

Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Nodir Yuldashev

Using Java PathFinder for Construction of Abstraction of Java Programs

Department of Software Engineering
Advisor: RNDr. Pavel Parízek, Ph.D.
Study Program: Computer Science, Software Systems

First of all, I would like to thank my advisor for his patience and invaluable remarks and suggestions. I also want to thank my family for their support in my studies.

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, April 15, 2009

Nodir Yuldashev

Table of Contents

| | |
|---|----|
| Chapter 1. Introduction..... | 6 |
| 1.1 Goals of the work..... | 7 |
| 1.2 Structure of the text..... | 8 |
| Chapter 2. Background..... | 9 |
| 2.1 Software Components..... | 9 |
| 2.1.1 Example..... | 11 |
| 2.2 Behavior Protocols..... | 14 |
| 2.2.1 Example..... | 16 |
| 2.3 Java PathFinder..... | 18 |
| 2.3.1 State space explosion problem..... | 18 |
| 2.3.2 Search- and VMListeners..... | 19 |
| 2.3.3 Choice generators..... | 22 |
| Chapter 3. Construction of abstraction..... | 24 |
| 3.1 Algorithm outline..... | 24 |
| 3.2 Simplification to single-threaded applications..... | 25 |
| 3.3 Generalization to multi-threaded applications..... | 28 |
| 3.4 Notes on operators' processing..... | 30 |
| 3.4.1 Sequence operator..... | 30 |
| 3.4.2 Alternative operator..... | 32 |
| 3.4.3 Repetition operator..... | 33 |
| 3.4.4 And-parallel operator..... | 34 |
| Chapter 4. Implementation details..... | 38 |
| 4.1 Program input..... | 39 |
| 4.2 Detection of events..... | 40 |
| 4.1.1 Processing method invocations..... | 42 |
| 4.1.2 Processing RETURN instruction..... | 42 |
| Chapter 5. Evaluation..... | 44 |
| 5.1 Influence of parallelism on operators' detection..... | 47 |
| 5.2 Case study..... | 49 |

| | |
|---|----|
| Chapter 6. Related work..... | 52 |
| Chapter 7. Conclusion..... | 56 |
| Bibliography..... | 57 |
| Appendix A. List of used abbreviations..... | 59 |
| Appendix B. Contents of CD ROM..... | 60 |
| Appendix C. User manual..... | 61 |
| C.1 Installation..... | 61 |
| C.2 Running BytecodeAbstractor..... | 61 |
| C.2.1 Providing target application sources..... | 62 |
| C.2.2 Command line execution..... | 62 |
| C.2.3 Execution using BytecodeAbstractor GUI..... | 62 |
| C.3 BytecodeAbstractor GUI description..... | 63 |
| C.3.1 Generating program input..... | 64 |
| C.4 Sample component abstraction..... | 65 |

Název práce: *Using Java PathFinder for Construction for Abstraction of Java Programs*

Autor: *Nodir Yuldashev*

Katedra (ústav): *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *RNDr. Pavel Parízek, Ph.D.*

e-mail vedoucího: *Pavel.Parizek@mff.cuni.cz*

Abstrakt: *S rostoucí složitostí moderních softwarových systémů se jejich verifikace stává velmi obtížnou úlohou. Techniky formální verifikace a analýzy slouží k nalezení chyb v kódu nebo pro prokázání, že kód splňuje určité vlastnosti. Populární technika automatické verifikace je model checking, který využívá procházení stavového prostoru. Nicméně model checking je náchylný k problému exponenciálního nárůstu počtu stavů (state explosion) a proto nemůže být použit pro složité vícevláknové softwarové systémy. Obecné řešení tohoto problému (state explosion) spočívá ve vytvoření abstrakce cílového systému a následném použití tohoto modelu k verifikaci. V rámci diplomové práce jsme navrhli a implementovali nástroj pro konstrukci abstrakce Java komponent v jazyce behavior protocol, který využívá model checker Java PathFinder pro procházení stavového prostoru. Výsledky experimentů na několika netriviálních komponentách ukazují, že nástroj může být použit v praxi.*

Klíčová slova: *behavior protocols, Java PathFinder, state explosion, model checking, softwarové komponenty*

Title: *Using Java PathFinder for Construction for Abstraction of Java Programs*

Author: *Nodir Yuldashev*

Department: *Department of Software Engineering*

Supervisor: *RNDr. Pavel Parízek, Ph.D.*

Supervisor's e-mail address: *Pavel.Parizek@mff.cuni.cz*

Abstract: *The growing complexity of software systems makes the verification of the systems very difficult. Techniques of formal verification and analysis are used to find bugs in the code, or to prove that the code satisfies some properties. A popular automated verification technique is model checking, which uses state space traversal. However, model checking is prone to state explosion and therefore does not scale to complex multi-threaded software systems. Common solution to this problem (state explosion) is to create an abstraction of the target system, and then verify only the abstraction. We have designed and implemented a tool for construction of abstraction of Java components in behavior protocols, which is based on the Java PathFinder model checker. Results of experiments on several non-trivial components show that the tool can be used in practice.*

Keywords: *behavior protocols, Java PathFinder, state explosion, model checking, software components*

Chapter 1. Introduction

Software systems nowadays form an important part of our life. They are replacing more and more parts of all types of human activity. And with the permanent integration into human activities the size of software systems increases dramatically, which also results in growing complexity. This raises an issue of maintainability of modern software systems.

Taking into account growing complexity of computer software, it seems that Component Based Software Engineering (CBSE) offers convenient solution to this problem. It is a paradigm, which states that computer software can be built-up from software components, much like hardware components, from which a complex computer systems are built.

Software systems are built in modular manner, from well-defined units called software components. Software components are independent deployment units with well-defined interfaces. Components can cooperate with each other only by these interfaces, which makes them reusable. In this sense, they can be viewed as black boxes, which are replaceable with another black box with the same interface.

Also, with the growing complexity of software systems, verification of the systems becomes very difficult. Techniques of formal verification and analysis are used to find bugs in the code, or to prove that the code satisfies some properties. But employing the techniques of formal verification manually is not feasible anymore with modern systems. Automated techniques have to be used instead in order to verify the system correctness.

There exist, in general, two types of automated techniques of formal verification: *automated theorem proving* and *model checking*. First technique infers proof of system correctness from a description of a system, which is represented by the sets of axioms and inference rules. Second technique – model checking verifies given properties by traversing the system's state space.

Model checking can be used to automatically check compliance of a given software component to the given specification. Model checking makes use of state space traversal, which is prone to state explosion. So, using this technique to check

commercial software seems ineffective, as software nowadays often have more than millions of lines of codes.

Two approaches for addressing state explosion problem exist: *abstraction* and *compositional verification*.

Instead of verification of code, it is possible to derive an abstraction of code and then model check the abstraction against some properties. Many languages and formalisms for abstraction of behavior of code were proposed over the past years – preconditions + postconditions, trace-oriented formalisms (LTS, process algebras, behavior protocols, etc.) and others.

Techniques of compositional verification exploit the structure of component systems in order to make verification more feasible (verification techniques typically do not scale well).

Compositional verification is possible only if an abstraction of each component is available – when verifying a single component, abstraction of the rest of the system (all other components in the application) is used.

In this thesis, we are interested in sequences (traces) of method calls (important events), and therefore we focus on trace-oriented formalisms that allow to model/specify traces of important events (e.g. method calls). Several types of trace-oriented formalisms exist: finite state machine, LTS, process algebras, behavior protocols.

1.1 Goals of the work

As was mentioned above, efficient compositional verification requires an abstraction of code. Our goal in this thesis is to create an effective tool for automated extraction of abstraction of code which is able to detect the parallelism and which will be based on state space traversal. In order to achieve this goal some techniques to fight state explosion problem will have to be developed. We focus on Java code and we chose behavior protocols (BP) as the formalism to be used in this thesis, since, unlike FSM and LTS formalisms, they allow to model parallelism explicitly. In the FSM and LTS formalisms the parallelism is encoded into choice among sequences of

interleavings, i.e. parallelism is modeled implicitly. Other process algebras, which allow to model parallelism, have the disadvantage of not being close to code.

BP are the modeling formalism, which is similar to regular expressions or process algebras. They support operators for sequences, choice, parallelism and repetition. In BP, method invocations and returns on component interfaces are atomic events.

The tool will produce an abstraction of implementation of the given software component by recording only method calls on interfaces required by this component. It will also generate a model of interaction of the component with its environment (all other components in software system) by recording only method calls on interfaces provided by given component. Our tool will be based on the Java PathFinder, which is a model checker for Java programs.

1.2 Structure of the text

Chapter 2 provides a background, which gives a deeper insight into software components, behavior protocols, and to the Java PathFinder model checker used in the thesis. An algorithm for construction of an abstraction of a software component is described in Chapter 3. Specific details of the implementation, such as input file format, program GUI and detection of method calls and returns from them are described in Chapter 4. Chapter 5 evaluates chosen algorithm given in previous chapter. The rest of the text contains related work in Chapter 6 and conclusion in Chapter 7.

Chapter 2. Background

This chapter will provide the reader with the necessary background information.

Section 2.1 describes software components in more details including a component-based software system (CBSS) example. Section 2.2 introduces behavior protocols and gives an example of behavior protocol of a component of CBSS given in Section 2.1. Finally, Section 2.3 provides information on Java PathFinder, including strategies the tool applies to fight state space explosion problem and the mechanism of choice generators, which are used to systematically explore state space of a Java program. Also, this section briefly describes the JPF extension mechanism, such as SearchListener and VMListener interfaces.

2.1 Software Components

Informally, software components are the building blocks of software. And, as such, they can be viewed as black boxes, functionality of which is not visible externally. Software components should also be reusable in different contexts. Formally, a software component is defined as “a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties” [3].

Components can be bounded together using their interfaces. We differ two types of interfaces: *required* and *provided*. Required interfaces are the interfaces specifying the services required by the given component in order to function properly. Required interfaces are provided by another components in the system. Provided interfaces are the interfaces, which the given component offers to its clients. Using analogy to hardware components, interfaces are the sockets, slots and connectors by which the hardware components are connected together. Figure 1 illustrates the required and provided interfaces and their schematic presentation. As seen from the figure, a provided interface of one component is bound to a required interface of another component.

The software component's contract, specifying its expected use by the clients and

reactions to such valid usage has to be defined in a formal way. Some kind of formal behavior specification can be used to achieve this. Component's behavior can be specified via pre-/postcondition pairs or via event-trace based formalism, e.g. process algebras.

Component-based software systems are designed and implemented according to some specific component model. Component model is a framework specifying set of rules and concepts describing the behavior of individual components as well as a complete component-based system. These set of rules can be defined using an architecture definition language (ADL), which allows to specify a structure of a component-based software system.

Component models are divided into two types – *flat* and *hierarchical* component models. Hierarchical component models allow nesting of software components whereas flat component models do not. Thus, all components in flat component models are primitive. Hierarchical component models, on the other hand, can contain both primitive and composite software components, the consequence directly following from component nesting. In composite components, primitive components represent leafs of a hierarchy of composite components. They can be viewed as a black boxes. In this sense, composite components are gray boxes composed of nested subcomponents, which are interconnected via bindings on interfaces.

Flat component models are typically industry-developed. These models are, therefore, simple, and corresponding platforms consisting of a component model and a runtime environment for the components are more complete and stable. Examples of a flat component models are, e.g. EJB [7] and COM [11].

Development of hierarchical models-based component platforms (e.g. SOFA [5], Darwin [12], and Fractal [4]) is driven by academic world. Due to this fact, these platforms have many advanced capabilities, such as behavior modeling and verification, etc. Though, on the other hand, most of them have limitations, such as very limited runtime environment. And, as these component models have specific goals, they mainly provide tools for the design of a component application and do not take into account its implementation.

