Charles University, Prague
Faculty of Mathematics and Physics

MASTER THESIS

Pavel Jančík

Checking Compliance of Java Implementation with TBP Specification

Department of Distributed and Dependable Systems
Supervisor: RNDr. Pavel Parízek, Ph.D.
Study program: Computer Science, Software Systems

2010

Words cannot express my gratitude to my advisor Pavel Parízek. I would like to thank Pavel for his personal and professional approach, useful advice and suggestions, assistance and because he opened area of verification to me.

I would like to thank Kateřina Topilová for language corrections. Finally I want to thank my family and Iva Doležalová for their support in my life and studies.

Prohlašuji, že jsem svou diplomovou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I wrote this master thesis on my own and listed all used sources. I agree with making this thesis publicly available.

Prague, August 4, 2010 ....

Pavel Jančík

Název práce: Checking compliance of Java implementation with TBP Specification
Autor: Pavel Jančík
Katedra (ústav): Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: RNDr. Pavel Parízek, Ph.D.
e-mail vedoucího: pavel.parizek@d3s.mff.cuni.cz

Abstrakt:
Verifikace je postup umožnující zlepšit spolehlivost aplikací složených z komponent. Jedním z aktuálních témat je kompozice komponent a ověřování, zda-li je tato kompozice korektní z pohledu komunikace mezi nimi. Pro popis této komunikace (chování) slouží behavior protokoly (BP).
V současné době jsou již vyvinuty nástroje ověřující kompozici komponent. Tyto nástroje ovšem fungují na bázi porovnávání BP a tedy implicitně předpokládají, že implementace komponenty odpovídá danému BP. Odtud vyplývá důležitost nástroje na ověření, zda chování dané implementace odpovídá připojenému BP.
Pravidla pro komunikaci komponent společně s BP se obvykle vytvářejí v raných fázích vývojového cyklu. Během implementace typicky dochází k různým ať již úmyslným či neúmyslným, odchylkám od původního návrhu. Tento přístup k vývoji software ještě zvyšuje důležitost ověření, zda aktuální implementace vyhovuje původnímu návrhu BP.
Výsledkem práce je nástroj, který ověřuje, zda chování implementace jedné komponenty v jazyce Java odpovídá chování definovanému v připojeném threaded behavior protokolu. Dále práce obsahuje vyhodnocení nástroje na větších příkladech pro nastínění aplikovatelnosti přístupu v praxi.
Klíčová slova: verifikace, model checking, komponenty, behavior protokoly

Title: Checking compliance of Java implementation with TBP Specification
Author: Pavel Jančík
Department: Department of Distributed and Dependable Systems
Supervisor: RNDr. Pavel Parízek, Ph.D.
Supervisor's e-mail: pavel.parizek@d3s.mff.cuni.cz

Abstract:
Verification is a method of increasing reliability of component based applications. Component composition and its verification from the communication point of view, is one of the current research topics. Behavior protocols (BPs) are an abstraction used to describe communication among components.

Tools, that are able to verify component composition, have already been developed. These tools concern only the BP level and they implicitly presume that implementation of the components conforms to given BP. The importance of verification of implementation compliance with BP is obvious.

The BP and rules for communication among components are created during the early steps of the development cycle. Various deviations from the original BP (intentional or unintentional) arise during the implementation phase. This approach reflects the importance of a checker tool.

Checker tool, which will ensure conformance of a single component implementation written in Java with given threaded behavior protocol, is the final result of our work. The work also contains an evaluation of the tool in bigger real life examples.
Keywords: verification, model checking, software components, behavior protocols

Table of Contents

1 Introduction....5
1.1 Components....5
1.2 Verification....5
1.3 Behavior protocols....6
1.4 Component verification....7
1.5 Goals....8
1.6 Structure of work....8
2 Background....9
2.1 Threaded Behavior Protocol....9
2.1.1 TBP description....9
2.1.2 TBP Example....12
2.2 TBPLib....14
2.2.1 TBP specification processing....14
2.2.2 Automatons....15
2.2.3 Resolved provisions....16
2.3 Java PathFinder....17
2.3.1 State space traversal....17
2.3.2 Extensions....18
3 Checker....20
3.1 Environment creation....20
3.1.1 Parameters of generated calls....21
3.1.2 Relation between environment and provisions....21
3.1.3 Generating of the environment....22
3.2 TBP checking....24
3.2.1 Automatons handling....25
3.2.2 Variables evaluation....29
3.2.3 Checker state and backtracking....33
3.2.4 Program state extension....34
3.2.5 Handling of threads....36
3.2.6 Synchronization....38
4 Implementation....39
4.1 Checker overview....39
4.2 Environment generator....44
4.3 Exception handling....47
5 Evaluation....48
5.1 Experiments....48
5.2 Future work....50
6 Conclusion....52
7 Bibliography....53
Appendix A - TBP Grammar....54
Appendix B - Checker configuration....57
Appendix C - User documentation....59
Appendix D - Content of attached CD-ROM....60

1 Introduction

1.1 Components

Large software systems are nowadays often built from components that allow the use of the "*divide and conquer*" approach. By splitting an application into small pieces with well-defined dependencies, we can fight the complexity of large systems.

