interfaces. Since the tool supports value specifications only for selected data types, see Appendix A, the designer has to modify those methods which use the unsupported data types in parameters.

Unfortunately, the modification of the method data types is not the most suitable solution. It requires the additional effort before the component can be verified, moreover, this approach can be considered as error-prone. Event though value specifications techniques provide a comfortable way to define possible parameter values, there is still a threat that some of execution paths will not be reachable due to these modifications. See Section 9.1.1 where this problem is further addressed.

6.2.2 Verification

As the verification process proceeds, the tool is notifying the designer about the number of states which were explored. Also, whenever an implementation error is detected, it is immediately reported.

After finishing the verification process, the statistics about the progress of the verification are printed. Additionally, implementation errors, if there were any, are also shown in order to provide to the user an opportunity to interpret the results of the verification.

6.2.3 Results Interpretation

The verification process distinguishes two types of implementation errors: the errors identified by JPF and the errors identified by BPChecker.

JPF Errors These errors are identified directly by the Java PathFinder Model Checker. They represent the same set of errors which could be identified by the original Java PathFinder tool. E.g. deadlocks, unhandled exceptions or property violations. For more about these errors and about defining user's properties see [4].

Behavior Protocol Violations These errors are representing a violation of the component's frame protocol. The application is detecting these errors when tracing the events occurring on the interfaces of the component.

Behavior Protocol Violation

The primary goal of this work was to verify that the component obeys its frame protocol. Therefore, to report the violation of the behavior protocol is the task of the highest priority.

Since the communication absorbed by the component is generated by Manager cooperating with BPChecker, it is inherently correct and does not have to be considered. Therefore, only the communication emitted by the component needs to be verified.

To sufficiently describe each situation which represents violation of a frame protocol, the states of BPChecker and JPF are recorded whenever an even violating the given protocol is emitted.

The state of BPChecker is represented by the trace of events that occurred before the error event was detected. To express the state of Java PathFinder, JPF VM stack trace of the thread which emitted the error event is saved. For the sake of completeness, stack traces of other live threads are also recorded.

Although this information identifies the particular protocol violation, it may be sometimes difficult for the designer to analyze and interpret a very long error trace. Moreover, the situation is complicated by the fact that there can be more then one trace leading to the error. This causes that the notifications reporting several protocol violations with different traces are de facto reporting one and the same error.

Facing this obstacle would be beyond the scope of this work, therefore it is recommended to study the paper [21] which introduces several approaches to the problem of the error trace complexity. Other documents dealing with counterexamples and error traces are [22, 23, 24].

Nevertheless, event though the error trace interpretation can be sometimes problematic, the verification of components from the Demo Application was flawless. Following Section shows several examples of the verification and error interpretation.

```
public int GetFlyTicketsByFreqFlyerId(int FlyerId) {
 if (FlyerId == 0)
   return 0;

 // Due to permutation of calls
 // the protocol violation occurs
 iCsaTicketDb.GetTicketsByFlyerId(FlyerId);

 iAfTicketDb.GetTicketsByFlyerId(FlyerId);
 ...
}
```

Figure 6.2: Example 1, Source Code Fragment

6.2.4 Examples

This section presents the verification process of the Demo Application and selected examples of the error identification. The component's implementation codes were intentionally modified and behavior protocol errors were inserted in order to demonstrate the functionality of the prototype implementation.

Example 1

Example 1 shows an error identified when verifying the *FlyTicketClassifier* component. In Figures 6.2 and 6.3 the reader can consider the fragments of the implementation code and the corresponding frame protocol. It can be seen that after the execution of the method `iAfTicketDb.GetTicketsByFlyerId(FlyerId);` the behavior protocol violation should be reported.

After launching the process of the verification, the protocol violation showed on Figure 6.4 is reported. This message describes the occurred event, so the designer is able to identify the place (notice the source-line number in the StackTrace field), where the error occurred, and the circumstances which lead to this error.

Example 2

The Example 2 shows another success story of the software component code verification.

Figure 6.5 shows the fragment of the frame protocol of the *IpAddressManager* component. Protocol express an ability of the *IPAddressManager* to accept a

```
?iTicketDb.GetFlyTicketsByFlyerId:0 {
  (
    !iAfTicketDb.GetTicketsByFlyerId:0
    ;
    !iCsaTicketDb.GetTicketsByFlyerId:0
  )
  +
  NULL
}
```

Figure 6.3: Example 1, Frame Protocol Fragment

```
Behavior Protocol Violation detected
Event: !iCsaTicketDb.GetTicketsByFlyerId:0^
Protocol Checker Trace:
      ?iTicketDb.GetTicketsByFlyerId(int FlyerId=6):0^;
      !iCsaTicketDb.GetTicketsByFlyerId:0^
State Number: 126
Depth: 1
StackTrace:
        at FlyTicketClassifierImpl.GetTicketsByFlyerId
        (FlyTicketClassifierImpl.java:114)
```

Figure 6.4: Example 1, Behavior Violation Report

method call requesting a mode change (`?iManagement.UsePermanentIpDb:2`↑) at any time, but the component cannot respond immediately.

The protocol defines that the component has to wait until the processing of all pending calls on the *iDhcpListener* interface is finished and then it can emit the response (`?iManagement.UsePermanentIpDb:2`↓).

However, such a behavior is not reflected by the implementation of the component. The component is emitting the response immediately after the mode changing method is absorbed. This conflict is during the verification process detected and the protocol violation message appears, see Figure 6.6.

```
(
  (
    ?iDhcpListener.RenewIpA:6 {...}*
    +
    ?iDhcpListener.RequestNewIp:7 {...}*
  )
  |
  ?iManagement.UsePermanentIpDb:2^
) ; !iManagement.UsePermanentIpDb:2$
```

Figure 6.5: Example 2, Frame Protocol Fragment

```
Behavior Protocol Violation detected Event:
?iManagement.UsePermanentIpDb:2$ Protocol Checker Trace:
        ...
        ?iDhcpListener.RenewIpA:6^;
        ?iManagement.UsePermanentIpDb:2^;
        !iManagement.UsePermanentIpDb:2$
StackTrace: ...
```

Figure 6.6: Example 2, Protocol Violation Report

Chapter 7

Evaluation

In Section 1.5 the goals of this thesis were stated. Section 2.4 has further specified the particular tasks that need to be target. This Chapter discusses the achieved results with respect to declared goals.