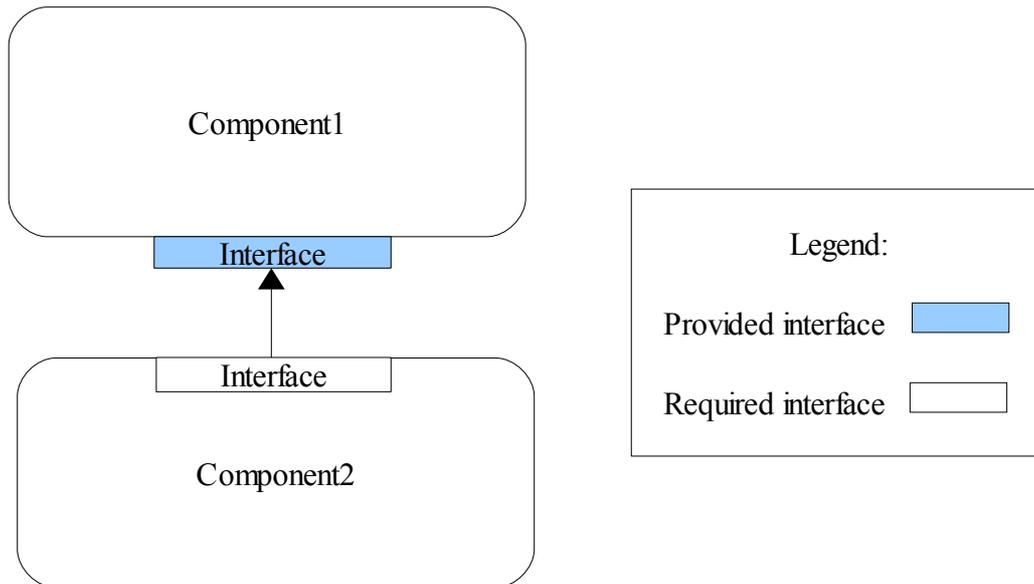


Figure 1: Provided and required interfaces

2.1.1 Example

Here we give an example of a component system – a demo application for Fractal component model [4]. It is an airport service for providing wireless Internet connection. The connection is granted to the owners of the frequent flyer card if they have a valid ticket and to the owners either of the first class or business class tickets and other passengers prepay the connection time by a credit card if they want wireless Internet connection. Client's session starts as soon as the client authenticates and terminates when one of the following events occurs:

- the client disconnects from the wireless network – any prepaid time not used up during the session being terminated can be used up in future sessions assuming the client's user name will not expire until then;
- client's fly ticket becomes invalid or all of the client's prepaid time is used up – the session terminates immediately and the client can start a new prepaid session.

The key part of the system is implemented in Fractal components, although clients communicate with the system via JSP web pages. Environment in this example is divided into three areas/networks:

- a) Airport WiFi – The public wireless network that clients connect to.
- b) Airport LAN – All Fractal components run on computers in this network. The communication between this network and the Airport WiFi is separated by the Firewall that is controlled by the Firewall Fractal component.
- c) Internet – The part representing the outer world. It hosts central servers and web services (e.g. credit card web services, air-carriers database servers) and also the client communication goes there (if not blocked by the Firewall).

On the lowest level the client connections are managed by the DhcpServer Fractal component. This component assigns IP addresses to new clients and notifies other demo Fractal components when clients disconnect, so that their sessions can be terminated. The DhcpServer component might also communicate with some sort of wireless network access point, in order to get more accurate information about client connection and disconnection events.

Figure 2 provides an overview of the application being discussed. It can be seen that the application consists of several software components, e.g. FrequentFlyerDatabase, Token, Card Center, Arbitrator, etc. Most of them are hierarchical components which contain smaller components, e.g. FlyTicketDatabase (contains FlyTicketClassifier, AfDbConnection, CsaDbConnection components) and Token (contains Timer, ValidityChecker, CustomToken components) components.

2.2 Behavior Protocols

Behavior protocols [13] is a formalism for modeling of a component behavior. It describes the component behavior – communication of the component with its environment and, communication with its subcomponents in case it is a composite component. In this sense, behavior protocol is a regular language for specification of component behavior. This simple process algebra is used by SOFA and Fractal component models.

More formally, behavior protocol is an expression which enlists a set of finite *traces* of atomic *events* on components' provided and required interfaces. In behavior protocols, events directly map to events on implementation level, such as method invocations on interfaces and returns from those methods.

We distinguish between two types of events: *request* and *response*, which correspond to method invocation and return from method, respectively. Thus, these two types of events make up a single method call. In behavior protocol syntax, request is denoted by “^” or “↑” and response – by “\$” or “↓”. In our implementation and throughout the thesis the first notation will be used.

Also, we must distinguish between method calls on required and provided interfaces, because from the component's perspective an event can be *emitted* or *accepted*. Method calls on required interfaces in behavior protocols are denoted by “!” (emission of an event), whereas method invocations on provided interfaces are denoted by “?” (accepting an event).

We call *event token* a smallest behavior protocol unit, which satisfies the formula:
[emit | accept] event [request | response].

Taking into account above said, there are four types of event tokens:

1. ?interface.method^ (acceptance of request);
2. !interface.method^ (emission of request);
3. ?interface.method\$ (acceptance of response);
4. !interface.method\$ (emission of response).

Sequence of event tokens is called a *trace*. What is meant by trace is better understandable if we switch from languages to finite automats (FSM). Behavior protocols can also be represented as finite automats. Path formed following the

specific execution of an automaton from starting state till the end state is called a *trace*, which corresponds to our definition of a trace, given at the beginning of the paragraph. Communication of the component with its environment is represented by traces of events that correspond to method calls on required and provided interfaces.

Behavior protocols are constructed using four operators:

1. sequencing operator, denoted by “;”;
2. alternation operator, denoted by “+”;
3. repetition operator, denoted by “*”;
4. and-parallel operator, denoted by “|”.

Let A and B be behavior protocols. Sequencing these two protocols $A ; B$ results in a new protocol which generates traces, where all traces generated by A are concatenated with all traces generated by B . Using alternation operator $(A + B)$ we create new protocol, which generates traces either from A or B . Repetition operator (A^*) generates new traces with any finite number of occurrence of traces of A , and an empty trace. Finally, and-parallel operator $(A | B)$ generates all interleavings of traces of protocols A and B . If we switch to graph representation of behavior protocols all interleavings of two protocols would look like full n -ary tree, where n is the number of protocols to be interleaved.

Along with or-parallel operator, there are another shortcuts defined for simplification of behavior protocols. These are given in Table 1. Here, protocol A represent a method body, which can be also an empty protocol.

| Shortcut | Abbreviated expression |
|-------------------------------|---|
| <code>!iface.method</code> | <code>!iface.method^ ; ?iface.method\$</code> |
| <code>?iface.method</code> | <code>?iface.method^ ; !iface.method\$</code> |
| <code>?iface.method{A}</code> | <code>?iface.method^ ; A ; !iface.method\$</code> |

Table 1. Shortcuts for behavior protocols.

In order to model a behavior of specific component models using behavior protocols it is necessary to distinguish component's *frame* from component's *architecture*. Component's all external required and provided interfaces form its frame. Depending on component's type, the component's architecture can be

expressed by an implementation of a component in some programming language (for primitive components) or by its internal structure, which can be defined in ADL (for composite components).

Thus, component's *frame protocol* specifies how the component can be used by its environment and defines how it reacts to the requests coming from the environment. Formally, frame protocol defines valid set of traces of atomic events on the component's frame. Component's *architecture protocol*, on the other hand, specifies the internal behavior of the component itself.

2.2.1 Example

Here we give an example of behavior protocol of a software component. Demo application for Fractal component model introduced in Chapter 2.1.1 contains `FrequentFlyerDatabase` component (Figure 3). As can be seen from the figure, this component provides only one interface (black rectangle in the figure) – `IFrequentFlyerAuth`. It requires two interfaces – `IFlyTicketDb` and `IFlyTicketAuth`.

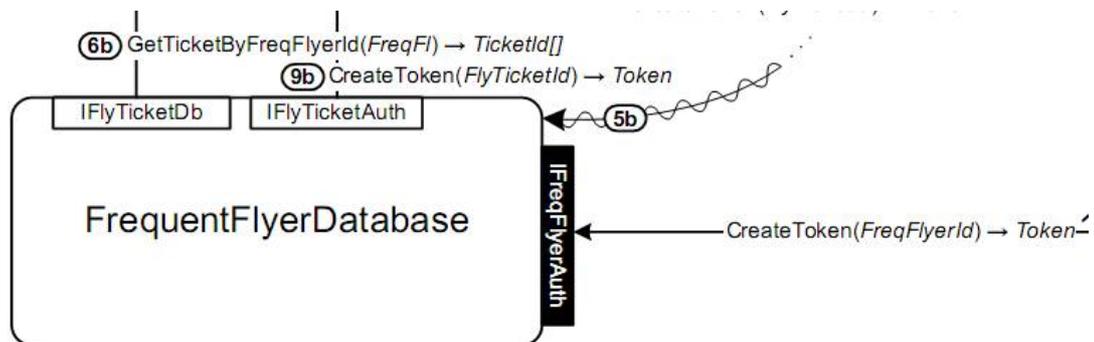


Figure 3: *FrequentFlyerDatabase* component

This component's behavior protocol is given in Listing 1. `FrequentFlyerDatabase` component is used for authentication of users of wireless Internet service using their frequent flyer ID. Figure 3 shows, that

IFrequentFlyerAuth interface has a method CreateToken(FreqFlyerId). So, FrequentFlyerDatabase component accepts CreateToken requests through its provided interface.

```
(
  ?IFrequentFlyerAuth.CreateToken {
    (
      !IFlyTicketDb.GetFlyTicketsByFrequentFlyerId;
      (!IFlyTicketAuth.CreateToken + NULL)
    )
    +
    NULL
  }
) *
```

Listing 1: Behavior protocol of FrequentFlyerDatabase component

After receiving CreateToken request FrequentFlyerDatabase component checks whether the supplied ID is valid. If supplied ID is incorrect, error is returned representing outer NULL protocol in Listing 1. If it is valid, then FrequentFlyerDatabase component emits GetTicketByFrequentFlyerId(FreqFlyerId) request through its required interface IFlyTicketDb. GetTicketByFrequentFlyerID function returns the set of FlyTicketIds which the frequent flyer bought. If this set is not empty, FrequentFlyerDatabase component emits request CreateToken(FlyTicketId) with FlyTicketId which is currently valid through its second required interface IFlyTicketAuth to receive the Token, which it will emit as a response to the request through provided interface. This situation is represented in Listing 1 by behavior protocol of CreateToken function.

If the set returned by GetTicketByFrequentFlyerId(FreqFlyerId) is empty, then an error should be returned to the caller, representing the situation that either the provided FreqFlyerId is incorrect or that frequent flyer with this ID is new and

hasn't bought any tickets yet. This situation is represented by inner `NULL` protocol in Listing 1.

2.3 Java PathFinder

Java PathFinder [6] (JPF) is a model checker developed at NASA, which is intended for verification of compiled Java programs (Java bytecode). JPF executes given program in all possible ways, i.e. it considers all branches of the state space of the program. By traversing the state space of the program, JPF searches for unhandled exceptions and deadlocks.

In order to be able to execute program in all possible ways, JPF contains its own implementation of Java Virtual Machine (JVM), which executes Java bytecode instructions generated by Java compiler. This approach is very effective in the way, that it gives the possibility to track which instruction is being executed currently, current thread number, etc. Generally, JPF can be thought of as a platform for state space exploration of Java programs + monitoring of execution of bytecode instructions.

2.3.1 State space explosion problem

State space explosion problem is one of the biggest obstacles when using methods based on exhaustive state space traversal in practice. The problem is, that the modern day real world software is so complex, that the amount of states exceeds modern computers' computational power. This issue is called *state space explosion* problem.

State space of a software system increases dramatically with introduction of a parallelism. In particular, size of the state space of a program depends exponentially on the number of threads used in the program.

In order to reduce state space of the program being examined, JPF uses such techniques as partial order reduction of the set of transitions, reduction of state storage costs and configurable search strategies. Among these techniques partial order reduction is one of the most important mechanisms used in JPF.

This technique considers only context switches caused by instructions that can have effects across thread boundaries, e.g. modifications of shared fields in the object by `PUTFIELD` instructions. JPF uses reachability information from the garbage collector in order to achieve partial order reduction.