For the purposes of this work we assume that a *component* is a software module that encapsulates a set of related functions and meta-data. Meta-data describe provided services and dependencies of the component as well as other properties of the component. To be able to verify components, we assume that a behavioral specification is attached to each component. From the programmer's point of view, a component is a class or a set of classes. The class has to fulfill the requirements specified by the component system, which for example means that some interfaces exist that the class has to implement.

There are two main types of component systems - flat and hierarchical. Hierarchical systems support component nesting. There are two types of components in hierarchical systems - primitive and composite. *Composite components* are build from sub-components. *Primitive components* do not have internal structure as the composite ones, they consist of implementation in a programming language. Examples of hierarchical component systems are *SOFA* and *Fractal*.

Flat component systems support only primitive components. Well known examples of the flat component systems are *Component Object Model* (COM) and *Enterprise JavaBean* (EJB).

There are many interesting topics related to components - describing component composition, managing component life-cycle, component distribution, specification and verification of component behavior. This work focuses on verification of a primitive component implementation.

1.2 Verification

There are lot of ways how to produce reliable software. Production of a software begins with a managed development process. Thorough testing is an essential part of the development cycle. However, some errors are hard to find by testing - for example concurrency errors like race conditions. It is hard to ensure that testing covers all execution paths or all parallel thread interleavings. Moreover, testing is able to discover errors, but can not prove that there are no errors. Such guarantee is needed for really reliable systems.

Verification is useful in many steps of development cycle for its ability to find errors. The sooner an error is found, the cheaper its correction. During the analysis part of the development cycle the model of the system can be verified by model checkers to ensure that the system will behave in an expected way. On the other hand, later in implementation and testing phase, checkers can prove that implementation corresponds to the model and that implementation is free of specific low level errors.

Different approaches how to verify properties against the implementation exist. Human made proof is the oldest one. People are able to use interesting extraordinary ideas to solve really hard problems. On the other hand there is problem with scaling and sometimes they are the source of errors. So we will focus only on the automated or interactive verification. Automated verification is done by a checking tool. In the interactive case humans provide only hints, main part of the work is done by tools.

As we have mentioned before we verify the source code against behavior specifications. Behavior is typically described in temporal logics (LTL or CTL) or in specialized languages - for example behavior protocols, BLAST checker's own language etc. The formalism we use in our work is *behavior protocols*. We describe it more in depth later in this work (in section 2.1).

There are two main approaches used in the tools for automated source code verification – model checking and static analysis. The *model checking* is the most suitable approach for the verification of code against temporal behavior specifications. Model checkers traverse all possible program states. For each program state it checks whether the properties hold or not. Existing program model checkers can be divided into two groups - checkers that work like VMs and checkers based on predicate abstraction.

Checkers in the first group are able to check multi-threaded applications. They incorporate (on-the-fly) partial order reduction to reduce the number of possible thread interleavings. The Java PathFinder or Zing checker are in this category.

Checkers in the second category group program states together and process more states at once. Predicate abstraction is usually used to group states together. Typically they use the "*counter-example guided abstraction refinement*" (CEGAR) approach. First a coarse (imprecise) abstraction is used. If the abstraction violates given requirements then counter-example of the violation is analyzed for feasibility. If the counter example is feasible then an error is reported, otherwise the infeasible trace is used to refine the abstraction. Then the cycle continues with the new more precise abstraction.

Checkers in both groups assume that they are given a closed system- that means we have to provide an executable program which contains main method and does not depend on any input from the environment (no network, keyboard or other input). In our case we check a single primitive component implementation so we do not have a closed system. This problem is in the technical papers sometimes called a "*problem of missing environment*" [1] [2]. To solve it, we have to generate the environment. More details about the generating can be found in chapter 3.1.

1.3 Behavior protocols

The family of behavior protocols was designed at Faculty of Mathematics and Physics, Charles University. These behavior specification languages are used in SOFA and Fractal component models. The protocol family consists of initial "*Behavior protocol*" (BPs) followed by "*Extended behavior protocol*". The newest member is "*Threaded behavior protocol*" (TBP) which is described later in chapter 2.1.

Behavior protocols are formal high level models of component behavior, based on process algebras. The term "*Component behavior*" means communication among

components. It includes calls to provided and required interfaces of the component so that the verification can detect the communication or composition errors. Behavior protocols enable us to check whether the implementation fulfills the specification. BPs allow reasoning about correctness of component composition or component substitutability.

The most important concept of the BPs are *events*. They represent calls to interfaces of the component. Events are atomic actions. Only one action can be processed at a time. There are four different types of events: calls and returns to/from the provided and required interfaces. In the original version of BP all types of actions were represented by one expression. In newer versions they are separated.

1.4 Component verification

The goal of verification is to prove or disprove that given properties hold. In component composition verification we are interested in the following 3 scenarios - composition correctness, component substitutability and code conformance.

Composition correctness verification aims to prove that there are no communication errors in the component based application. Possible communication errors are bad activity and no activity. *Bad activity* error means that one component can generate an event which the second component cannot handle according to its behavior specification. *No activity* error occurs when at least one component, that expects events to be able to finish, exists, but there is no component or thread that can generate these events. The no activity error unlike the bad activity one is violated passively, the violation is not connected to any event. The composition correctness verification supports the development of reliable component based applications.