For the sake of the goal (G1), Chapters 3 and 4 were crucial. In Chapter 3 a complex analysis of a primitive component verification has been conducted and the optimal solution was progressively refined. Furthermore, Chapter 4 presented an approach of identified solution to a primitive component verification in very detail. Consequently, in Section 4.3 the crucial issues of the component verification have been successfully addressed, which confirmed a suitability of the solution.

In order to discuss the performance of the prototype implementation and thus evaluate the fulfilment of the goal (G2), several tests have been performed. The test data were already presented in Chapter 6, here the results of the tests are discussed.

Additionally, with an assistance of the Distributed Systems Research Group [11] was able to verify the specified components by the Software Component Model Checker (based on the research presented in [8]). Thanks to this, performances and abilities of both the tools could be confronted. A short description of this verification tool can be found also in Section 8.3.1.

All the presented verification tests are included with some additional in the *Carmen* distribution and they can be launched by the user. For the instructions describing execution of the tests see Appendix A.2.6.

7.1 Test Settings

Before the results of the tests can be presented, this Section provides the information necessary to properly interpret the presented results.

The results of the tests are presented in the tables denoting the verification process statistics together with key parameters affecting the verification. The Table 7.1 summarizes the meaning of the monitored parameters.

Unique States	Number of unique states reached during the verification
Visited States	Total number of all reached states
Time	Total time of the verification process
States/Second	Number of Visited States explored per second
Value Specification	Set of values used as input/output parameters
Cycle Limit	Parameter restricting number of loops generated by the repetition operator

Figure 7.1: Descriptions of Testing Parameters and Statistics

The tests performed on the prototype application are denoted by the *Carmen* key word, test performed by the Software Component Model Checker (DSRG Checker) developed at [11] are denoted by the *DSRG* key word.

DSRG Checker Restrictions

Although *DSRG Checker* was, similarly to *Carmen*, developed for a primitive component verification, different concepts employed during the development constrain the abilities of this tool.

First, the usage of the alternative and repetition operators is restricted and therefore it is necessary to modify the behavior protocols of the verified components. Before the verification can be started, a behavior component is simplified in order to avoid problematic situations (e.g. alternative operator problem,...). Additionally, the higher levels of parallelism are also limited by this solution (see [25] presenting the approaches to a behavior protocol simplification).

With motivation to precisely confront the performance of both the tools, in cases when *DSRG Checker* required modifications of a behavior protocol, *Carmen* was tested with these simplified versions of protocols and the results of the additional tests are also provided.

And second, *DSRG Checker* needs to generate an environment to a component under discussion before the verification process can be started. The elapsed time of the environment generation process is not included in the column presenting the total time of the verification.

Testing Data

The components verified in the tests are included in the *Carmen* distribution, which is available on the attached cd.

Implementations of the components along with corresponding behavior protocols are placed in the *input/* distribution directory. See Appendix A for details.

7.2 Results

This section presents the results of the tests.

All the tests were run on Pentium 4 3.0 GHz with 2.0 GB RAM, Windows Server 2003 OS.

7.2.1 FlyTicketClassifier Test

The *FlyTicketClassifier* component represents a relatively simple component implementation. Although a frame protocol of this component is non-trivial, it does not contain any parallel operators which are the main factor influencing a size of the state space.

Since the frame protocol of the *FlyTicketClassifier* component does not contain any repetition or parallel operators, *DSRG Checker* do not require any simplifications of the frame protocol and thus both tools verified the component against the original protocol.

	# States		**Time**	**States/ Second**	**Value Specification**	**Cycle Limit**
	Unique	**Visited**				
Carmen	922	1 920	3s	640	[0,1,2,3,4,5,6]	7
DSRG	6 519	10 254	4s	2563	[0,1,2,3,4,5,6]	2

Figure 7.2: Results of the FlyTicketClassifier Verification

The results of the test presented in Table 7.2 shows that the size of the state space of *Carmen* is more smaller then the *DSRG Checker* state space. This is caused by the necessity to include the generated environment in the *DSRG Checker* state space.

In contrast, verification times are similar. While *DSRG Checker* verifies a closed system, Carmen simulates an environment during the verification and therefore the progress of the verification is slower.

7.2.2 ValidityChecker Test

The *ValidityChecker* component represents a component with a complex frame protocol employing several parallel and repetition operators. The frame protocol of this component is shown in Figure B.3 which can be found in Appendix B.

Table 7.3 presents the statistics of the verification process performed by *Carmen*. It can be seen that the state space traversed during the verification is enormous even thoug the simple value specification and low cycle limit were used.

	# States		Time	States/ Second	Value Specification	Cycle Limit
	Unique	Visited				
Carmen	422 922	1 336 360	50m 32s	441	[1]	3

Figure 7.3: Results of the ValidityChecker Verification

Because *DSRG Checker* requires modifications of the frame protocol before the verification process can be started, in order to confront its performance with the performance of *Carmen*, additional tests verifying the simplified frame protocol against the *ValidityChecker* component were conducted. The simplified frame protocol can be found on Figure B.4 in Appendix B.

	# States		Time	States/ Second	Value Specification	Cycle Limit
	Unique	Visited				
Carmen	6 074	14 898	34s	438	[1]	3
DSRG	166 977	378 437	9m 30s	663	[1]	2

Figure 7.4: Results of the ValidityChecker Verification, Simplified Protocol

In Table 7.4 are presented the results of additional tests, it can be observed that the state space traversed by *Carmen* is much smaller. Again, *DSRG Checker* requires to include a generated environment inside the state space.

When confronting results of both the checkers, it can be seen that *DSRG Checker* still have better performance when considering number of states explored per second, but when it comes to the total time of the verification process, *Carmen* reaches remarkably better result.

7.2.3 Arbitrator Test

The *Arbitrator* component represents the second test focused on a verification of a complex component against the frame protocol containing parallel operators.

Similarly to the previous test, *DSRG Checker* employed modifications on the frame protocol, therefore the component was tested against two protocols. The original protocol can be seen on Figure B.1, the simplified version in Figure B.2. The key difference between these protocols lies in the level of parallelism.

	# States		Time	States/ Second	Value Specification	Cycle Limit
	Unique	Visited				
Carmen	1 082 999	3 117 552	47m 40s	1090	[1]	3

Figure 7.5: Results of the Arbitrator Verification

Table 7.5 presents the results of the verification conducted by *Carmen* using the original frame protocol. Even though the frame protocol of the *Arbitrator* component is not as complicated as the protocol of the *ValidityChecker* component, the implementation of the *Arbitrator* component is more complex and therefore the state space explored during the verification is so large.