2.3.2 Search- and VMListeners

Java PathFinder can be extended with `VMListeners` (Virtual Machine Listeners), which can be used, e.g., to monitor executions of specific bytecode instructions and `SearchListeners`, which provide necessary methods for monitoring search process and state space exploration.

Listener instances register themselves with respective object, i.e. `Search` or `VM` objects for `SearchListeners` and `VMListeners`, respectively. Registered listeners get notified when the monitored object (subject) performs certain actions.

The definition of `VMListener` interface can be found in Listing 2. As can be seen from the figure, `VMListener` interface provides many methods to monitor instructions' executions, thread statuses, synchronization and dynamic allocation of objects.

Specifically, `executeInstruction` and `instructionExecuted` methods are called before and after execution of a bytecode instruction of a component, respectively. Methods `threadStarted`, `threadWaiting`, `threadNotified`, `threadInterrupted`, `threadTerminated` and `threadScheduled` are called when a thread's state has changed accordingly. Note that these methods are called only on the threads explicitly defined in a component's code. The same assumption is correct for all methods related to objects.

```

public interface VMLListener extends JPFLListener {

    /* JVM is about to execute the next instruction */
    void executeInstruction (JVM vm);

    /* VM has executed next instruction
       (can be used to analyze branches, monitor PUTFIELD / GETFIELD and
       INVOKExx / RETURN instructions) */
    void instructionExecuted (JVM vm);

    /* new Thread entered run() method */
    void threadStarted (JVM vm);

    /* thread is waiting for signal */
    void threadWaiting (JVM vm);

    /* thread got notified */
    void threadNotified (JVM vm);

    /* thread got interrupted */
    void threadInterrupted (JVM vm);

    /* Thread exited run() method */
    void threadTerminated (JVM vm);

    /* new thread was scheduled by VM */
    void threadScheduled (JVM vm); // might go into the choice generator
                                   // notifications

    /* new class was loaded */
    void classLoaded (JVM vm);

    /* new object was created */
    void objectCreated (JVM vm);

    /* object was garbage collected (after potential finalization) */
    void objectReleased (JVM vm);

    /* notify if an object lock was taken (this includes automatic
       surrender during a wait()) */
    void objectLocked (JVM vm);

    /* notify if an object lock was released (this includes automatic
       reacquisition after a notify()) */
    void objectUnlocked (JVM vm);

    /* notify if a wait() is executed */
    void objectWait (JVM vm);

    /* notify if an object notifies a single waiter */
    void objectNotify (JVM vm);

    /* notify if an object notifies all waiters */
    void objectNotifyAll (JVM vm);

    /* garbage collection cycle started */
    void gcBegin (JVM vm);

    /* garbage collection cycle finished */
    void gcEnd (JVM vm);

    /* exception was thrown */
    void exceptionThrown (JVM vm);

}

```

Listing 2: VMLListener interface definition

Unlike VMLListeners, SearchListeners are notified by Search object, which

performs state space exploration process. SearchListener interface definition is given in Listing 3. Methods `searchStarted`, `stateAdvanced`, `stateBacktracked`, `searchFinished`, `stateProcessed` are standard ones for monitoring state space exploration process. Additional methods are added to SearchListener interface to extend monitoring of search process. Method `searchConstraintHit` is called when search constraints defined were encountered, e.g. maximal depth of search tree is reached. When predefined properties are violated, such as `NotDeadlockedProperty` and `NoUncaughtExceptionsProperty`, `propertyViolated` method is called.

```
public interface SearchListener extends JPFLListener {

    /* got the next state */
    void stateAdvanced (Search search);

    /* state is fully explored */
    void stateProcessed (Search search);

    /* state was backtracked one step */
    void stateBacktracked (Search search);

    /* a previously generated state was restored
       (can be on a completely different path) */
    void stateRestored (Search search);

    /* JPF encountered a property violation */
    void propertyViolated (Search search);

    /* we get this after we enter the search loop,
       but BEFORE the first forward */
    void searchStarted (Search search);

    /* there was some constraint hit in the search, we back out
       could have been turned into a property, but usually is an
       attribute of the search, not the application */
    void searchConstraintHit (Search search);

    /* we're done, either with or without a preceding error */
    void searchFinished (Search search);

}
```

Listing 3: SearchListener interface definition

In JPF, state space exploration is performed in the depth-first search manner by default. Notification model for depth-first search is given in Figure 4. It is possible to illustrate on this model the relationship between VMLListeners and SearchListeners: VMLListener methods are called for each transition of given automaton.

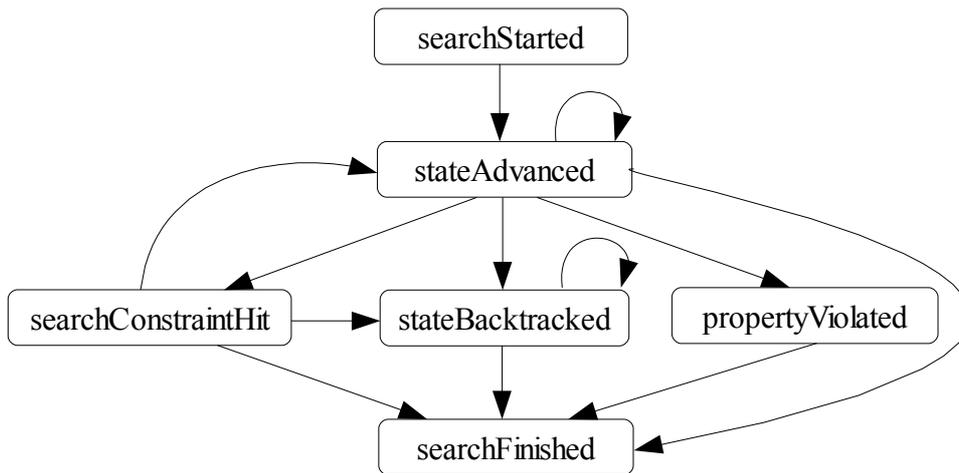


Figure 4: Depth-first search notification model

2.3.3 Choice generators

To systematically explore the state space JPF uses the mechanism called *Choice Generators*. Choice Generators are classes which enumerate all choices from a type specific interval. Namely, these classes start the new transition, thus creating and structuring JPF state space.

Before going any further, we will define some key terms to be used overall this thesis. These definitions are taken from JPF documentation.

State is a snapshot of the current execution status of the application (mostly thread and heap states), plus the execution history (path) that lead to this state. Every state has a unique ID number.

Transition is the sequence of instructions that leads from one state to the next. There is no context switch within a transition, it's all in the same thread. There can be multiple transitions leading out of one state (but not necessarily to a new state).

Choice is what starts a new transition. This can be a different thread (i.e. scheduling choice), or different "random" data value.

In other words, possible existence of choices is what terminates the last transition, and selection of a choice value precludes the next transition. Terminating a transition corresponds to creating a new choice generator, and letting the SystemState know

about it. And selection of a choice value means to query the next choice value from this choice generator (either internally within the JVM, or in an instruction or native method). Figure 5 illustrates the relationships between Transitions, States and Choices.

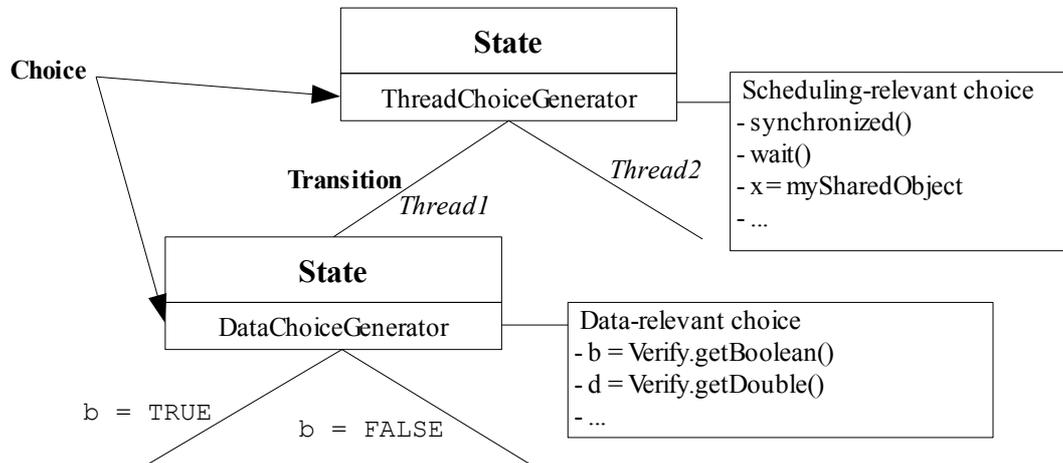


Figure 5: Relations between states, transitions and choices

Every state in JPF state space has exactly one choice generator assigned, regardless of whether it is a scheduling-relevant or nondeterministic data-relevant choice generator.

Chapter 3. Construction of abstraction

As indicated in Section 1.1, the goal of this thesis is to create a tool for construction of a model (abstraction) of component's implementation and a model of its usage by the rest of a system (other components in the system), i.e. model of interaction between C (component) and E (environment). Since, in this thesis, we chose behavior protocols as the formalism for the constructed abstraction, the terms “C-E interaction model” and “behavior protocols” are used interchangeably in the rest of the text. This chapter closely describes an algorithm for a component abstraction and analyses the choices made in the design of the algorithm.

3.1 Algorithm outline

In order to achieve the goal of this thesis a modeling formalism had to be selected. Behavior protocols were chosen as the modeling formalism, in which the abstraction of components is to be expressed. Justification of the choice is given in Chapter 1.

Implementation of the algorithm is based on Java PathFinder model checker, which is used for verification of Java bytecode programs.

Events (method invocations and returns) are acquired by a JPF listener that is a part of the tool. `VMLListener` method is used to monitor `INVOKE` and `RETURN` instructions, i.e. to detect all relevant atomic events. And `SearchListener` methods are used here for processing data obtained from a current state, i.e. for derivation of behavior protocol and for estimating running time of the algorithm.

We assume that the system consists of two parts – the target component C and its real environment E (the rest of the system).

The input of the algorithm is (i) Java code of the whole system (Java program with `main` method), (ii) list of required and provided interfaces that form the boundary between C and E, and, (iii) mapping of methods of interfaces given in (ii) to atomic events. And the output is the model of interaction between C and E in the formalism of behavior protocols.

Java PathFinder (JPF) is used to perform the state space traversal of the Java

program (composed of Java code of C and E). The protocol is constructed on-the-fly during state space traversal of the Java program – JPF listener mechanism is used for this purpose. Listener mechanism provided by JPF is used to gather traces of important events (method invocations and returns). All the logic of the construction algorithm is implemented in the listeners.

Names of methods to be monitored in the listener are derived from the interfaces – all (public) methods of the interfaces (on C-E boundary) listed in an input file are monitored.

Next, in this chapter, we will describe the simplified algorithm which generates model of C-E interaction for single-threaded applications (Section 3.2). Then we generalize the algorithm to multi-threaded applications satisfying conditions given in Section 3.3. Section 3.4 discusses processing of application's state space with respect to BP operators.

3.2 Simplification to single-threaded applications

For the sake of simplicity we first start with the design of an algorithm for generation of a behavior protocol of a single-threaded application. Main idea is, that after a state has been fully processed, meaning all choices were explored for the state, it contains a complete behavior for the subtree of the state space for which that state is a root. Maintaining this invariant leads to a complete behavior model of an application as soon as a root state has been fully explored. In order to achieve this, the following has to be done:

- during processing of an unexplored transition, a sequence of method invocations and returns from methods on interfaces forming monitored component's frame (C-E boundary) is recorded via analysis of all executed bytecode instructions and attached to the transition, and,
- during backtracking, the C-E protocol is derived from the sequences of atomic events associated with transitions and from the structure of the state space.

A high-level representation of the algorithm is given in Listing 4 in pseudo-code.