On the other hand *component substitutability* verification takes place in the maintenance part of the program life cycle, when part of the program needs to be changed. There are two types of component substitutability - for general and specific application. *Substitutability for a specific application* is intended to verify whether a component can be replaced by another component in the given composition. This is appropriate when newer version of the application is created. The g*eneral substitutability* verification detects if a component can be replaced by the given one in all possible compositions. Such substitutability is inspired by the off-the-shelf components vision. In this scenario the author of the component checks if the newer version of the component can be sold as a safe replacement of the older one.

Code conformance verification ensures that current component implementation conforms to its behavior specification. The code conformance proof is important, because tools that check the other scenarios use behavior specification as their input, which means they assume that implementation and behavior specification match.

1.5 Goals

The goal of our work is the implementation of a tool that will verify the code conformance of a single primitive component written in Java against a specification in the TBP. This tool should be easy to use.

The proposed solution should:

1. solve the problem of missing environment by generating an artificial environment for the component.
2. provide the JPF extension which will gather sequences of events on component interfaces and check if these sequences of calls are permitted by the TBP.

Secondary goal of this thesis is to gain experience from the implementation part. That means to emphasize on some of the troublesome TBP properties that complicate the checking, or to point out some language constructs that are hard to handle with the TBP.

A tool for checking the code conformance against original BP has already been implemented, but it does not use the TBP as its input. Our implementation should only use ideas from this tool, but should not be an extension of this tool.

1.6 Structure of work

In chapter 2 we introduce tools and formalisms used in our work. Chapter 3 describes the ways to fulfill our goals. In this chapter we also introduce the ideas and methods we use.

In chapter 4 we describe how the checker and the environment generator are implemented. Ideas from the 3rd chapter are applied to the implementation.

Fifth chapter contains an evaluation of the checker on real life size applications. In the 6th chapter we summarize the results of our work.

2 Background

2.1 Threaded Behavior Protocol

Threaded behavior protocol [3] (TBP) is a language for specification of SW components behavior. TBP was designed with the syntax of an imperative programing languages in mind. It introduces java-like synchronization, conditions and loops and separates behavior description into 2 different parts - provisions and reactions. *Provisions* describe the permitted use of the component, ie. legal order of calls to provided interface. *Reactions* describe method behavior in terms of calls to required interfaces, ie. possible call sequences caused by each provided method.

2.1.1 TBP description

TBP specification consists from 5 parts. There is a small example of the TBP protocol at the end of this section.

Type definitions section. Enumeration types that could be used later in the specification are defined here. These types are used for variables and method parameters, type consists from a name and a set of unique entries. The entries represent all possible values for variables of given type. There is one predefined type – mutex. *Mutex* is an enumeration type as well and it can be used for synchronization in the TBP specification. Mutex has two entries - locked and unlocked.

The *variables section*. In this section the user defines variables that could be referred to later in the reaction and thread section. Each variable has its own name, type and initial value. All these values are specified in the declaration of the variable. Each variable is always well defined (unlike in some programming languages where uninitialized variables are possible). The variable is visible only inside the component where it was declared.

The *provisions section.* The syntax of provisions is based on the original BP. Provisions grammar is similar to regular expressions. Method call, which is one of the basic provisions, is represented by an expression in form *?interfaceName.methodName(parameters):retVal*, where the character '*?*' means that the component accepts the call to provided interface. The *interfaceName* specifies the name of the provided interface which accepts the event. The same holds for the method name. The parameters can be thought of as an abstraction of permitted values of the parameters used in the implementation. The parameters refine the behavior of the method. Count and types of the parameters have to match the method definition in the reaction section. The *retVal* value of the method is optional. Another basic provision is *NULL*, that declares that no call is accepted. More complex provisions can be created by operators - sequencing '*;*' repetition '*', alternative '+', and-parallel '|', or-parallel '||', and by unary operators for reentrancy - unlimited

'|?' and limited '|n', where n is decimal positive number. Parenthesis can be used to override default operator precedence.

The *sequencing operator* means that the events of the left operand take place first and they are followed by the events of the right operand. The *repetition operator* stands for zero up to any finite number of repetitions of its operand. The *alternative operator* permits events either from left or right operand. Intermixing of events from both operands is denied.

The *and-parallel operator* ('|') permits interleaving of operand events. Let *I* be any interleaving of events from the *A | B* expression. If we remove all events generated by the operand *A* in *I*, then remaining events are permitted by the second operand *B*. That means that events of both operands must occur. The events from each operand have to be in the order permitted by that operand. But all interleavings of events generated by the operands are permitted. For example *(A ; B) | (X ; Y)* expression is equivalent to *(A ; B ; X ; Y) + (A ; X ; B ; Y) + (A ; X ; Y ; B) + (X ; A ; B ; Y) + (X ; A ; Y ; B) + (X ; Y ; A ; B)*.

The *or-parallel operator* ('||') is syntactic sugar. It permits parallel composition of operands but does not insist that both operands are fulfilled. *A || B* is equivalent to *A + B + (A | B)*.

The *limited reentrancy operator* ('|n') can be replaced by series of or-parallel operators. The *(A |n)* expression is equivalent to *A || A || .... || A* where *n* is count of A in the resulting expression.

The *unlimited reentrancy operator* ('|?') may be replaced by the limited one. The number used for limiting the operator is derived from the composition, it is the number of threads in all components [4]. The behavior protocols do not allow recursion (even indirect) in events. It is not permitted to call itself in the reaction.