	# States		Time	States/ Second	Value Specification	Cycle Limit
	Unique	Visited				
Carmen	435	592	2s	296	[1]	3
DSRG	4 033	9 324	4s	2331	[1]	2

Figure 7.6: Results of the Arbitrator Verification, Simplified Protocol

Statistics of the tests verifying the component against the simplified version of the frame protocol are presented in Table 7.6. Interpretation of the results confirms the conclusions made in the previous test. *Carmen* is able to verify a complex protocol without the need for further modifications (in contrast with *DSRG Checker*) and even when both the checkers use the same protocols, Carmen is still able to verify the component faster.

7.2.4 Alternative Operator Test

This test demonstrates the solution of the alternative operator problem. We are verifying the *FlyTickerClassifier* component, but its frame protocol was modified in order to cause the alternative operator problem.

Because the solution of the alternative operator problem is not supported by *DSRG Checker*, the results could not be confronted. The results of this test are presented in Table 7.7.

The distribution also provides an additional test solving the alternative operator problem, this test verifies the *Seller* component. The component was implemented with motivation to demonstrate the alternative operator problem on a simple example. Thanks to this, the solution introducing the postponed events is here clearly illustrated and the user can easily trace the verification process.

	# States		Time	States/ Second	Value Specification	Cycle Limit
	Unique	Visited				
Carmen	273	494	1s	494	[0,1,2,3,4,5,6]	7
DSRG	N/A	N/A	N/A	N/A	N/A	N/A

Figure 7.7: Results of the Alternative Operator Test

7.2.5 AccountDatabase Test

The *AccountDatabase* test demonstrates the verification of a sophisticated component implementation against a complex frame protocol which contains several parallel operators.

	# States		Time	States/ Second	Value Specification	Cycle Limit
	Unique	Visited				
Carmen	60 397 977	78 152 637	27h 31m	790	[1]	3

Figure 7.8: Results of the AccountDatabase Verification

The results presented in Table 7.8 show really big state space, whose exploration took several hours. However, before the state space generated by this component could be entirely explored, the memory assigned to the verification process was exhausted, and therefore the verification process was stopped.

In this case, the *Carmen* tool cannot be used as a proof of correctness, but it still provides valuable information and its application mainly in debugging phases of development appears to be more then useful.

7.3 Summary

The bottom line is that the goals stated in Section 1.5 were fulfilled.

Moreover, results of the tests have shown that *Carmen* is able to verify complex components in a reasonable time. It should be also highlighted that no other reductions of a frame protocol are required.

The confrontation of *Carmen* with *DSRG Checker*, which requires additional reductions of a frame protocol, has revealed that *Carmen* reaches remarkably better performance than *DSRG Checker* event when verifying components against simplified versions of the protocols.

Chapter 8

Related Work

8.1 Software Component Systems

There are many component system used in the software industry. One of the most known are CCom are DCom, members of Microsoft's component systems family [26]. Others are e.g. Enterprise JavaBeans (EJB) by Sun Microsystems [27] and CORBA Component Mode (CCM) by Object Management Group [28].

Although the solution developed in this work is primarily focused on the Fractal Component Model and the Case Study in Chapter 6 was conducted on a Fractal component application, the approach can be used to verify the SOFA components as well. SOFA Component Model [5] is a project developed at [11] and similarly to Fractal, it introduces hierarchical components and behavior protocols.

8.2 Model Checkers

8.2.1 Bandera

Bandera [29] represents a set of tools and modules which are designed to verify Java programs.

Bandera accepts a complete Java program as an input. A given program is then with assistance of modules translated into a language that can be verified by a specified model checker. Originally, Spin [30] and Java PathFinder were supported as model checkers. Currently, only a Bandera specific model checker Bogor [31] is recommended.

Although Bandera is not intended to verify software components, it decomposes a target program into a part which will be verified and the rest that will be represented by specially generated environment. This approach is very similar

to the environment generation concept presented in Section 3.3.1. Bandera also allows to use value specifications for the method parameters of the given classes.

The biggest disadvantage of this tool is that Bandera's recent release is an alpha version which is not fully stable yet.

8.2.2 Other Model Checkers

The Zing model checker [32] is a model checker that accepts a given program in a specific language - Zing specification language [33]. Before verification, the target program has to be translated into model defined by this specification language and then this model is verified against user-defined assertions.

The SLAM model checker [34] developed by Microsoft is a tool formally verifying device drivers for the Windows operating systems. Again, creates model of a target program and verify whether the specific properties are hold. These properties are described in a special language called SLIC [35].

Other interesting project is Charmy [36] which is also testing consistency between software architecture and the functional requirements. SPIN model checker [30] is employed as the verification engine.

8.3 Component Behavior Checking

8.3.1 DSRG Component Checker

The Software Component Model Checker [8] was developed by the Distributed Systems Research Group [11] at Charles University and similarly to the approach employed in this thesis it uses Java PathFinder in cooperation with BPChecker.

However, the tool is differently employing JPF and BPChecker. A motivation to preserve the original Java PathFinder implementation reveals the *Missing Environment Problem*. To face this, the concept introduced in Section 3.3.1 was implemented. For more information about the environment generation see [37].

The different approach employed by implementation is substantially modifying the verification process. Before the verification can be started, it is necessary to generate an environment in order to obtain a closed system that can be verified by JPF. Here, an environment is generated from a simplified version of a given frame protocol.

Moreover, this approach does not support solutions of the problems related to the usage of the repetition and alternative operators (described in Section 4.3). The usage of the parallel operator is also limited. Therefore, every frame protocol has

to be preprocessed in order to simplify and reduce the unsupported forms of the protocol.

The reductions and modifications of the frame protocol are described in [25]. As the fundamental reduction should be highlighted a translation of the parallel operator. Every parallel operator between three or more subprotocols is replaced with all the distinct pairs of subprotocols, where each pair is connected with the parallel operator. Compare Figures B.1 and B.2 which are showing the protocol before and after reductions.

Although the components are verified agains the protocols which are only simplified versions of the original protocols, this tool is still valuable. Thanks to the employed reductions, the performance of the tool is very good. Furthermore, the advantage of portability to the new versions of Java PathFinder should not be overlooked.

For the performance comparisons between both the solutions, see Chapter 7.

Chapter 9

Conclusion

The goals of the thesis, as declared in Section 1.5, have been reached.

The most significant outcome of this work is *Carmen: Software Component Model Checker*, the prototype application implementing the solution presented in Chapter 4. In Chapter 6 the features of the implementation were demonstrated on a real life component application. The evaluation of the performance statistics presented in Chapter 7 has shown that the solution proposed and implemented by this work fulfils the stated goals and, in particular cases, exceeds expectations.