The `transitionEvents` data structure is a sequence (linked list) that is used to store atomic events on the C-E boundary that are performed during a transition in the state space. The sequence is filled in the `instructionExecuted` handler procedure (lines 5-14), which is called by JPF for each bytecode instruction executed during traversal of the state space of a Java program.

When the still unexplored transition is terminated, JPF calls the `stateAdvanced` handler procedure. In this procedure, a protocol of the form `e1 ; e2 ; ... ; eN` is created from the events `e1, e2, ..., eN` that are stored in the `transitionEvents` sequence, and then the protocol is associated with the transition (line 17).

```
1 INIT
2   transitionEvents = {}
3 end
4
5 instructionExecuted(insn)
6   if isMethodInvocation(insn)
7     if isMethodOfComponent(insn.methodName)
8       transitionEvents += { ?insn.methodName^ }
9     else transitionEvents += { !insn.methodName^ }
10  if isReturnFromMethod(insn)
11    if isMethodOfComponent(insn.methodName)
12      transitionEvents += { !insn.methodName$ }
13    else transitionEvents += { ?insn.methodName$ }
14  end
15
16 stateAdvanced()
17   transition.protocol = createSequence(transitionEvents)
18   transitionEvents = {}
19   curState.protocol = ""
20 end
21
22 stateBacktracked()
23   curState.protocol =
24   mergeProtocols(curState, prevState, transition)
25 end
```

Listing 4: High-level representation of the algorithm for construction of C-E protocol

The resulting C-E protocol is created from the sequences of events (protocols of the form $(e_1 ; \dots ; e_N)$) in the `stateBacktracked` handler procedure, which is called by JPF when it backtracks over a transition. Specifically, the `mergeProtocols` procedure is called (lines 23-24), which merges the protocols associated with the transition and with the states `curState` and `prevState`, adding some protocol operators during the merge, and then attaches the protocol to `curState` — here, `curState` denotes the start state of the transition (i.e. the current state after backtracking) and `prevState` denotes the end state of the transition.

The logic performed in the `mergeProtocols` procedure is depicted on Listing 5 (again in the form of pseudo-code). First, the protocols associated with the given transition (value of the parameter `transition`) and previous state (parameter `prevState`) are merged using the sequence operator “;”, and this intermediate protocol is stored into the `tempProtocol` variable – this operation expresses the fact that method invocations and returns from that methods (on the C-E boundary) performed in the transition are executed before each trace of events encoded in the protocol associated with the previous state.

The form of the resulting protocol depends on the structure of the state space:

- if the current state (value of the parameter `curState`) has only one successor, then the protocol stored in the `tempProtocol` variable represents the complete subtree of the current state in the state space (i.e. all traces of events performed in the subtree);
- if the current state has more than one successor (e.g. as a consequence of nondeterministic data choice), then the protocol stored in `tempProtocol` corresponds only to one state space branch that starts in the current state. Sub-protocols p_1, p_2, \dots, p_N corresponding to all branches starting in the current state are composed using the alternative operator “+” — after JPF backtracks over all transitions leading from the current state, the protocol associated with that state has the form $p_1 + p_2 + \dots + p_N$.

```

1 string mergeProtocols(curState, prevState, transition)
2   string tempProtocol =
3     "(" +
4     transition.protocol +
5     " ; " +
6     "(" + prevState.protocol + ")" +
7     ")"
8   if (curState.successors.count == 1) return tempProtocol
9   else /* curState.successors.count > 1 */
10    if curState.protocol == "" return tempProtocol
11    else return curState.protocol + " + " + tempProtocol
12 end

```

Listing 5: Implementation of the mergeProtocols method

Note that protocols involving only the operators “;” and “+” are created in the mergeProtocols method. Loops in a protocol, i.e. points where the repetition operator “*” should be used, cannot be derived from the structure of the state space of a Java program for two reasons: (i) loop in the state space corresponds to a back edge to an already visited state (i.e. not a loop in the program code), and (ii) loop in the program is represented by a sequence of transitions in the state space — in particular, it is not possible to use an algorithm based on detection of back edges in the state space, since the back edges correspond to transitions to already visited states, not to jumps to the beginning of loops in program code. Therefore, only sequences — protocols of the form $p_1 ; p_2 ; \dots ; p_N$ — are derived during the state space traversal. The repetition operator is applied in the post-processing phase to transform sequences of the same sub-protocols ($p ; p ; \dots ; p$) to protocols of the form p^* .

3.3 Generalization to multi-threaded applications

An extension of the algorithm described above to multi-threaded Java programs (Listing 6) exploits the fact that, in JPF, thread scheduling is done only at transaction boundaries; specifically, for each transition, all bytecode instructions in the transition

are executed by the same thread. The C-E protocol for a multi-threaded Java program is constructed in two steps. First, a sub-protocol $prot_i$ is constructed for each thread T_i separately — i.e. only the transitions executed by T_i are considered — using the algorithm described in previous section. Then, all the sub-protocols $prot_1, \dots, prot_N$ for all threads T_1, \dots, T_N are composed using the parallel operator $|$, yielding a C-E protocol of the form $prot_1 | prot_2 | \dots | prot_N$ (lines 50-54 in Listing 6). However, this way, a correct and precise behavior protocol can be extracted only for those Java programs that satisfy the following restriction: (i) all threads that call some methods on the C-E boundary during their lifetime have to be started (created) before the first method call on the C-E boundary and terminated after last method call on C-E boundary and (ii) no thread can influence the control-flow of any other thread. In particular, the proposed extension does not work for cases, where some threads are created dynamically in the component C and/or when the choice between branches of an if-else statement in one thread depends on the value of a shared variable set in another thread (see Chapter 5. Evaluation).

The main issue related to state space traversal of complex programs with many parallel threads is that state explosion may occur. The key idea behind our approach to addressing the state explosion problem is that only a part of the state space is traversed during extraction of a behavior protocol for given Java program. Assuming that all threads are running before the first method call on the C-E boundary is performed, it is sufficient to traverse only one interleaving of the threads in order to identify the sub-protocols $prot_1, \dots, prot_N$ corresponding to individual threads T_1, \dots, T_N .

Technically, this optimization is implemented by exploring only one choice for each choice generator (CG) related to thread scheduling; other choices in such a CG are ignored (lines 40-41 in Listing 6).

Additionally, after a branching occurs due to a nondeterministic data choice, only that events which are generated by the thread in which this choice occurred are considered (lines 4, 9, 25, 29, 37-39). Upon return to the initial trace all threads are considered again (lines 3, 44-48). Although, this optimization technique is correct only for the cases where all threads created by the component are always available, it significantly reduces the size of resulting behavior protocol.

The restriction to one choice for each thread scheduling-related CG is correct with respect to identification of actions performed by individual threads, since the choices that were not selected will be enabled at the next scheduling point (i.e. when the next thread scheduling-related choice is to be made). Note also that choice generators related to nondeterministic data choice are not altered at all.

3.4 Notes on operators' processing

As mentioned above, behavior protocols have the following 4 operators: sequence (“;”), alternation (“+”), repetition (“*”) and and-parallelism (“|”). Implementation of each of these operators is discussed in details in the following subsections. Detection of operators differs significantly depending on whether the application being processed is single-threaded or multi-threaded. With single-threaded software components behavior protocol inferring will turn, basically, into an algorithm of conversion of finite automaton to regular expression [2].

With parallel, i.e. multi-threaded software components, however, it is much more complicated. In this case the component can be thought of as an interleaving of several finite state machines and exactly the problem of extracting of individual finite automata from this complex finite state machine is of the main concern.

3.4.1 Sequence operator

The sequence operator is the most trivial operator of behavior protocol syntax. All the method calls performed in one transition are delimited by sequence operator as all these events are generated by the same thread (see Figure 6). Furthermore, two subsequent states are also delimited by that operator if the transition was triggered by a nondeterministic data choice generator, i.e. if the next state is executed by the same thread.

```

1 INIT
2   transitionEvents[] = {}
3   monitoredThread = 0    //Monitor all threads
4   curThrNumber = 1
5   componentProtocol = ""
6 end
7
8 instructionExecuted(insn)
9   if isMonitoredThread(insn.threadNumber)
10    if isMethodInvocation(insn)
11      if isMethodOfComponent(insn.methodName)
12        transitionEvents[insn.threadNumber] += {?insn.methodName^}
13      else transitionEvents[insn.threadNumber] += {!insn.methodName^}
14    if isReturnFromMethod(insn.methodName)
15      if isMethodOfComponent(insn.methodName)
16        transitionEvents[insn.threadNumber] += {!insn.methodName$}
17      else transitionEvents[insn.threadNumber] += {?insn.methodName$}
18 end
19
20 stateAdvanced()
21   transition[curThrNumber].protocol =
22     createSequence(transitionEvents[curThrNumber])
23   transitionEvents[curThrNumber] = {}
24   curState[curThrNumber].protocol = ""
25   curThrNumber = getCurrentThreadNumber()
26 end
27
28 stateBacktracked()
29   curThrNumber = getCurrentThreadNumber()
30   curState[curThrNumber].protocol =
31     mergeProtocols(curState[curThrNumber], prevState[curThrNumber],
32                   transition[curThrNumber])
33 end
34
35 choiceGeneratorAdvanced(VM)
36   cg = VM.getChoiceGenerator()
37   if isDataChoiceGenerator(cg)
38     if (cg.ProcessedNumberOfChoices > 1) && (monitoredThread == -1)
39       monitoredThread = VM.getLastStateThreadNumber()
40     if isThreadChoiceGenerator(cg)
41       cg.setDone()
42 end
43
44 choiceGeneratorProcessed(VM)
45   cg = VM.getChoiceGenerator()
46   if isDataChoiceGenerator(cg) && VM.getThreadNumber == monitoredThread
47     monitoredThread = 0 //monitor all threads
48 end
49
50 searchFinished()
51   for curThrNumber = 1 to numberOfThreads-1
52     componentProtocol += curState[curThrNumber] + "|"
53   componentProtocol += curState[numberOfThreads]
54 end

```

Listing 6: High-level representation of the algorithm for construction of C-E protocol for multi-threaded applications

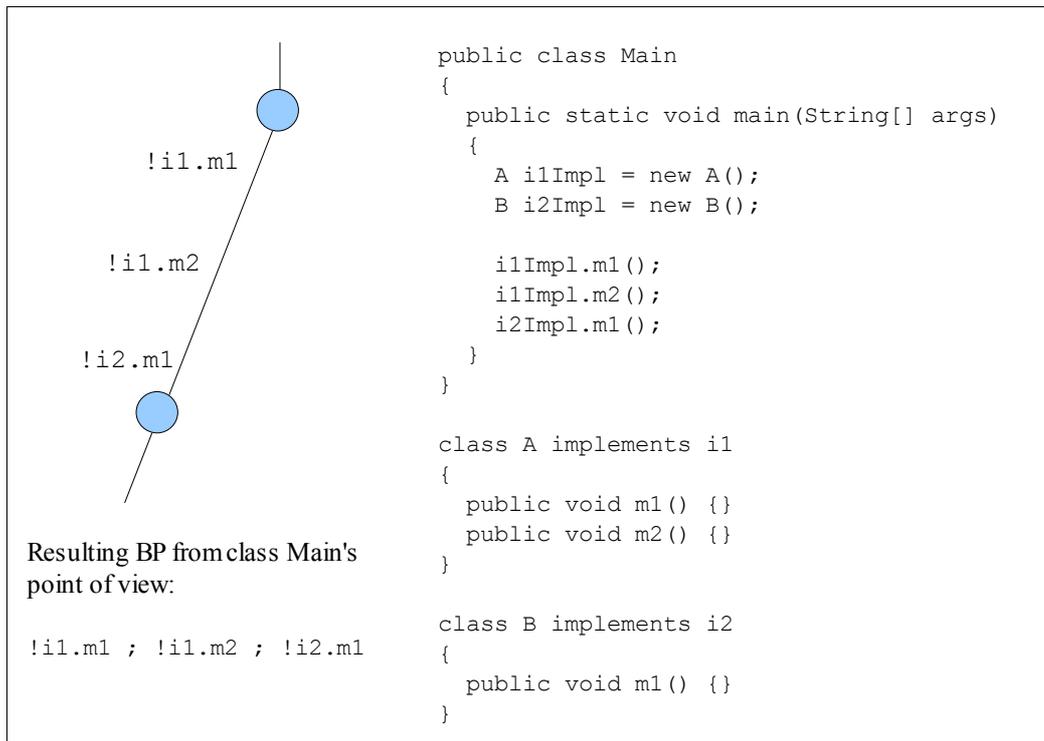


Figure 6: Sequence operator

3.4.2 Alternative operator

Alternative operator represents branches in the execution state space of a Java program. Branches are represented and managed by choice generators (see 2.3.3 Choice generators).