In the *reaction section* there is a behavior description of the methods defined in the component. More precisely it describes provided interfaces that the component implements and internal methods which are not visible outside the component. There are three types of basic statements in the reaction section - the first is the *variable assignment* ('<-' operator). This statement can assign values to the variables defined in the variables section. The second basic statement type is the *call expression* in form *! interfaceName.methodName(parameters)*, where the '!' character means emitting - the component calls its required interface method. The interfaceName specifies the name of the provided interface of the method called. The parameters are represented by a list of constants and variables. The parameter count and types have to match the parameters defined in the reaction of the method called if such a reaction is present. The third basic statement is *return*. Return means the end of a method as in common programing languages.

Such basic statements can be combined into more complex ones by the *sequencing ';' operator*. Note that its semantics is similar to standard programing languages but in the TBP case it is a binary operator so both operands have to be present. No semicolon after the last statement is permitted. Another way to combine the statements represent *while* and *if* expressions. After the keyword there should be a guard in round brackets and then a statement in braces. In the *if* statement there is a voluntary *else* branch. Figure 2.1 shows the grammar of the *if* statement. In figure 2.2 there is a syntax diagram of the *while* statement. Semantics is similar to the same constructs of common programing languages.

Guards represent conditions in the TBP. There is special guard '*?*' that means non-deterministic choice. Both true and false (else) branches are permitted, it is chosen nondeterministically, which branch will be used. The basic form of the guards is checking for equality or non-equality of a variable with constant. Figure 2.3 shows valid guard forms.

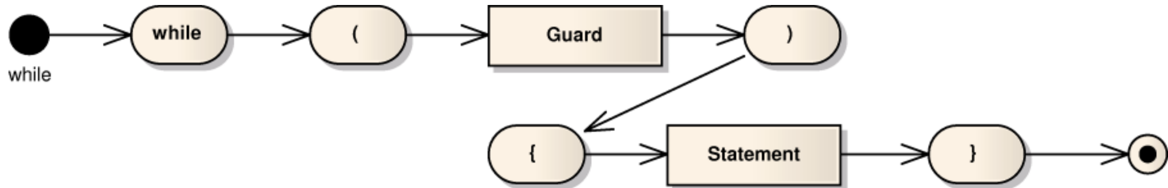

Figure 2.1: While statement grammar

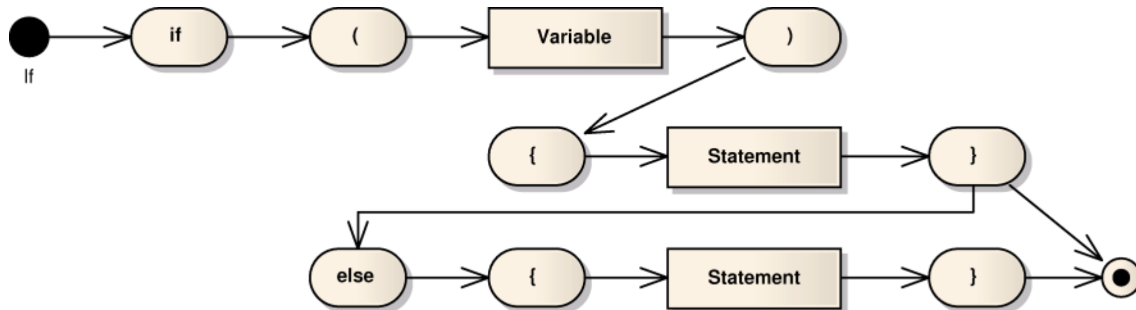

Figure 2.2: If statement grammar

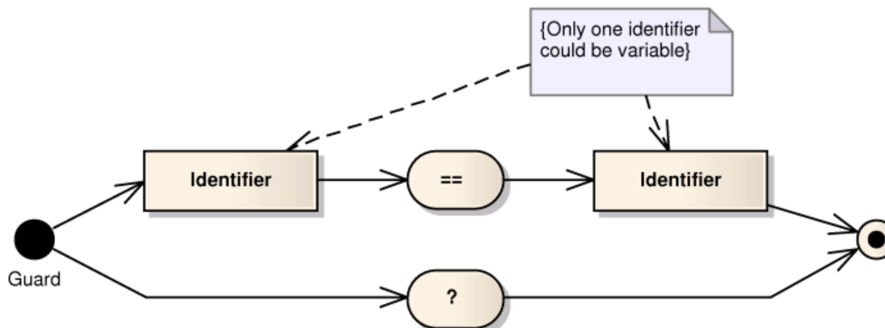

Figure 2.3: Guard grammar

Another supported statement is *switch*. The grammar for the switch statement can be found in figure 2.4. After the keyword there is a variable name in round brackets and then the switch body. The switch body consists of a list of cases with possible *default* case at the end. Each case consists of a constant value, colon character and a statement in braces. In contrast to Java or C language there is no *break* keyword. Only the one matching branch is executed. There is one special switch form, non deterministic operator '*?*' is used instead of a variable name in this form. The keyword '*case*' is used instead of constants. That means that all cases are permitted and one of them is selected nondeterministically.

The last imperative statement is *sync*. This statement is used for synchronization of parallel actions. Its syntax is straightforward and it is described in figure 2.5. After the sync keyword there is a variable name of mutex type in brackets followed by a statement in braces. When any thread is in the synchronized section no other thread can enter the same synchronization section or into synchronized section with the same mutex variable. Mutexes are considered to be standard TBP variables. There are no queues of waiting threads associated to mutexes.