Although the stated goals comprised numerous issues to solve and many questions had to be answered, this master thesis represents a a successful attempt for a tool that is able to comprehensively model check a software component against a behavior protocol.

9.1 Future Work

When it comes to future work, there are two tasks that may be aimed at in the future.

9.1.1 Extended Value Specification

Even though the prototype application supports two types of value specifications, this aspect can be still considered as constraining when verifying the real life component application.

If the application uses non-supported data types as input/output parameters of the provided/required methods, the source code has to be modified in order to contain only the supported data types, which is the requirement bringing extra

effort to the verification process. Moreover, this approach can be considered as error-prone.

To face this obstacle, en extension that would bring a more sophisticated approach to value specifications could be implemented. The desired solution should be able to specify values for more complex data types, e.g. arrays of basic data types, String or user defined classes. To achieve this, containers, carrying set of values defined by the user, could be employed.

Inspiration for this solution could be also find in [37, 29].

9.1.2 Java PathFinder Plugin

The JPF Modification concept (Section 3.3.3), which is employed by the prototype implementation, is bringing the drawback of unfeasible portability to the new versions of Java PathFinder.

The obvious challenge is to summarize modifications of the JPF into a plugin which could be smoothly appended into the future versions of Java PathFinder.

Bibliography

[1] F. Plasil, S. Visnovsky. Behavior Protocols for Software components. *IEEE Transactions on Software Engineering*, 28, no. 11, Nov 2002.

[2] E. Bruneton, T.Coupaye, M. Leclerc, V. Quema, J-B. Stefani. An Open Component Model and Its Support in Java. *7th SIGSOFT International Symposium on Component-Based Software Engineering (CBSE7)*, May 2004.

[3] J. Adamek and F. Plasil. Component Composition Errors and Update Atomicity: Static Analysis. *Journal of Software Maintenance and Evolution: Research and Practice 17(5)*, Sep 2005.

[4] NASA. Java PathFinder Model Checker, http://javapathfinder.sourceforge.net/.

[5] The SOFA Project. http://sofa.objectweb.org/.

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

[7] Component Reliability Extensions for Fractal Component Model, http://kraken.cs.cas.cz/ft/public/public_index.phtml.

[8] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software components: Combining Java PathFinder and Behavior Protocol Model Checker.

[9] J.Kofron, J. Adamek, T.Bures, P. Jezek, V. Mencl, P. Parizek, F. Plasil. Checking Fractal Component Behavior Using Behavior Protocols. 2006.

[10] F.Plasil M. Mach and J. Kofron. Behavior Protocol Verification: Fighting State Explosion. http://nenya.ms.mff.cuni.cz/publications/Kofron-kokorin04.pdf.

[11] Distributed Systems Research Group, http://nenya.ms.mff.cuni.cz/.

[12] E. Clarke, O.Grumberg, and D.Peled. Model Checking. *MIT Press*, Jan 2000.

[13] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal.Volume 10, Number 2*, April 2003. http://ti.arc.nasa.gov/people/wvisser/ase00FinalJournal.pdf.

[14] A. Groce and W. Visser. Heuristics for Model Checking Java Programs. *International Journal on Software Tools for Technology Transfer (STTT), Volume 6, Number 4*, December 2004. http://ti.arc.nasa.gov/people/wvisser/stttheur.ps.

[15] A. Groce and W. Visser. Model Checking Java Programs using Structural Heuristics. *Proceedings of ISSTA 2002.*, July 2002. http://ti.arc.nasa.gov/people/wvisser/issta02_paper.ps.

[16] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, R. Washington and W. Visser. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods in Systems Design Journal, volume 25*, September 2004.

[17] G. Lindstrom, P. Mehlitz and W. Visser. Model Checking Real Time Java Using JavaPathfinder. *Proceedings of the Third International Symposium on Automated Technology for Verification and Analysis (ATVA)*, October 2005.

[18] D. Giannakopoulou, C. S. Pasareanu, J. M. Cobleigh. Assume-guarantee Verification of Source Code with Design-Level Assumptions. *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, May 2004.

[19] Java Reflection API. http://java.sun.com/docs/books/tutorial/reflect/index.html.

[20] A. Plsek. Carmen: Software Component Model Checker, Developer Documentation, 2006.

[21] J. Kofron P. Jezek and F.Plasil. Model Checking of Compnent Behavior Specification: A Real Life Experience.

[22] V. Kumar N. Kumar and M.Viswanathan. On the Complexity of Error Explanation. *VMCAI'05, ACM*, 2005.

[23] M. Naik T. Ball and S. Rajamani. From Symptom to Cause: Localizing errors in counterexample traces. *Proceedings of POPL 2003, ACM*, 2003.

[24] W. Visser. A. Groce. What Went Wrong: Explaining Counterexamples. *Proceedings of the SPIN Workshop on Model Checking of Software Programs*, 2003.

[25] P. Parizek and F. Plasil. Specification and Generation of Environment for Model Checking of Software Components.

[26] Microsoft. Component object model technologies. http://www.microsoft.com/net.

[27] SUN Microsystems. Enterprise JavaBeans. http://java.sun.com/ejb.

[28] CORBA Component Model. *http://www.omg.org/technology/documents/formal/components.htm*.

[29] SAnToS laboratory. Bandera. http://bandera.projects.cis.ksu.edu/.

[30] Spin Model Checker. http://spinroot.com.

[31] Bogor tool set. http://bandera.projects.cis.ksu.edu.

[32] S.K.Rajamani J.Rehof Y.Xie T.Andrews, S.Qadeer. Zing: A a model checker for concurrent software. http://research.microsoft.com/zing/.

[33] Microsoft Zing Research Team. Zing Specification Language. http://research.microsoft.com/zing/ZingLanguageSpecification.pdf.

[34] S.K. Rajamani T.Ball. The slam project: Debugging system software via static analysis.

[35] S.K. Rajamani T.Ball. Slic: A specification language for interface checking.

[36] Charmy. http://www.di.univaq.it/charmy/.

[37] P. Parizek and F. Plasil. Specification and Generation of Environment for Model Checking of Software components.

[38] ANT, http://ant.apache.org. http://fractal.objectweb.org/.

[39] Fractal ADL. http://fractal.objectweb.org/tutorials/adl/index.html.