When a backtrack occurred from the state where branching began, protocols of the branches are kept in predecessor's data structure. This structure contains complete behavior protocols of the branch. Thus, predecessors' protocols are appended parenthesized into the protocol of branching-root state using alternative operator “+”.

Figure 7 illustrates the relationship between JPF state space and the source code for alternative operator and shows resulting BP from Main class's point of view. The branching in the state space is triggered by nondeterministic data choice at line 7. For the case when `b` is equal to `TRUE` lines 9-10 are executed. Lines 12-13 are processed

when `b` equals to `FALSE`. The figure also demonstrates the drawback that both branches can have a common prefix, in this particular case `!i1.m1`, which eventually can make resulting protocol unnecessarily large and hard to read.

3.4.3 Repetition operator

Given the specific shape of a program state space due to DFS traversal, it could be possible to implement following algorithm for detection of cycles:

- 1) for each event `U` in protocol, the numbers of states, which JPF visited while traversing from the event preceding `U` to the event `U` are stored;
- 2) upon encountering a back transition (after event `U1`) to some state `S`, event `U2`, which comes after state `S` is found, and the protocol between `U1` and `U2` is put into `(...)*`.

This algorithm would be correct, because in Java code either whole cycle is inside `if`-branch or `else`-branch, or whole `if-else` operator is inside of a cycle. Thus, it cannot happen that the state space will split inside of a cycle and will not reconnect with other branches before the end of the cycle.

The problem is, that during state space traversal by JPF JVM, several (or all) iterations of a loop may be represented by one transition in the state space. Hence, repetition operator cannot be derived in general. Repeated code is fully contained in one transition. So, it cannot be detected, that JPF returned to the visited state.

There are two possible solutions to the problem:

1. POR modification – event on interface will break the transition, i.e. `ThreadInfo.breakTransition()` function will be called;
2. Identification of cycles after traversal, in post-processing, by identifying the sequences of equal sub-protocols and translating them into repetitions (i.e. `A ; A ; A → A*`).

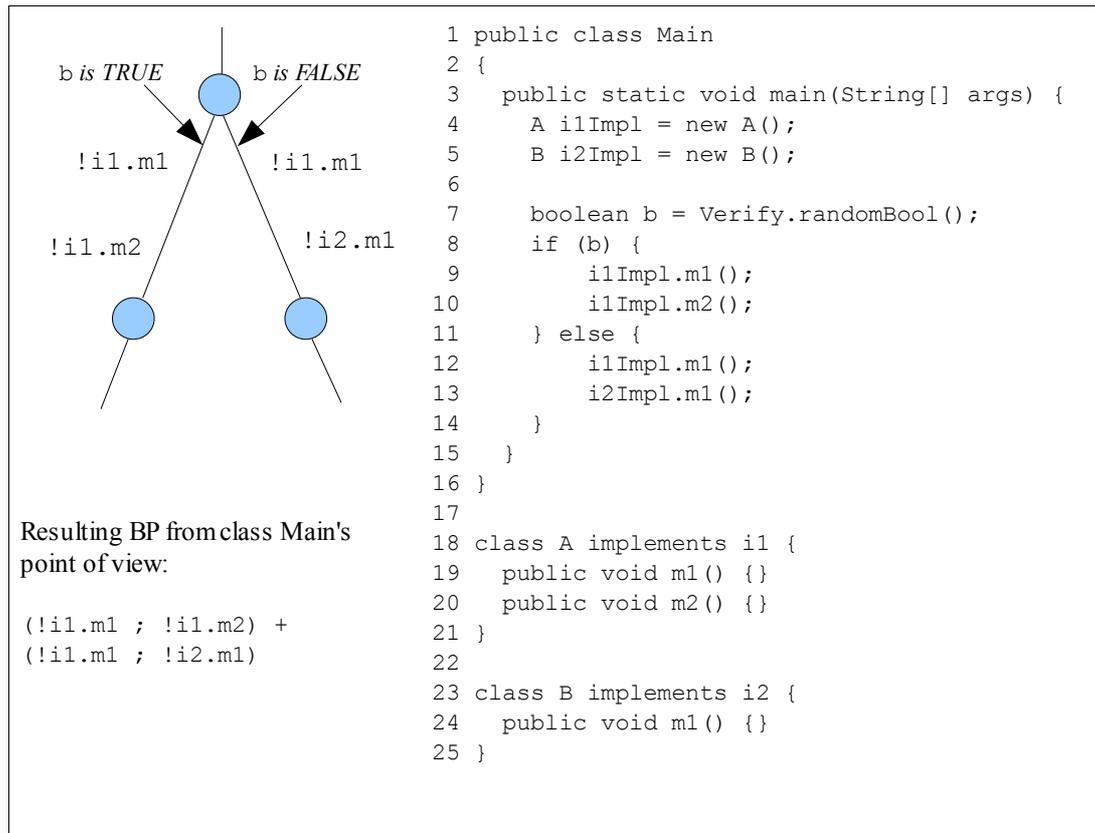


Figure 7: Alternative operator

Algorithm based on back transitions in the state space cannot be used as JPF executes whole cycles in one transition. Solution based on POR modification also cannot be used, because POR is switched off in our implementation (see Chapter 5. Evaluation). Thus, post-processing solution has been chosen.

3.4.4 And-parallel operator

AND-parallel operator was at the same time most interesting and most difficult one to implement. Implementation of this operator resulted in serious changes of data structures used in the program and algorithm modifications itself.

In order to implement the detection of AND-parallel operator, it is important to understand that JPF state space is not a state space of some thread in the component. It

Two solutions exist for the detection of parallelism:

- 1) VM object passed to `VMListener` contains an identification number of a thread (`vm.getThreadNumber()` method). Thus it is possible to find out in which thread given code is being executed and to interconnect protocols with different thread numbers with AND-parallel operator.

The advantage of this solution is that there is no need to explore all thread interleavings – it is enough to explore only one interleaving using `ThreadInfo.index` to detect a parallelism. In order to achieve this, each thread-scheduling choice generator (TSCG) has to be instructed that other choices should not be considered except the one which have already been chosen. Nondeterministic data choice generators, on the other hand, must not be affected.

The above-mentioned technique of traversing of only one interleaving is correct if all the threads have been started before calling any of monitored interfaces' methods, i.e. threads cannot be dynamically created in the component. Additionally, in order to correctly generate a C-E protocol, it is required that each thread will get to run in the only interleaving being explored.

Thus, at the time of first calling of interface method the number of threads is given, and is equal to the number of operands of AND-parallel operator. Resulting BP will have a following form: $(...) \mid (...) \mid \dots \mid (...)$, i.e. AND-parallel operator will be only on the top-level.

Better precision of resulting BP can be achieved only by traversing all interleavings of the JPF state space. But as the state space is prone to state explosion when the application is multi-threaded, exploring all interleavings is not scalable and efficient solution.

- 2) For each two events A and B in the given graph (in our case JPF state space), it is possible to remember whether they were

executed in the order A, B or B, A or both. This will allow to detect if these events are executed in parallel or not. But this alternative solution will not work if all interleavings except one are excluded from traversing.

We have chosen the first solution. The argument in favor of the first solution is that it is considerably faster than the second one, since only one interleaving is explored. Traversing whole state space of the application is inefficient, because of exponential growth of the number of state spaces when the application is multi-threaded.

Chapter 4. Implementation details

This chapter describes the implementation of the algorithm for generation of model of C-E interaction for specific class of multi-threaded applications defined in previous chapter. As was mentioned, the implemented tool *BytecodeAbstractor* makes use of JPF model checker for verification of Java bytecode programs, extending it with VM- and SearchListeners.

Important requirement for the implementation was: as few changes as possible to the JPF. In fact, the implementation does not alter the tool at all. With this, however, several problems arise, such as the impossibility to derive a repetition operator and Partial Order Reduction (POR)-triggered problems.

BytecodeAbstractor constructs behavior protocol by traversing the state space of given Java bytecode program. The output of the program is a file containing the derived behavior protocol and the execution time. As the implementation has to be able to derive C-E interaction model also for multi-threaded applications the state space of the target application is stored, internally, in hash map which hashes the state spaces of each threads using thread ids:

```
HashMap<Integer, Stack<StateData>>.
```

It can be seen from the representation of the target application state space that each thread's state space in target application is represented by `Stack` data structure containing the stack of states of current execution trace.

State of the target application's state space is defined as:

```
private class StateData {
    public int StateNumber = 0;
    public int ThreadNumber = 0;
    public String Protocol;
    public Queue<StateData> Successors;
}
```

The state's id is represented in `StateNumber` field. The `ThreadNumber` field contains the thread which is/was running in this state (Note: only one thread can run at

the same time in one state). Behavior protocol generated for this state is stored in `Protocol` field. The `Successors` queue in the `StateData` class contains all immediate successors of the state once this state has been fully processed.

After transition from one state to another, `stateAdvanced()` method is called on the `BehaviorProtocolAbstractor` class object, which implements the `SearchListener` interface for JPF. In this function, part of the behavior protocol created during the execution of the code in previous state is pushed onto trace stack of the thread which run the previous state. When the state is backtracked, i.e. when, the `stateBacktracked()` method is called on `BehaviorProtocolAbstractor` object, the protocol of the previous state is integrated with the protocol of the state which is on top of the trace stack of the thread which run the previous state. Other details of behavior protocol derivation algorithm, such as state space branching processing, are described in previous chapter.

4.1 Program input

Program input data are:

- main class, which simulates an environment of the component (it is needed for JPF, as it works on complete Java programs and not on software components);
- list of interfaces (Interface classes in Java language), which are needed to be monitored to create a behavior protocol, and,
- list of methods of the interfaces given above to be tracked and mappings of the couple <interface, method> to events.

All input data can be specified in the program GUI. See Appendix C. User manual for the details on how to use `BytecodeAbstractor` GUI.

Input is defined in XML format, where all classes which implement interfaces to be tracked are defined, including these interfaces' methods. Every pair *Class.Method* is mapped to an event. This mapping is also given in the input. An example of an input is given in Listing 7. It is possible to generate the input file in the `BytecodeAbstractor` GUI.

Given XML is parsed and two maps of provided and required interfaces are created from this input. Maps have the following structure:

```
HashMap<String, HashMap<String, String>>.
```

The key of the map is the name of the class (including package name) which has to be tracked. Value of the map is a mapping of methods of the given interface to events.

4.2 Detection of events

In the previous chapters it was mentioned that method calls are basic blocks of behavior protocols. In order to be able to infer protocol of a method call itself we have to detect two things:

- invocation of method and,
- return from that method.

The detection of events is performed in the `instructionExecuted()` method of `BytecodeAbstractor` class. Besides the detection of method invocations and returns from these method invocations, it is checked if the instruction being executed is in the thread which has to be tracked in this branch of `DataChoiceGenerator`.