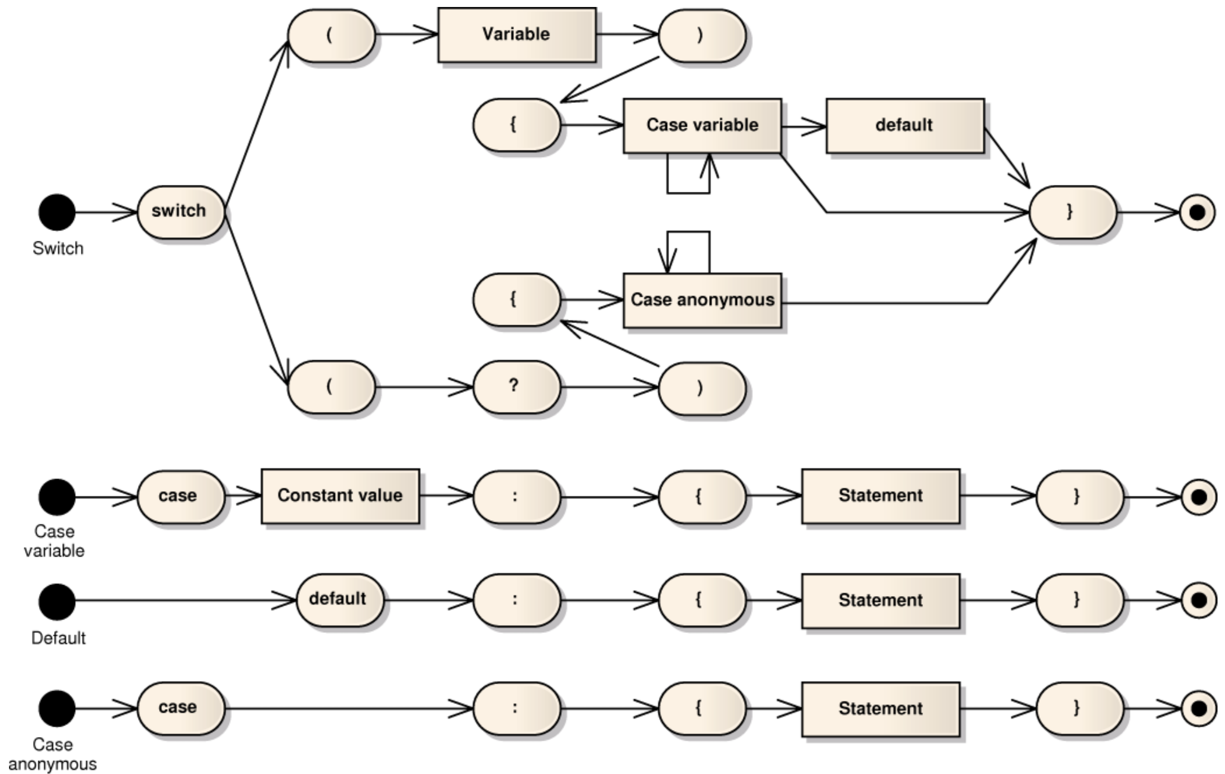

Figure 2.4: Switch statement grammar

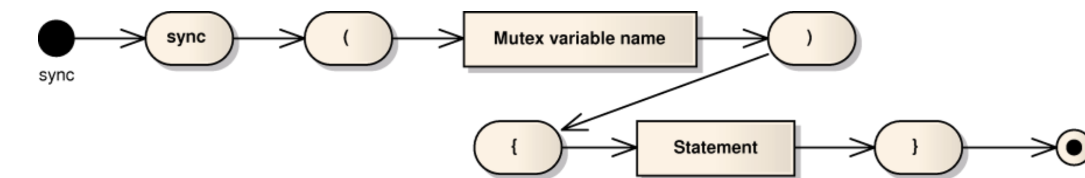

Figure 2.5: Sync statement grammar

Thread section. Behavior of threads that were created by the tested component is described in this section. The TBP supports empty thread section for components that do not create their own threads. TBP does not support creation of threads at runtime - the number of threads cannot change during the execution. Syntax of the thread section is the same as in the reaction section described above. Each thread created by the implementation must have its representation in the TBP protocol.

2.1.2 TBP Example

In the following paragraph we show a brief example of the TBP protocol and how it corresponds to the expected behavior. Assume the TBP protocol from figure 2.7.

In the provisions section we can see that up to 5 threads can call the P1.A() method in parallel (line 14). Assume that 2 such threads exist and both call the P1.A() method. The diagram 2.6 shows one of the possible event traces permitted by the protocol. The two swimmlines represent threads, time goes from top to down. Rectangles represent call events, rounded boxes represent internal TBP protocol steps. The call events (rectangle boxes) are observable.