Appendix A

User Documentation

A.1 Installation Manual

A.1.1 Requirements and Prerequisites

Carmen is a program developed as an extension of the Java PathFinder tool, which was written in the Java programming language. Java PathFinder is highly sophisticated verification tool, its hardware requirements was even increased by integrating prototype extensions. These requirements for running the application can be summarized as follows:

- Windows/Linux OS
- Java[tm] runtime 1.4 or above (1.5.1 or above highly recommended)
- At least 150MB of free memory.

Note: Although Carmen was developed as an extension for Java PathFinder, the user does not need to install these two parts of application separately. The original Java PathFinder is included in Carmen application and therefore is not mentioned as a prerequisite.

Carmen has been tested and validated on the following platform: Windows XP and Fedora Core 4. Although the application was developed on Java 1.4, successful testing was performed also with Java version 1.5, which is also highly recommended to use when launching Carmen.

A.1.2 Installing Carmen

Once the prerequisites of the program are satisfied, no installation procedures are needed. Simply copy the content of the distribution directory to directory reserved for Carmen and run Carmen, see Chapter A.2.

Modifying the Carmen Implementation As a part of the distribution, also the source code of the application is included. If a user wants to, he or she may modify the source code. In order to introduce these modifications into the version of Carmen that is being executed, it is necessary to perform a few tasks. Following procedure is describing necessary steps in detail.

Procedure:

1. Close running Carmen
2. Execute the ANT target "jar" (see *build.xml* file)
3. Run Carmen again

A.1.3 Content of the Distribution Directory

Figure A.1 is listing the content of the distribution directory.

Note: Due to the license agreement of the Protocol Checker tool [10], the source codes of the Protocol Checker are not contained.

bin/	Scripts for running Carmen
test/	Scripts for sample tests
carmen.bat	Win script for running Carmen
carmen	UNIX script for running Carmen
build/	Carmen build files
documentation/	Carmen documentation in PDF format
API/	Contains Carmen API documentation in HTML format
thesis/	Contains the master thesis in PDF format
userDoc.pdf	User Documentation
develDoc.pdf	Developer Documentation
input/	Recommended directory for placing components specifications and builds
build/	Component build files
components_xml/	Values specification files
basic-proto-Carmen.dtd	DTD for validatingthe component ADL files
specifications.xsd	XSD Schema for value specification files
sample.xml	Sample value specification file
Demo.fractal	Fractal ADL file used to run the sample tests
marketplace.fractal	Fractal ADL file to run the sample tests
lib/	Contains the .jar files used for running Carmen
externals/	External .jar files used for running Carmen
carmen.jar, jpf.jar, checker.jar, env_jpf.jar, env_jvm.jar	
output/	Contains output files of executed Carmen tests
src/	The source code of Carmen
build.xml	ANT file containing targets for running tests, builds, ...
default.properties	JPF properties file
jpf.properties	JPF properties file

Figure A.1: Content of the Distribution Directory

A.2 Running Carmen

This chapter describes a configuration and an execution of the component verification process. Before executing Carmen, it is highly recommended to study the execution procedures of the original Java PathFinder tool described in [4].

There are several ways to run Carmen. Sections A.2.1, A.2.2 and A.2.3 are describing these procedures in more details.

Before the verification can be launched, it is also necessary to provide to the tool build files of the component that is going to be verified, see Section A.2.4.

Section A.2.6 describes how to launch numerous sample tests which demonstrates the functionality of the tool.

A.2.1 Command Line Execution

Executing Carmen from the command line is easy:

```
> bin/carmen [-c config-file] [-show] [+key=value] [-o=outputFile] [-p=value]
ADL-specification-file component-name
```

Argument description:

- **-c config-file** optionally
- **-show** directs JPF to print out the configuration key/value pairs prior to running the application
- **+key=value** is convenient way to override configuration properties via the command line
- **-o=outputFile** redirects the output of the application to the 'outputFile'
- **-p=value** set the verbose level, see Section A.5.1
- **ADL-specification-file** - path to an ADL file containing specification of the component that is to be verified, for the definition of the ADL and other specification files see Section A.4
- **component-name** - the name of the component that will be verified

Note: There are two execution scripts, *carmen.bat* for Windows platforms and *carmen* for UNIX platforms.

A.2.2 Direct Execution

In case the user does not want to use the *bin/carmen* scripts, he or she may run Carmen directly:

```
> java vm-args cz.cuni.mff.jpf.carmen.Carmen carmen-args
```

carmen-args represents the same set of arguments described in the Command Line Execution in A.2.1.

Before using this procedure, make sure that the following code is reachable:

- .jar files contained in the *lib/* directory and its subdirectories
- optionally - your additional JPF extension classes (Listeners, properties etc.)

When launching Carmen directly, it is also recommended to increase the maximum heap space with the `-Xmx` VM argument (e.g. -Xmx1024m).

A.2.3 Embedded Execution

Similarly to JPF, Carmen can also be used embedded (e.g. an IDE), i.e. called from another Java application. A basic code sequence to start Carmen looks like this:

```
...
import cz.cuni.mff.jpf.carmen.Carmen;
...
void runCarmen(String[] args) {
  ...
  MyListener listener = new MyListener(..);
  listener.filterArgs(args);
  ...
  Carmen carmen = new Carmen(args);
  carmen.addVMListener(listener);
  carmen.runVerification();
  ...
}
```

A.2.4 Setting Sources

For the purpose of the verification, the tool needs to be provided with .class files of the component that is going to be verified. It is also necessary to place the .class files into the directory to which JPF VM can access.

For the convenient, the distribution directory structure contains the directory *input/build/* that belongs to the vm.sourcepath (the CLASSPATH of JPF VM, see [4]). Therefore, it is recommended to place the .class files of the component into this directory.

If the user wants to, he or she may change the vm.classpath and the tool will search for .class files in different directory. To change the vm.classpath, it is necessary to modify the *jpf.properties* file.

Note: The vm.classpath directory has to contain the directory *cz/cuni/mff/jpf/init/* with files *Substitute.class* and *Subs.class* . These files are used by Carmen during its initialization.

A.2.5 Host VM Execution

In order to keep the JPF state space as small as possible, JPF is capable to execute the non relevant part of the code in the Host Virtual Machine. This introduces a state space reduction and accelerates the verification.

Build files of those parts, which will be executed by Host VM have to be placed in *lib/env_jvm.jar*. Use ANT, Section A.2.6, to create the necessary jars.

See JPF documentation [4] for more details.

A.2.6 Test Execution

This approach of execution is intended to provide a convenient way to launch sample verification tests.

The distribution contains several component implementations with necessary specification files in order to demonstrate features of the application.