Java bytecode has four instructions for method invocations:

1. *invokeinterface*, which invokes a method implemented by an interface, searching the methods implemented by the particular runtime object to find the appropriate method;
2. *invokespecial*, which invokes an instance method requiring special handling, whether an instance initialization method, a *private* method, or a superclass method;
3. *invokestatic*, which invokes a *static* method in a named class;
4. *invokevirtual*, which invokes an instance method of an object, dispatching on the (virtual) type of the object. This is the normal method dispatch in the Java programming language.

```

<?xml version="1.0"?>
<Interfaces>

<Provided>
  <Interface>
    <Name>test1.FrequentFlyerDatabaseImpl</Name>
    <Method>
      <Name>CreateToken</Name>
      <Event>IFreqFlyerAuth.CreateToken</Event>
    </Method>
  </Interface>
</Provided>

<Required>
  <Interface>
    <Name>test1.IFlyTicketDb</Name>
    <Method>
      <Name>GetFlyTicketsByFrequentFlyerId</Name>
      <Event>IFlyTicketDb.GetFlyTicketsByFrequentFlyerId</Event>
    </Method>
    <Method>
      <Name>GetFlyTicketValidity</Name>
      <Event>IFlyTicketDb.GetFlyTicketValidity</Event>
    </Method>
  </Interface>
  <Interface>
    <Name>test1.IFlyTicketAuth</Name>
    <Method>
      <Name>CreateToken</Name>
      <Event>IFlyTicketAuth.CreateToken</Event>
    </Method>
  </Interface>
</Required>

</Interfaces>

```

Listing 7: Sample input in XML format

JPF has the parent class `InvokeInstruction`, from which all above-mentioned instructions are inherited. As in case of method invocation of class implementing an interface, it is not possible to infer the interface name, we need to have those interface names and methods given.

The second instruction we are interested in is the RETURN instruction. Java bytecode has different return instructions based on return type, e.g. *ireturn* (for `int`, `byte`, `short`, `char`, `boolean`, `byte` return types), *lreturn* (for type `long`), etc. This means that we should detect all of these *return* instructions. Fortunately, Java PathFinder has a superclass for all types of return instructions,

`ReturnInstruction`. So, the calls for these instructions can be detected using `instanceof` operator by checking if the given instruction is an instance of `ReturnInstruction` class.

4.1.1 Processing method invocations

When the invocation of method is detected both provided and required interfaces' maps given at the beginning are checked if they contain the name of the class as key. If they do, then the value assigned to the key is extracted from map. Then it is checked, if the method being invoked is contained in the internal map which maps methods to events. If it is not, then this method invocation is ignored. Further logic is the following:

- if the invoked method is in the map of provided interfaces, then the event protocol is “? event_name ^”;
- if the invoked method is in the map of required interface, then the event protocol is “! event_name ^”.

4.1.2 Processing RETURN instruction

Return instructions are processed similarly to method invocations' processing. Here too the name of the class's method, which has been processed is searched in both maps of provided and required interfaces as it is done in processing of method invocation.

Detection of the returns from interface methods are complicated by the fact that JPF JVM returns the class name implementing this method and not the interface name. Moreover, in JPF there is no way of retrieving of the interface names which the given class implements. This is the reason why the input file for `BytecodeAbstractor` has to explicitly include all the classes which implement a given interface. With this, however, default naming convention of the event “`Classname.Methodname`” becomes incorrect, since the class name in this case will be the name of the implementing class

and not the interface. This is easily solved by introducing event name into input file, so the class name can be mapped to the interface name which this class implements.

The logic for the processing of return instructions is as follows:

- if the method which have been processed is in the map of provided interfaces, then the event protocol is “! event_name \$”;
- if the method which have been processed is in the map of required interfaces, then the event protocol is “? event_name \$”.

Chapter 5. Evaluation

Our approach of automated extraction of component-environment interaction model based on the state space traversal combines advantages of the existing techniques based on both static and runtime analysis, and addresses some of their drawbacks. Particularly, it

- considers all execution paths of the system and thus it is comparable to static analysis-based techniques,
- is comparable to runtime analysis, since no abstraction of Java code is performed before or during the state space traversal, and
- is scalable and efficient, since only selected paths in the state space are actually traversed – state explosion does not occur.

The optimization mentioned in the last point is based on exploring only one interleaving of threads. Table 2 shows how this technique affects the explored state space of a program. It displays the results of application of BytecodeAbstractor tool on three sample programs. First one, Frequent Flyer, is relatively complex single-threaded program which is modeled in one JPF state. Firewall test program is a simple multi-threaded application without nondeterministic data choices. Finally, Transient Database is a relatively complex multi-threaded application which also includes nondeterministic data choices. To estimate the optimization achieved by suggested technique, let N be the total amount of states (terminated with data CG) of an application. Further, let m be the amount of threads in the application. If r_i represents the amount of nondeterministic data choice generators for the i -th thread in a trace then the amount of explored application states (ended with data CG) M is

$$M = \frac{N}{\frac{(r_1 + r_2 + \dots + r_m)!}{r_1! \cdot r_2! \cdot \dots \cdot r_m!}} .$$

However, the approach described in this thesis has its own limitations too. The main limitation is the restriction to a subset of Java programs. Specifically, in these

programs (i) no thread can influence the control-flow of any other thread, and, (ii) all threads have to already be started before the first method call on either provided or required interfaces being tracked. To explain the first restriction let us consider the program in Listing 8. In this example two threads are created in the `Main` class before calling any methods on required interfaces (lines 5-9). Additionally, class `Main` contains a static variable `param` (line 2) which is shared between the two threads created by this class. Specifically, class `A` changes this field randomly depending on its boolean field's value and immediately calls a method on required interface (lines 17-26). And class `B` calls different methods on the same required interface depending on the shared variable `param`.

| Test | Explored amount of states | Total amount of states | Execution time (ms) |
|---|----------------------------------|-------------------------------|----------------------------|
| Frequent Flyer | 1 | 1 | 281 |
| Firewall | 14 | 105 | 250 |
| Transient Database (simplified: 2 cycles) | ~190 | 120953 | 750 |
| Transient Database (full: 5 cycles) | 100816 | > 6500000 | 158391 |

Table 2: Optimization impact on the amount of explored states

When JPF traverses state space of that program, modification of shared variable will cause the break of the transition and scheduling-relevant choice generator will be set. As our approach for fighting state space explosion problem is based on traversing only one interleaving, another interleaving will not be considered. Thus either if-part (lines 34-36) or else-part (lines 37-39) of the second thread (implemented by class `B`) will not be explored.

Nevertheless, Java programs satisfying restrictions mentioned above still form an interesting group, e.g. server-side programs, where all threads are created and put into thread pool before any of them calls methods on business components.

Another limitation of our approach is that, in general, loop detection does not work, because of the way JPF constructs target program's state space. Loops in application code are often executed in one transition.

```

1  public class Main {
2      static int param = 0;
3
4      public static void main(String args[]) {
5          Thread th1 = new A();
6          Thread th2 = new B();
7
8          th1.start();
9          th2.start();
10     }
11 }
12
13 class A extends Thread {
14     requiredItf rItf = new requiredItf();
15
16     public void run() {
17         boolean b = Verify.randomBool();
18
19         if (b) {
20             Main.param = 0;
21             rItf.methodA();
22         }
23         else {
24             Main.param = 1;
25             rItf.methodB();
26         }
27     }
28 }
29
30 class B extends Thread {
31     requiredItf rItf = new requiredItf();
32
33     public void run() {
34         if (Main.param == 1) {
35             rItf.methodC();
36         }
37         else { //Main.param == 0
38             rItf.methodD();
39         }
40     }
41 }

```

Listing 8: Shared field problem.

As was mentioned before, JPF partial order reduction (POR) technique has to be

switched off for BytecodeAbstractor tool to function properly. The reason for this is that when POR is enabled JPF VM skips transitions to some steps completely as it matches those states to previously visited ones, even if those transitions are different than the ones already visited. Moreover, as JPF uses on-the-fly POR technique the result of each execution of BytecodeAbstractor yields different BP when POR is turned on. The solution for that problem can be the modification of POR in JPF to match only the end states of the transitions.

Additionally, behavior protocols generated by proposed approach may be large and complicated, thus harder to read and comprehend by users. In particular, branches of an alternative operator “+” in generated protocol have a common prefix, if there are some method calls on tracked interfaces between the nondeterministic choice associated with “+” and the if-else statement influenced by the choice. However, unreadability and complexity of generated behavior protocols do not influence their possible usage for automated compositional verification of component-based software systems or for other software engineering tasks in any way.

5.1 Influence of parallelism on operators' detection

Here we will review operators' processing after introduction of parallelism. Firstly, sequence operator detection algorithm remains as it is. The only change is the state space: instead of state space of Java program we will consider the state space of a specific thread.

Alternative operator detection is much more complicated after the introduction of parallelism. In the presence of multiple threads each time when branching occurs, we track events of only one thread, i.e. the thread, where this branching occurred. Upon backtracking to the initial trace, again, all threads have to be considered.

As repetition operator cannot be detected in JPF state space, we do not introduce any mechanisms to detect this operator. Post-processing of the resulting behavior protocol after the state space of an application have been explored can be used to replace a multiple sequential occurrence of a part of the protocol with the repetition operator.

```

1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         FirewallImpl fw = new FirewallImpl();
6
7         Thread th1 = new FwThread(fw, true);
8         Thread th2 = new FwThread(fw, true);
9         Thread th3 = new FwThread(fw, true);
10        Thread th4 = new FwThread(fw, false);
11
12        th1.start();
13        th2.start();
14        th3.start();
15        th4.start();
16
17        try
18        {
19            th1.join();
20            th2.join();
21            th3.join();
22            th4.join();
23        }
24        catch (InterruptedException ex) {}
25    }
26 }
27
28 class FwThread extends Thread
29 {
30     private IFirewall fw;
31     private boolean enable;
32
33     public FwThread(IFirewall fw, boolean enable)
34     {
35         super();
36         this.fw = fw;
37         this.enable = enable;
38     }
39
40     public void run()
41     {
42         int i = 0;
43
44         while (i < 5)
45         {
46             if (enable) fw.EnablePortBlock("192.168.0."+i);
47             else fw.DisablePortBlock("192.168.0."+i);
48
49             i++;
50         }
51     }
52 }

```

Listing 9: Firewall test program code

```
(
    ?IFirewall.EnablePortBlock*
    |
    ?IFirewall.EnablePortBlock*
    |
    ?IFirewall.EnablePortBlock*
    |
    ?IFirewall.DisablePortBlock*
)
```

Listing 10: Expected result of running the algorithm

Furthermore, we track only one interleaving of threads (see Figure 8). This greatly decreases computational time and thus addresses state space explosion problem. This optimization is correct as we have stated two conditions: (i) that all threads are started before calling monitored interfaces' methods, and (ii) that one thread cannot influence the execution-flow of another. These conditions enforce the fact that the same behavior occurs in all interleavings, thus making traversing of other interleavings redundant. Tracking of only one interleaving is achieved by setting all scheduling-relevant CGs to done after they have been advanced once. So, the branching in JPF state space is generated only by nondeterministic data CGs.

5.2 Case study

In this section we discuss the application of BytecodeAbstractor on a test program and present the resulting BP. The program being tested is given in Listing 9. As can be seen from the listing of the program there are no references to the `Verify` class meaning that no nondeterministic data choice can occur during execution of the program. Thus all transitions in JPF state space are caused by scheduling-relevant CGs. Note also, that the program code meets the requirements stated in previous subsections, namely, that all threads are started before any method calls (lines 7-15), and no thread affects the execution flow of another thread. Class `FwThread` defined in lines 28-52 simulates the clients of the program (environment) which call the

methods (`EnablePortBlock`, `DisablePortBlock`) of the provided interface `IFirewall` (lines 46-47). Each of four threads call the methods five times (on different IP addresses). Which method will be called in a thread is defined at the moment of creation of the thread (lines 7-10). Definitions of `IFirewall` interface and the `FirewallImpl` class implementing it are of no importance in our case.