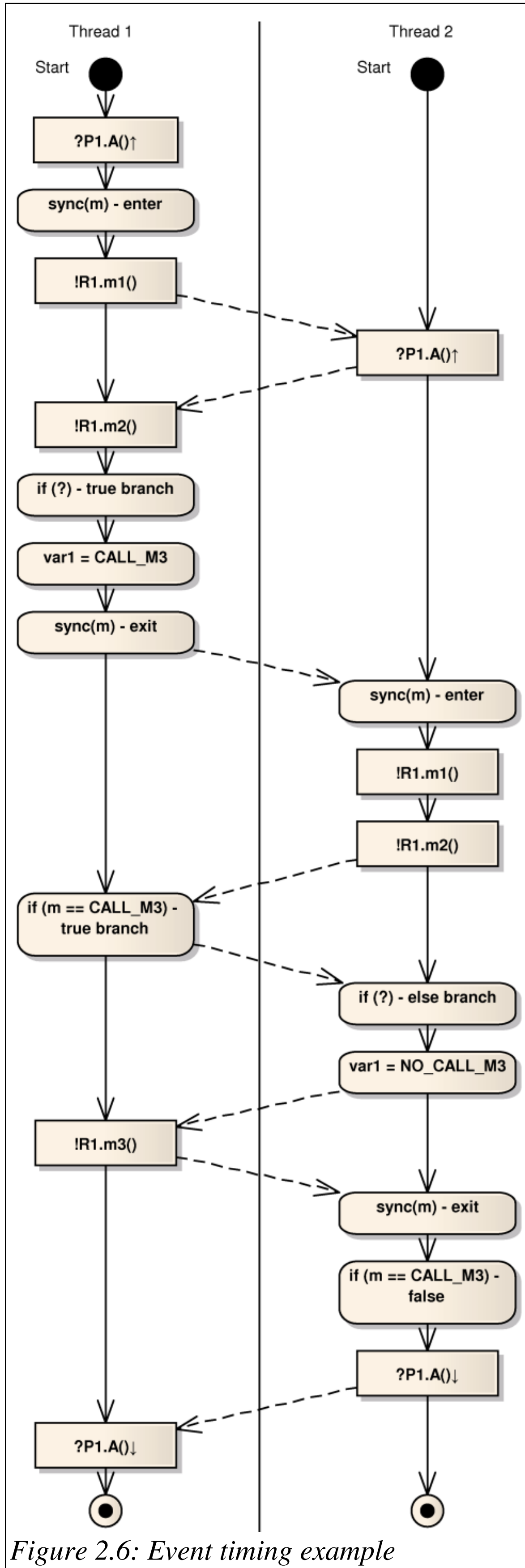

Figure 2.6: Event timing example

```
01 : component Component1 {
02 : types {
03 :   states = {
04 :     CALL_M3,
05 :     NO_CALL_M3
06 :   }
07 : }
08 : vars {
09 :   states var1 = CALL_M3
10 :   mutex  m
11 : }
12 : provisions {
13 :   example_provision {
14 :     ( ?P1.A() + NULL ) | 5
15 :    }
16 : }
17 : reactions {
18 :   P1.A() {
19 :     sync(m) {
20 :       !R1.m1() ;
21 :       !R1.m2() ;
22 :       if (?) {
23 :         var1 <- CALL_M3
24 :        } else {
25 :         var1 <- NO_CALL_M3
26 :       }
27 :     } ;
28 :     if (var1 == CALL_M3) {
29 :       !R1.m3()
30 :     }
31 :   }
32 : }
33 : threads {
34 :   EmptyThread { NULL }
35 : }
36 : }
```

Figure 2.7: TBP example

On the other hand not all event traces are allowed. If we assume that only one thread calls ?P1.A() method then event trace *?P1.A() ; !R1.m1() ; !R1.m3()* is not allowed. It is easy to see that there is no path between the !R1.m1() call and the !R1.m3() call that skips !R1.m2() in the P1.A reaction.

Another not so trivial example of denied call sequence follows. There can be no event trace that contains *!R1.m1() ; !R1.m1()* pattern. The thread, which emits first !R1.m1() call, has to be still in the sync section, because no !R1.m2() call takes place. So no other thread can enter into the sync section to be able emit the second !R1.m1() call.

2.2 TBPLib

The TBPLib is an open source library developed by DSRG[1]. It helps process the TBP protocols. The library exists in more versions, this chapter describes one of the older ones, which was used in the checker tool. Newer versions differ mainly in the 3rd (Automatons) and 4th (In-lining) stages. The library is separated into four main stages (see figure 2.8). At first the library parses given input protocol, then the (derivation) syntax tree is resolved and transformed into automatons. In the end the automatons are in-lined.

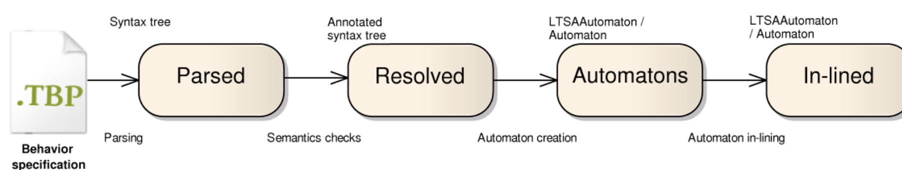

Figure 2.8: Processing of the TBP Specification

2.2.1 TBP specification processing

The first *parsing stage* is similar to lexical and syntax analysis in compilers. The TBPLib uses ANTRL to parse the input and to create derivation trees. Grammar files in tpblib/src/grammars directory are a very good reference for the TBP protocol syntax. Each part of the TBP protocol (types, variables, provisions, reactions and threads) is processed separately.