This Section presents possible approaches to the sample tests execution.

Table A.2 provides an overview and basic orientation to all sample verification tests which are included. The performance statistics of selected tests can be found in Section 7.2.

Test	Component	Notes
1	FlyTicketClassifier	
2	FlyTicketClassifier_err	Similar to Test 1, but contains a protocol violation, see 6.2.4
3	ValidityChecker	
4	ValidityChecker	With a simplified behavior protocol
5	Arbitrator	
6	Arbitrator	With a simplified behavior protocol
7	IpAddressManager	
8	IpAddressManager	Demonstrates a protocol violation, see 6.2.4
9	FlyTicketClassifier	Demonstrates solution of the alternative operator problem
10	Seller	The alternative operator problem on a simple component
11	AccountDatabase	Verification against complicated behavior protocol

Figure A.2: Sample Verification Tests Overview

ANT Execution

If the user has the ANT application properly installed (see [38] for the latest version), he or she may run the sample test through the targets provided by the *build.xml* file.

Note: The *build.xml* file contains also other targets, which are not inherently necessary to run Carmen. These targets (build, jar,...) may be used in special cases.

Direct Execution

The *bin/* directory contains subdirectory *tests/* which contains scripts, each for every sample test, that are able to launch verification tests directly from the command line.

A.3 Example Component Verification

This section presents all the steps necessary to launch Carmen.

Note: The tutorial presents verification of the *FlyTicketClassifier* component. All the specification and .class files can be found in the *input/* directory.

The step by step tutorial follows:

1. Set the Behavior Protocol File

1. Create the `FlyTicketClassifier.bp` file containing a frame protocol of the component.

2. Append thread suffixes to every event of the behavior protocol, see Section A.4.2 for detail instructions.

2. Set the Value Specification File

1. Check that only supported data types, listed in Table A.7, are used as parameters of required and provided methods of the component. If there are used any unsupported data types, it is necessary to modify the component implementation.

2. Create the Value Specification file, where value restriction for every parameter of a required/provided method will be specified. This process is described in Section A.4.3.

3. Modify the ADL File

Before the verification can be launched, following modifications of the Fractal ADL file are required:

1. In order to extend the ADL file, it was necessary to modify its DTD definition. A new DTD definition file was created, this file can be found in *input/*. Therefore, replace the original DTD definition line with
```
<!DOCTYPE definition SYSTEM "basic-proto-Carmen.dtd">
```

2. For the element `<component name="FlyTicketClassifier">` describing an architecture of the component that is to be verified, add following subelements:
2.1 Specify the behavior protocol file:
```
<protocol file="input/protocols/FlyTicketClassifier.bp"/>
```
2.2. Specify the Value Specification file:
```
<environment>
<valuesets classname="input/specifications/FlyTicketClassifier.xml"/>
</environment>
```

4. Set the Component Implementation

1. Place the *.class* files of the component into the directory specified in the *vm.classpath*, default is *input/build/*. See Section A.2.4 for more details.

5. Run Carmen

Having all the previous steps accomplished, the verification process can be launched by one of the ways specified in Section A.2, for example:

```
> bin/carmen -o=output/test1 -p=err input/Demo.fractal FlyTicketClassifier
```

A.4 Component Specification Files

For the verification of the component we need its source code, but only the source code itself is not sufficient for starting the verification process. It is also necessary to define the correct communication between the component and its environment.

For this reason are used the component specification files which describe the component interfaces and its behavior protocol. These files are containing also other properties modifying the verification process.

This chapter is describing structures of specification files and the procedure to create the specification files for the component that is to be verified.

The specification data are stored in the XML format.

A.4.1 Fractal ADL

The Fractal ADL file was introduced by the Fractal Component Model, it describes architecture of component systems, see [39]. For our purpose are interesting descriptions of primitive components which define their required and provided interfaces.

The Fractal ADL file uses the XML based syntax and can be easily extended.

```
<protocol file="input/protocols/arbitrator.bp"/>
```

Figure A.3: Behavior Protocol File Extension to ADL

The format of the Fractal ADL file was further extended in order to contain the information necessary to verify a primitive component. Two elements were added: the element specifying a file containing a behavior protocol, depicted on Figure A.3, and the element referring to a file containing values specifications, depicted on Figure A.4.

A.4.2 Behavior Protocol File

By a behavior protocol we mean a frame protocol of the component that will be passed to BPChecker. It is necessary to define behavior protocol with thread suffixes for every event of the protocol.

The behavior protocol file contains only the protocol itself, no XML tags are used. The ADL file specifies where the behavior protocol file to a given component can be found.

Thread Suffixes

To face the parallelism problem, see 4.3.3, the thread suffixes were introduced. Before the verification can be started, the behavior protocol of the component

```
<environment>
  <valuesets classname="input/specifications/FlyTicClassif.xml"/>
</environment>
```

Figure A.4: Values Specification File Extension to ADL

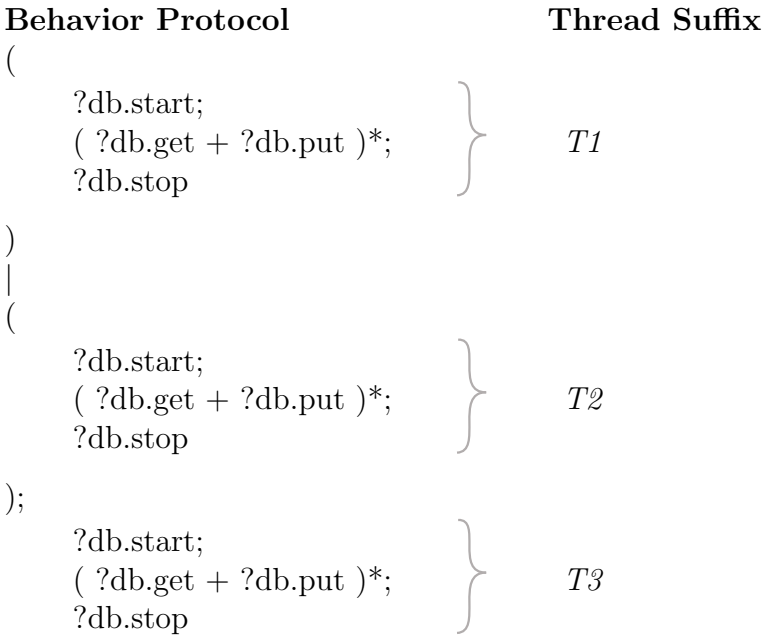


Figure A.5: Adding Thread Suffixes

has to be modified, it is necessary to append to each event token a thread suffix string.