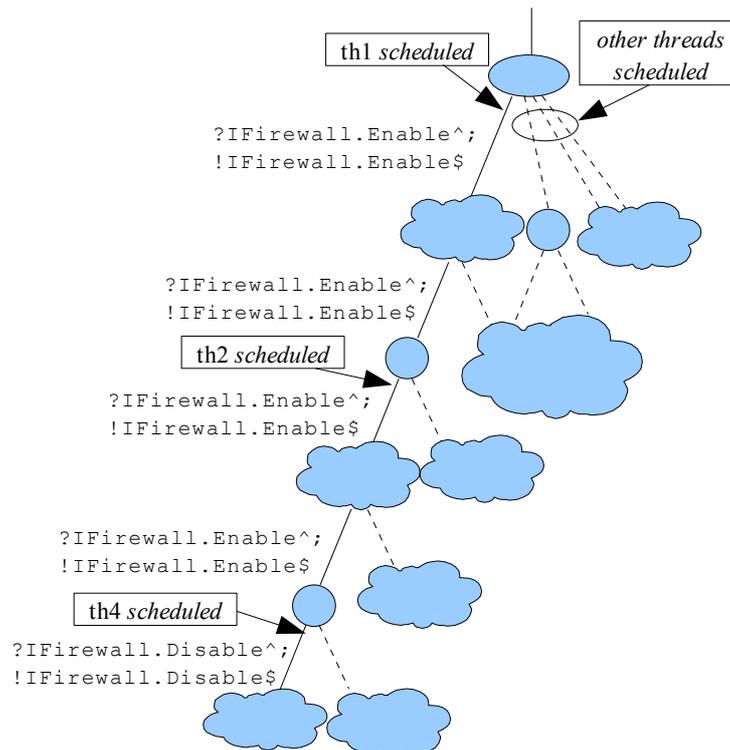


Figure 9: Firewall application state space

The result is obtained by traversing only one interleaving of the Firewall program's state space, which is shown in Figure 9 by solid lines. Dashed lines are not explored. The resulting BP expected from running of our application is presented in Listing 10. Application of the extraction algorithm presented in this thesis on Firewall program yields the BP given in Listing 11. As can be seen from both listings, obtained BP and expected BP are equal. However, the result obtained by the algorithm is more complex due to impossibility of detection of loops in JPF state space. Additionally, no

shortcuts listed in Table 1 are applied to the obtained result. Nevertheless, simple post-processing can solve these shortcomings.

```
( ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
)  
|  
( ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
)  
|  
( ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
  ?IFirewall.EnablePortBlock^ ; !IFirewall.EnablePortBlock$ ;  
)  
|  
( ?IFirewall.DisablePortBlock^ ; !IFirewall.DisablePortBlock$ ;  
  ?IFirewall.DisablePortBlock^ ; !IFirewall.DisablePortBlock$ ;  
  ?IFirewall.DisablePortBlock^ ; !IFirewall.DisablePortBlock$ ;  
  ?IFirewall.DisablePortBlock^ ; !IFirewall.DisablePortBlock$ ;  
  ?IFirewall.DisablePortBlock^ ; !IFirewall.DisablePortBlock$ ;  
)  
)
```

Listing 11: BP generated by the algorithm

Chapter 6. Related work

There are many existing techniques for extraction of model of C-E interaction from the code of software components. According to the underlying approach, they can be divided into three groups:

- (i) runtime analysis of the whole system, consisting of the component and its environment [9],
- (ii) static analysis of component's implementation, and,
- (iii) static analysis of real environment's code.

Techniques in the first group derive models from traces (sequences of events generated by an executing program that is being monitored) of important events that are recorded during execution of the whole system. For example, runtime analysis technique called *tracing* is used to (i) understand the behavior of a running program, and/or (ii) continuously capture interesting events during an execution of the program (we focus on method call-related events).

Tracing generally works on any program, however big and complex, because the decision which information (and how much) has to be stored in a trace log for further processing is done by the user. Thus an abstraction of the program's behavior is obtained. Finite state machine can be created afterwards from the trace log by merging equivalent abstract states into one. But this model (FSM) is not precise (complete), because only some execution traces have been considered.

In [17], tracing is performed “transparently” with the help of a runtime environment which monitors all calls of given methods in all processes. The model is an incomplete graph which is abstracted from the trace log. Incompleteness of the graph is caused by the fact that the model reflects only some execution paths for chosen combinations of input.

Another approach is to track software components in CBSS given in [16]. It is, in principle, runtime analysis technique as only information about some execution paths is gained. In this approach the component itself has to implement specific interface and call tracing-related methods in the same way as logging methods are called. So,

the code of the component have to be manually modified in order for tracking to function properly.

FSM inferring technique given in [18] is also based on dynamic analysis. This technique infers FSM from sets of traces which somehow were already generated using a runtime analysis-based tool, such as Java PathExplorer (JPaX) presented in [19]. It can be used to extend program's bytecode with procedures to emit events to an observer during its execution. Obtained events can be further checked against a high-level specification provided by user. The tool, however, does not support automated generation of a behavior specification directly from Java code.

Thus, although runtime analysis techniques are typically quite precise, they have incomplete coverage, as models generated by them capture only behavior that actually occurs during systems' execution. For example, the technique presented in [8] aims at extraction of a specification of correct usage of component's API. Authors assume that a specification reflects the most typical usage of the given API, because the code is mostly correct and only some parts of it are buggy. Then, the obtained specification (in the form of an FSM) can be checked and fixed by the author of the API or the user. Reliable results are produced by the technique only if, for a particular component, it is applied on many applications that use the component, e.g. Java core libraries such as `java.util` and system libraries – `glibc` and `win32`, etc. Hence, this technique cannot be used for components which are used only in a few applications.

Static analysis-based techniques, on the other hand, infer models either from the code of the component (component-side static analysis) or a client (client-side static analysis), or from both. Component-side static analysis is based on the analysis of the code of an isolated component – a model of valid usage of the component by its clients (real environment) is derived, where *valid* means that no error occurs in the component if the environment interacts with the component according to the model. Since static-analysis based techniques are inherently imprecise due to abstraction they perform, typically an approximate model of C-E interaction is extracted. For example, the technique presented in [14] extracts a specification of valid usage of a Java class in the form of a finite state machine. The extraction is performed in two steps — first, predicate abstraction is used to construct a boolean model of the Java class, and then a

variant of the L^* algorithm for learning of a regular language (FSM) is used to derive the specification.

Unlike component-side analysis, client-side static analysis aims at analysis of the code of a real environment (instead of a component) with the goal of constructing an approximate model of actual use of the component by its environment. A representative of this group is the technique proposed in [15], which extracts the model in the form of finite automaton from the environment's code via abstract interpretation-based collection of traces of important events and summarization of the traces. Only a typical usage of the component is captured, assuming that the typical usage is very often correct — spurious traces (noise) are filtered out during summarization phase.

Static analysis technique, presented in [20], is based on extraction of complete model of the code in C for the purpose of further model checking. It is similar to extraction of models as in Bandera [21] or SLAM [22]. Models extracted by this technique are not high-level behavior specifications; rather they are low-level models, which are more or less equally complex as the code itself.

The approach presented in this thesis combines advantages of both dynamic analysis-based approaches and static analysis-based approaches, and addresses drawbacks specific to each of the other classes of approaches. Specifically, it, being based on state space traversal, allows to obtain information about all traces (for all inputs and thread interleavings), thus it can be compared to runtime analysis precision-wise and has completeness of static analysis-based techniques.

However, state space traversal works mostly on smaller programs because of state explosion problem. In our approach we fight this problem by traversing only one thread interleaving for each scheduling-relevant transition. Thus, this technique achieves that the target program's state space will be dependent on the number of possible combinations of input values. Presented technique does not influence the detection of parallelism in the application – in order to determine whether two events can occur in parallel, it is enough to know the identification number of relevant threads (JPF allows this) during traversing of a single execution path. Thus there is no need to check if any given two events will occur in different orders in different execution paths. Additionally, unlike [16], our technique does not require

modifications of the source code and uses model checking, i.e. all execution paths are explored in order to obtain a behavior model.

Chapter 7. Conclusion

Our implementation does not pretend to be a complete tool for automatic extraction of models of C-E interaction from component code. It has an exploratory role as it is the first attempt to automatically infer a behavior protocol of given software system by traversing its state space. The abstraction of a component yielded by the BytecodeAbstractor tool can further be used, e.g., to model check against another component implementing the same interfaces [1].

The next logical step in further development of BytecodeAbstractor tool would be the implementation of post-processor of inferred behavior protocols in order to make the protocols more compact using abbreviations, which are described above in this thesis and to replace multiple sequential occurrence of the same expressions with repetition operator. In order to do this, behavior protocol parser will be needed.

It is possible to improve the algorithm presented in the thesis to support Extended Behavior Protocols (EBP) [10]. For example, support for state variables can be added. State variables store information with scope beyond a single method call to model component states. The challenge will be to find an appropriate mapping from EBP state variables to Java program variables. Then it would be possible to relate events (method calls) onto specific states (state variables' values).

Another possible improvement of BytecodeAbstractor could be some heuristics to speed up the generation of behavior protocol and solving POR inferred problems, which are discussed above.

It is also possible to modify our tool so that it can be executed in parallel on different machines and the resulting behavior protocol would be inferred by putting the results taken from all machines together.

Bibliography

1. P. Parizek, F. Plasil, J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker.
2. C. Neumann. Converting Deterministic Finite Automata to Regular Expressions. Mar. 16, 2005.
3. C. Szyperski. Component Software – Beyond Object-Oriented Programming, First Edition, Addison-Wesley, 1998.
4. Fractal component model: <http://fractal.objectweb.org>
5. SOFA component model: <http://sofa.objectweb.org>
6. Java PathFinder: <http://javapathfinder.sourceforge.net>
7. Enterprise Java Beans Specification, version 2.1, Sun Microsystems, Nov 2003. <http://java.sun.com/products/ejb>
8. Ammons, 2002: G. Ammons, R. Bodik, J.R.Larus, Mining specifications, 2002.
9. D. Lie, A. Chou, D. Engler, D.L.Dill. A Simple Method for Extracting Models from Protocol Code.
10. Kofron, 2007: J. Kofron, Behavior Protocol Extensions, 2007?
11. COM: Component Object Model Technologies.
<http://www.microsoft.com/com>
12. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures, In Proceedings of the 5th European Software Engineering Conference (ESEC), LNCS, vol. 989, 1995.
13. F. Plasil and S. Visnovsky. Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, 2002.
14. Alur, R., P. Cerny, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes, ACM SIGPLAN Notices, 40(1), 2005.
15. Shoham, S., E. Yahav, S. Fink, and M. Pistoia. Static Specification Mining Using Automata-based Abstractions. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM Press, 2007.
16. Gao, J. Zhu, E.Y. Shim, S. Lee Chang. Monitoring software components and

- component-based software. In Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International.
17. Thomas Arts, Lars-Åke Fredlund. Trace analysis of Erlang programs. In Erlang Workshop, ACM SIGPLAN, 7 Oct 2002, Pittsburgh, Pennsylvania, USA.
 18. D. Lorenzoli, L. Mariani, and M. Pezzè. Inferring state-based behavior models. In proceedings of the International Workshop on Dynamic Analysis (WODA) co-located with the 28th International Conference on Software Engineering (ICSE), 2006.
 19. K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.
 20. D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In Proceedings of the 28th Annual International Symposium on Computer Architecture, July 2001.
 21. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In Proceedings of the International Conference on Software Engineering (ICSE 2000), pages 263-276, Nov. 2000.
 22. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Proceedings of the SPIN 2000 Workshop on Model Checking of Software (LNCS 1885, Springer), pages 242-252, Aug. 2000.