Types are translated directly into `Typedef` instances. The `Typedef` class represents user defined enumeration types. These instances hold type name and a list of permitted values. It is the final representation of types in the TBPLib. Each variable from the variables section is defined by a single simple node that holds strings with the variable name, type and initial value. TBPLib represents values of variables as strings. This approach helps with debugging a lot, but can slow down automatons processing when we compare values of variables. The other three sections are directly transformed into derivation trees. Classes used as tree nodes can be found in the `parsed` package. The provisions use different parse tree nodes than the reactions and the threads. The nodes are named according to the represented entities. In this step we limit the reentrancy, which means that an unlimited reentrancy operator is converted into the limited reentrancy one with a given limit.

The second step (*resolving stage*) is equivalent to semantic analysis in compilers. This step transforms the parsed derivation trees into resolved trees. In the parsed trees objects are identified by names (strings from the TBP specification). In the resolved trees the string

1 Distributed Systems Research Group

identifiers are replaced by references to created representatives. These representatives of the variables and types are created during the transformation into the resolved form. Various checks also take place during the transformation. The TBPLib checks whether used variables are defined in the variables section, whether the value assigned to the variable is permitted by the variable type etc. The parsed and resolved trees have similar shape. The classes used to represent nodes in the trees are stored in the `resolved` package. The division of nodes into the imperative and the provisions nodes is the same as in the first parse step.

The third step (*automaton generating stage*) is the most important step of checking. This step transforms the resolved derivation trees into the automatons. Only the provisions, reactions and threads are transformed. The transformation creates nondeterministic automatons, but the provisions automatons can be converted into deterministic ones. With the possible exponential state explosion, of course. The structure of the automatons will be described later in this chapter. The provisions are transformed one be one into automatons and they are stored in one hashmap under their names. The same applies to the thread automatons. The reaction is not saved under its name but under a `Binding`.

The *binding* presents a way how to solve problems with parameters. The reactions in the TBP may have parameters as procedures in other programming languages do. Note that the relation between the parameters of reactions and the parameters in the implementation is not defined. The binding contains a reference to a reaction (the method name) and it holds values of the parameters. One resolved tree that represents a reaction is transformed into many automatons. There is a binding for any combination of values of the parameters. (The cartesian product of sets with possible parameter values). The parameters in the automaton are replaced by constants according to the binding. That is possible because the TBP does not allow assigning values to the parameters.

The last transformation step available in the TBPLib is in-lining of the automatons, where the automatons for the threads are connected with automatons for the reactions.

The TBPLib is able to join components together into bigger ones. This functionality is helpful if we check compliance of the composed components with BP.

The TBPLib contains support for automatons exporting. This is very useful for debugging. The automatons can be exported into ".*dot*" files. These standard text oriented files can be displayed by *Graphviz* [5] among others.

Our work does not use the component composition or automatons in-lining. The environment generator uses the provisions section from the TBP specification in the resolved form. The checker tool uses the TBP protocol transformed into the automatons.

2.2.2 Automatons

The TBPLib uses self-made automatons. The source code related to the automatons can be found in the ltsa package. There are two types of automatons - the *provisions automatons* and the *reactions automatons*. Both types of automatons are *state transition systems*.

The states in both forms of automatons are the same. The state holds the list of outgoing edges and a flag to mark visited states. The provisions automatons may have multiple final states, the reactions ones contain exactly one final state.

The edges are the same for both types of automatons. The edges in different types of automatons hold different values. The edges are directed. They hold the source and target

states, so it is possible to go against the direction of the edge. The edge holds either a symbol or a generic object. The symbol is an identifier of the edge and it is used in the provisions automatons. The objects are used in the reactions automatons and they represent actions on the edges. The actions represent the statements from the reactions and the threads. Sometimes we call the edges according to the type of the action that the edge represents.

Possible action types are null, assignment, emit, undefined emit, condition and mutex enter.

The null edge type represents the edge with no associated action. That means that the component does not emit a call.

The assignment edge contains references to the variable and to the new value. The new value can be a reference to a constant, another variable, result of the last emit action or result of the function call directly. The last two possibilities look similarly, but the first one permits more internal actions between the function call and the assignment of the return value to the variable.

The emit and undefined emit edge types represent actions. The edges represent calls to the required interfaces. In the emit case, the reaction automaton that defines behavior of the called method is known. In the undefined emit case, which is the common one in our checker tool, there is no known automaton for the called method. In this case the return value is nondeterministically chosen from all possible values of the assigned type.

The return action represents return from the current method. The edge holds a reference to the return value.

The sync enter action is represented by a reference to the mutex type. The opposite sync exit action is represented by the standard assignment action where the mutex variable is set the UNLOCKED value.

The conditional edge types represent guards. The condition can permit or deny going through the edge. The if, switch and while statements in the TBP are converted to the conditional edges followed by the subautomatons for the branches. The condition has three base forms - always true, comparing on equality and test whether the value is in the given list. Each condition can be negated. The always true form is used to represent '?' (non-deterministic choice) in the TBP protocol. The equality form of the condition compares the value of the variable with a constant or it compares two constants. It is not possible to test whether values of two different variables are equal. The last possible condition type checks whether the value of the variable is in the given list of constants. This conditional form is used to represent the 'default' case in the switch statement (in negated form). The switch statement can be substituted by a sequence of if commands, but we can reduce the number of states in the created automaton if we have the value not in given list conditions. The switch statements can be represented by a single state, where there is a conditional edge for each case branch.