The thread suffixes are specifying which method call response belongs to which method call request. Since there can be more then one invocation of the same method in the behavior protocol, to avoid these collisions, it is needed to define a unique thread suffix for every method call.

Theoretically, as a thread suffix can be used every sequence of characters and the application allows defining for every method call its unique suffix. But for the practical reasons, it is recommended to define one unique thread suffix for a sequence of events that will be executed be one thread.

This informally means that the user has to divide a behavior protocol into parts according to the appearance of the parallel operators, see Fig A.5. After that, to each of this part has to be assigned its thread suffix (e.g. string "T1","T2",...) which is then subsequently appended to every event-token from this fragment of the protocol.

Figure A.6 is showing behavior protocol with appended thread suffixes. Please note that although to execution of this protocol two threads are needed, thread suffix "T3" is also appended. The "T1" or "T2" suffix could be also used instead of the "T3" but appending unique suffix to every fragment is recommended in order to avoid possible conflicts and errors in complex protocols.

```
(
        ?db.start:T1;
        ( ?db.get:T1 + ?db.put:T1 )*;
        ?db.stop:T1
(
|
(
        ?db.start:T2;
        ( ?db.get:T2 + ?db.put:T2 )*;
        ?db.stop:T2
);
        ?db.start:T3;
        ( ?db.get:T3 + ?db.put:T3 )*;
        ?db.stop:T3
```

Figure A.6: Behavior Protocol with Thread Suffixes

Since behavior protocol syntax allows assigning response and request method calls almost arbitrarily, this approach is not working for complicated protocols. Because only the designer knows which response is corresponding to which request, it is under his responsibility to correctly define the thread suffixes.

A.4.3 Values Specification File

For the purpose of the environment simulation, the application requires to define value restrictions of parameters for every provided and required method.

The values specification file is using the XML format, therefore, the XSD file *input/specifications.xsd* can be used for the validation.

Although application allows avoiding values restrictions for the specific data types (e.g. int, char, ...), it is highly recommended to specify restrictions for these data types. Otherwise, the state space of the application will be enormous which directly affects the time consumed by the verification process.

There are two types of restrictions:

- Enumeration - the user can enumerate possible values for the specific method parameter
- Interval - the user can define an interval of values, which will be considered as possible values

The table A.7 is listing all data types and possible value restrictions, which can be used when defining the input/output parameters for methods.

Data Type	No Restriction	Enumeration	Interval
Boolean	No Restrictions Needed		
Short	✓	✓	✓
Byte	✓	✓	✓
Char	✓	✓	✓
Integer	✓	✓	✓
Long	✗	✓	✗
Float	✗	✓	✗
Double	✗	✓	✗

Figure A.7: Value Restrictions for Data Types

Other Properties

Component Class This property specifies the main class representing the component implementation. The instance of this class, which in fact represents the component, is verified by the application. See the Section A.2.4 and the chapter "*Configuring JPF Runtime Options*" in [4] for instructions how to specify the classpath used for loading the classes required by the component implementation.

Cycle Limit The cycle limit property defines restrictions for the usage of the repetition operator (see Section 4.3.5). The value specified by this property is an integer value greater then 1.

Summarization

Table A.8 is summarizing all elements which have to be contained inside the values specification file.

A.5 User Interface

A.5.1 Output Messages

To inform the user about the progress of the verification process, messages describing a current state of the process are being printed to the output.

The application is able to inform the user very thoroughly about each step of the verification process. Since there are usually thousands of states explored during the

Element	Description
Interfaces	Required and Provided interfaces needs to be defined
Provided Interface	Input values restrictions necessary for every method
Required Interface	Output values restrictions for every method necessary
Method Description	For every required method specify: • Name and a values restriction for output parameters For every provided method specify: • Name and values restrictions for every input parameter
Value Restrictions	• Value restrictions to specify values of input/output parameters of a method • For data types `void` and `boolean` leave the element field empty

Figure A.8: Values Specification File Elements

process of the verification, printing messages about every action would adversely affected performance of the program.

In order to provide the user possibility to balance number of messages with performance of verification, the application allows setting the verbose level for messages printing.

Additionally, when running the application by embedding it inside another program, see Section A.2.3, the user can define, by employing the Search-/VMListeners techniques (described in [4]), his own way to collect statistics or even extend functionality.

Recommendation For the best performance results, it is recommended to launch the application with verbose level set to *err*, which is printing only the error messages. Beside possible error reports, the program will print short notice every 100th reached state.

Verbose Level

Messages are evaluated by priorities, therefore the user can determine which messages will be omitted and which will be printed to the output.

The user can specify, which messages will be printed, by setting the input argument **-p=value**, see also Section A.2.1. The following table is showing possible verbose levels.

Parameter Value	Description
err	Only errors
war	Errors and warnings
states	Errors, Warnings and Entered States
all	All messages are printed

Initialization Process Messages

The application is starting be reading the input file and verifying the format of a behavior protocol.

```
Reading the input file "FlyTicketClassifierImpl.xml" :
Instantiating Manager...
Checking the Behavior Protocol of the component...
    ->Predefined thread-suffixes has been found.
Manager Successfully Instantiated.
Instantiating the Java PathFinder...
Java PathFinder Successfully Instantiated.
```

Then we can launch Java PathFinder and initialize it:

```
----- starting JPF on class:cz.mff.jpf.FlyTicketClassifierImpl
----- JPF Initialization started.
```

At first we have to initialize Manager, which is managing the progress of the verification:

```
---- Manager initialization...
     Validating the Events... finished successfully.
     Validating the Restrictions... finished successfully.
---- Manager initialization successful.
```

And now we can initialize Java PathFinder itself. This process will finish by creating the first state (with empty trace and number 0):

```
threadStarted: 0
Instruction executed: 0 : invokestatic java/lang/Object/...
Instruction executed: 0 : return
Instruction executed: 0 : iconst_0
Instruction executed: 0 : anewarray java/io/ObjectStreamField
...
Instruction executed: 0 : astore_1
Instruction executed: 0 : return
>[Manager]      0     0
         trace :
--------------- JPF Initialization Finished successfully.
```

Verification Process Messages

These messages are describing the progress of the verification.

The verification process is starting by the message

```
------------------- Search started
```

When the verification is running, two general types of messages can be obtained:

- Message describing current state
- Message describing currently executed instruction

State Advanced Messages

Note: State Advanced Messages are printed only when verbose level is set to "*states*" or "*all*".