Appendix A. List of used abbreviations

| | | |
|--------------|---|--------------------------------------|
| ADL | – | Architecture Definition Language |
| API | – | Application Programming Interface |
| BP | – | Behavior Protocols |
| C-E protocol | – | Component-Environment protocol |
| CBSE | – | Component Based Software Engineering |
| CBSS | – | Component Based Software System |
| CG | – | Choice Generators |
| COM | – | Component Object Model |
| EBP | – | Extended Behavior Protocols |
| FSM | – | Finite State Machine |
| GUI | – | Graphical User Interface |
| JPaX | – | Java PathExplorer |
| JPF | – | Java PathFinder |
| JVM | – | Java Virtual Machine |
| LTS | – | Label Transition System |
| POR | – | Partial Order Reduction |
| SOFA | – | Software Appliances |
| TSCG | – | Thread-Scheduling Choice Generator |
| VM | – | Virtual Machine |
| XML | – | eXtended Markup Language |

Appendix B. Contents of CD ROM

This thesis is accompanied by the CD ROM containing binaries and source codes of the implementation and a set of examples. The CD ROM is organized as follows:

`/bin/`

Contains the scripts for conveniently running BytecodeAbstractor

`/build/BytecodeAbstractor.jar`

Executable JAR archive containing build of BytecodeAbstractor

`/build/samples/`

Compiled sources of BytecodeAbstractor sample programs

`/build.xml`

Ant script for building the application, creating executable

BytecodeAbstractor JAR archive. Contains also the scripts to run the samples

`/doc/thesis`

Electronic version of this document

`/input/`

Directory to put BytecodeAbstractor target application. Additionally, contains program input files. For convenience, compiled samples are also placed here

`/lib/`

Libraries required for building BytecodeAbstractor

`/prerequisites/`

Software prerequisites of the application: Sun Microsystems JRE 1.6 for Windows OS. Additionally, for convenience, the folder contains Ant tool and Sun Microsystems JDK 1.6 for Windows OS

`/samples/`

Directory with sample applications

`/src/`

Source codes of the BytecodeAbstractor

Appendix C. User manual

C.1 Installation

BytecodeAbstractor program is developed as an extension for the Java PathFinder (JPF) tool, which is a model checker for Java bytecode programs that requires Java 1.5 or above to run.

Although BytecodeAbstractor was developed as an extension for JPF, the user does not need to install JPF and the tool separately. BytecodeAbstractor includes JPF in its installation and therefore JPF is not given as a prerequisite.

BytecodeAbstractor does not need specific installation to run. Simply copy the content of the distribution directory to the directory for BytecodeAbstractor and run the tool, see Chapter C.2.

Source code of BytecodeAbstractor tool is also included as a part of the distribution. So, user can modify the application if he or she wants to. Before running BytecodeAbstractor with these modifications a few steps have to be performed:

1. Close running BytecodeAbstractor, if any.
2. Execute ANT target “jar” (*build.xml*).
3. Run BytecodeAbstractor.

C.2 Running BytecodeAbstractor

This chapter presents instructions on how to run BytecodeAbstractor tool. It is highly recommended to study the execution procedures of the JPF tool (see JPF documentation) before executing BytecodeAbstractor itself.

BytecodeAbstractor can be run in several ways. Before launching BytecodeAbstractor, it is necessary to provide the compiled sources of the target application, see Section C.2.1. Sections C.2.2 and C.2.3 will describe different running procedures in detail. Description of BytecodeAbstractor GUI is given in Section C.3.

Section C.4 gives the instructions on how to launch BytecodeAbstractor on a sample target application demonstrating the functionality of the tool.

C.2.1 Providing target application sources

In order to construct an abstraction of the component, BytecodeAbstractor needs to be given .class files of the complete target application. The compiled sources have to be put into a directory which JPF VM can access.

The distribution directory contains *input/* directory that is included into the class path of JPF VM (*vm.sourcepath* property in *jpf.properties* file). Compiled sources of the target application can be put into that directory to ease the procedure of providing target application .class files to BytecodeAbstractor tool.

Alternatively, the path where BytecodeAbstractor will search for target sources can be changed by modifying *vm.classpath* property in *jpf.properties* file.

C.2.2 Command line execution

Executing BytecodeAbstractor from the command line can be done by typing the following:

```
> bin\run [-c config-file] [+key=value]
[-output=output-file] -main=Main-class -input=input-file
```

Arguments:

- **-c configuration file** – optional, configuration file to be used (default is *jpf.properties*)
- **+key=value** – optional, run-time override of configuration properties
- **-output=output-file** – optional, specifies the file name for the application output (default is *./bc2bp.out*)
- **-main=Main-class** – specifies the path to `Main` class of the target application relative to *vm.classpath*
- **-input=input-file** – specifies the path to XML input file describing events and associated (provided/required) interfaces

C.2.3 Execution using BytecodeAbstractor GUI

For the convenience, BytecodeAbstractor tool is provided with GUI allowing a

user to specify all input parameters in a simple and intuitive way. To run the GUI simply type the following in command line:

```
> bin\runGUI
```

C.3 BytecodeAbstractor GUI description

BytecodeAbstractor comes with the GUI, which allows the user to conveniently run the program without the need to type all the necessary input in command line. Main window of the GUI is given in Illustration 1. It allows the user to specify the main class which will be executed by JPF, BytecodeAbstractor input specification file and the output file, which will contain generated behavior protocol once the program finishes.

Field “Class name to be executed” lists all the classes in the application class path (the classes in *bcel.jar*, *env_jpf.jar*, *env_jvm.jar* and *jpf.jar* are not listed). So the target program has to be in BytecodeAbstractor class path, this is achieved, e.g. by copying the target program into *input/* folder.

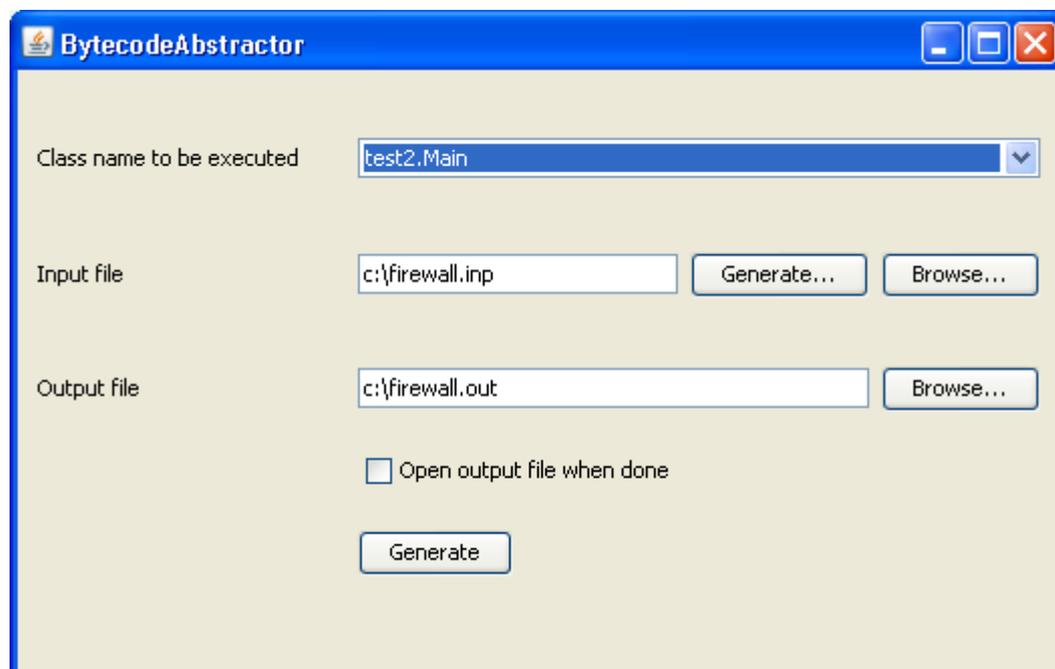


Illustration 1: BytecodeAbstractor application GUI.

Fields “Input file” and “Output file” specify where to find an input file, respectively, file to write BytecodeAbstractor output to. In addition, application input file can be generated by pressing “Generate...” button. See next section for the instructions on how to generate the application input file. It is also possible to select whether BytecodeAbstractor should open the output file once the program is finished. This is done by checking “Open output file when done” check box.

C.3.1 Generating program input

After pressing “Generate...” button on the GUI's main window input generation window is displayed (Illustration 2). This window lists all the interfaces and the classes implementing some interfaces. Initially, those classes are listed in the “Required interfaces” list. Provided interfaces then can be moved to “Provided interfaces” list by selecting an interface in “Required interfaces” list and pressing “Switch panes” button and vice versa. Interfaces which are neither provided nor required and nonetheless are still listed in the window can be deleted by selecting an interface and pressing “Delete” button.

When some interface, either provided or required, is selected, the field “Selected interface methods” lists all the methods this interface defines. An unnecessary method can be deleted by selecting this method and pressing “Delete” button. If the method being deleted is the last one for selected interface, this interface is deleted as well. Selecting a method also causes the event name to be displayed in “Event name” text box. The event name is constructed using the template “Classname.Methodname”. User can modify the event name if he or she wants to by modifying the text box and pressing Enter. After everything is set up correctly, the input file can be generated by pressing “Create file” button. Once the file has been created the “Input file” field is modified to point to the generated file.

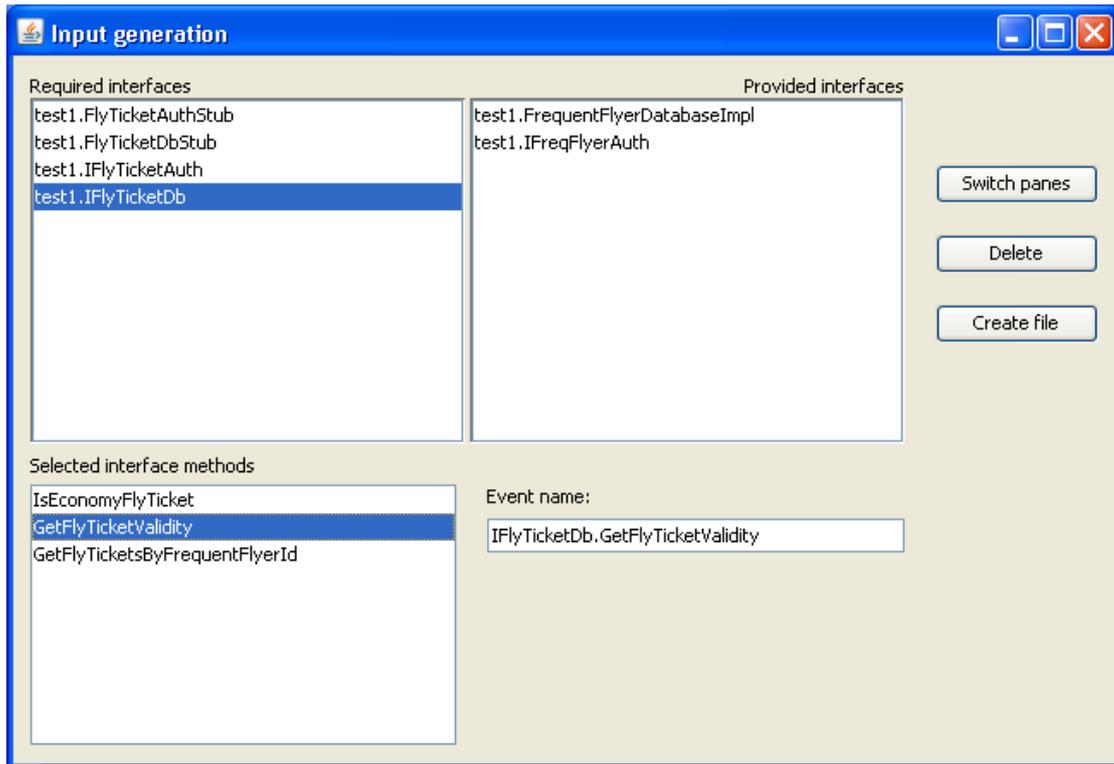


Illustration 2: Program input generation window.

C.4 Sample component abstraction

This section describes all the steps necessary to launch BytecodeAbstractor on a sample *FrequentFlyer* application. Input file specifying interfaces to trace and compiled source files of the sample can be found in *samples/* and *samples/frequentFlyer/* directories, respectively.

After all the following steps are successfully completed the default output file *bc2bp.out* will be generated in *bin/* directory.

The step-by-step tutorial follows:

1. Set the target application

Copy .class files in *samples/frequentFlyer/* directory into a directory specified in *vm.classpath* property, default is *input/* directory. Thus, the .class files will be placed in *input/frequentFlyer/* directory.

2. Set the input file

Copy input file *FrequentFlyer.inp* situated in *samples/* directory into *input/* directory.

3. Run BytecodeAbstractor

After all previous steps are accomplished, BytecodeAbstractor can be run using one of the ways specified in Section C.2, for instance:

```
> bin\run -main=FrequentFlyer.Main  
-input=../input/FrequentFlyer.inp
```