Manager has inserted an event on provided interface of the component, therefore the new state was generated:

```
> [Manager]      1     1
  trace : ?iFlyTicketDb.IsFlyTicket(int FlyTicketId=0):0^
```

State was backtracked:

```
< [Manager]      3     3
  trace : ?iFlyTicketDb.IsFlyTicket(int FlyTicketId=0):0^
```

Reached state that was already visited. JPF will backtrack in the next step:

```
* [0]      2     4  :Source= FlyTicketClassifierImpl.java:137
  trace : ?iFlyTicketDb.IsFlyTicket(int FlyTicketId=0):0^;...
```

State was processed, all execution paths leading from this process were explored, JPF will backtrack:

```
. [Manager]      3     3
  trace : ?iFlyTicketDb.IsFlyTicket(int FlyTicketId=0):0^;...
```

Instruction Executed Messages

Instruction Executed Messages are printed only when the verbose level is set to "*all*".

An instruction was executed:

```
Instruction executed: 0 : aconst_null
Instruction executed: 0 : areturn
```

When the JPF is invoking a required method, some instructions can not be executed and has to be skipped, see Section 5.1.2. Application is then printing:

```
Instruction skipped: 0 : aconst_null
Instruction skipped: 0 : areturn
```

Summarization Statistics

After the verification, the application will print the statistics and then the list of errors which have been identified by the verification process:

```
...
-------------------- Search finished

============================================================
Statistics:
    State space: 922 unique states
                 1920 visited states
    Elapsed time: 4 seconds



==================================
Initialization errors:    No Errors Found
Java PathFinder errors:   No Errors Found
BehaviorProtocol errors:  No Errors Found
==================================

---------------- JPF terminated
```

Error Messages

There are three general types of error:

Initialization Error
Found when preparing for the verification. When such an error is identified, the initialization will fail and the program ends.

Java PathFinder Error
Identified by the JPF during the verification: deadlocks, property violations,...

Behavior Protocol Error
Reports the violation of the behavior protocol
Example:

```
Behavior Protocol Error detected
  Error Message: Manager has detected the Event,
              that is not in compliance with Behavior Protocol.
  Event:!iIpMacTransientDb.Add:0^
  Protocol Checker trace:?iDhcpServerController.Start:0^,...
  State number: 18471
  Depth: 5
  Error detected in: callRequestDetected()
  StackTrace: at IpAddressManagerImpl.RequestNewIpAddress
                 (IpAddressManagerImpl.java:275)
```

Appendix B

Behavior Protocol Examples

```
(
  (
    ?ILogin.LoginWithFlyTicketId:0
    ;
    ?ILogin.Logout:0
  )
  |
  ?ITokenCallback.TokenInvalidated:1
  |
  ?IDhcpCallback.IpAddressInvalidated:2
  |
  ?IDhcpCallback.IpAddressInvalidated:3
)
```

Figure B.1: Arbitrator Component, Behavior Protocol

```
(
  (
    (
      ?ILogin.LoginWithFlyTicketId:0
      ;
      ?ILogin.Logout:0
    )
    |
    ?ITokenCallback.TokenInvalidated:1
  )
  ;
  ?IDhcpCallback.IpAddressInvalidated:2
  ;
  ?IDhcpCallback.IpAddressInvalidated:3
) + (
    (
      ?ITokenCallback.TokenInvalidated:5
      |
      ?IDhcpCallback.IpAddressInvalidated:6
    )
    ;
    (
      ?ILogin.LoginWithFlyTicketId:7
      ;
      ?ILogin.Logout:8
    )
    ;
    ?IDhcpCallback.IpAddressInvalidated:9
) + (
    (
      ?IDhcpCallback.IpAddressInvalidated:10
      |
      ?IDhcpCallback.IpAddressInvalidated:11
    )
    ;
    (
      ?ILogin.LoginWithFlyTicketId:12
      ;
      ?ILogin.Logout:13
    )
    ;
    ?ITokenCallback.TokenInvalidated:14
)
```

Figure B.2: Arbitrator Component, Simplified Behavior Protocol

```
(
  (
    ?IToken.SetEvidence:0
    |
    ?IToken.SetValidity:1
    |
    (
        ?IToken.SetAccountCredentials:2 {
            !ICustomCallback.SetAccountCredentials:2
         }
        +
        NULL
    )
  )
  ;
  ?ILifetimeController.Start:3^ ;
  !ITimer.SetTimeout:3^ ;
  [?ITimer.SetTimeout:3$, !ILifetimeController.Start:3$]
)
;
(
  ?IToken.InvalidateAndSave:4 {
      !ITimer.CancelTimeouts:4;
      (!ICustomCallback.InvalidatingToken:4 + NULL);
      !ITokenCallback.TokenInvalidated:4
  }*
  |
  ?IToken.InvalidateAndSave:5 {
          !ITimer.CancelTimeouts:5;
          (!ICustomCallback.InvalidatingToken:5 + NULL);
          !ITokenCallback.TokenInvalidated:5
  }*
  |
  ?IToken.InvalidateAndSave:6 {
      !ITimer.CancelTimeouts:6;
      (!ICustomCallback.InvalidatingToken:6 + NULL);
      !ITokenCallback.TokenInvalidated:6
  }*
  |
  ?ITimerCallback.Timeout:7 {
      (!ICustomCallback.InvalidatingToken:7 + NULL);
      !ITokenCallback.TokenInvalidated:7
  }*
)
```

Figure B.3: ValidityChecker Component, Behavior Protocol

```
(
  (
    ?IToken.SetEvidence:0
    |
    ?IToken.SetValidity:1
  )
  ;
  ?ILifetimeController.Start:3^ ;
  !ITimer.SetTimeout:3^ ;
  [?ITimer.SetTimeout:3$, !ILifetimeController.Start:3$]
)
;
(
  ?IToken.InvalidateAndSave:4 {
      !ITimer.CancelTimeouts:4;
      (!ICustomCallback.InvalidatingToken:4 + NULL);
      !ITokenCallback.TokenInvalidated:4
  }*
  |
  ?IToken.InvalidateAndSave:5 {
          !ITimer.CancelTimeouts:5;
          (!ICustomCallback.InvalidatingToken:5 + NULL);
          !ITokenCallback.TokenInvalidated:5
  }*
  |
  ?ITimerCallback.Timeout:7 {
      (!ICustomCallback.InvalidatingToken:7 + NULL);
      !ITokenCallback.TokenInvalidated:7
  }*
)
```

Figure B.4: ValidityChecker Component, Simplified Behavior Protocol