2.2.3 Resolved provisions

The environment generator uses the provisions in the parsed form. The environment is generated directly from this form. Each operator is represented by the node of the tree and its operands are represented as children of the node. The leaves of the tree are represented by the null and accept nodes. There are 5 types of internal nodes. They represent operators from the provisions section of the TBP specification.

2.3 Java PathFinder

Java PathFinder (JPF) [6][7] is an open source tool developed by NASA. JPF is intended to verify programs in Java. More precisely it is a model checker for Java bytecode. The JPF contains its own JVM that executes a program to in order to verify it. The JVM is optimized for observability of a program state. The JPF comes with standard example extensions that can verify whether a program is deadlock free, does not dereference null and some other low level properties. The JPF is highly configurable and extensible.

2.3.1 State space traversal

Standard JVMs execute only one program trace, they move only forward. Unlike standard JVMs the JVM integrated in the JPF is able to move forward and backward. Such behavior is essential for the checking purposes. Non-revertible actions are problematic for the backward moves. Such actions are for example the IO or network operations. The JVM in the JPF executes bytecode instructions of course. Steps in the JPF are represented by instruction bundles called transitions. Transitions present a higher level of abstraction created over the instructions.

Transition is a sequence of instructions. There is no context switch within a transition, all instruction in the transition are executed in the same thread. Multiple transitions can be leading out of one state. The concept of transitions is closely related to the partial order reduction (POR). The JPF employs on-the-fly POR to reduce the state space of the program. This reduction is on the safe side. A transition contains state changes that are not visible to the other threads. State change that is visible to the other threads is a scheduling choice point which ends the transition, so only one such change is possible.

State consists of 2 main components, the state of the JVM and the current and next choice generator. Choice generators are described later in the extensions sections. The JVM state contains a heap, stacks and program counters of all the threads.

This paragraph describes the behavior of the JPF while using the default depth first search strategy for traversing the state space. This strategy is implemented in the `DFSearch` class. Our checker tool uses the `DFSearch` and is closely related to it. The `DFSearch` executes a program in all possible ways. If you imagine state space as a tree (where the parent-child relation is represented by a forward JVM step) then `DFSearch` traverses such tree in depth first search manner. At first the JPF goes forward and executes transitions one by one until it reaches the end of the program or a state which has been reached (and processed) before. In this case the JPF stops the execution of the program and makes one step back (backtracks). Then the JPF makes a different decision (plans a different thread or generates a different value) and continues in forward processing. If all the decisions have been processed, then the JPF backtracks again into the previous state. Note that a stack is enough to store all possible states to which the JPF can backtrack.

2.3.2 Extensions

The JPF is a very good tool for research purposes because it is easily extensible. There are many ways how to extend the JPF. The most important extending mechanism in the JPF are *listeners*. Other mechanisms are *choice generators* or *model java interface* (MJI).

Choice generators produce next states from the current one. The choice generators are responsible for the management of unexplored transitions. They differ in a way how they traverse the state space. If we omit a few special search classes, that do not traverse the whole space state (typically they do not backtrack at all), then the choice generators differ in used heuristics. It means that for the same state they generate the same descendant states, but the order of descendant states is different. There are two possible choice types - scheduling choices and data choices. The *scheduling choices* are related to the synchronization and thread switching. The *data choices* are represented by "random" data values. The Java `Random` class or the `Verify` class from the JPF causes the data scheduling choices. Types of the choices and examples of code that invokes given choice are in figure 2.9.

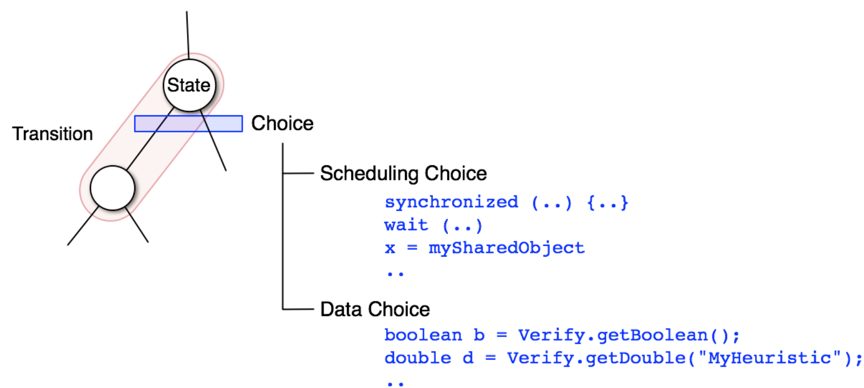

Figure 2.9: Choice examples [7]

The *MJI* is typically used to implement library classes. These classes are not checked and they are executed atomically. Such progress makes the state space smaller. It is necessary to create an interface, that will be visible to the program being checked, and then create an implementation of the interface. The implementation has to modify the internal JPF state. That means that we cannot simply use the standard library implementation of the interface. This progress is similar to the standardized Java Native Interface (JNI) in JVM. The MJI and JNI differ in the level of abstraction.

The *listeners* are based on an observer pattern. The JPF provides interfaces that notify the listener when an event takes place in the executed program. There are two types of listeners -VM listeners and search listeners.

The *VM listener* handles events related to the JVM and the executed program. There are lots of different events. The JPF notifies the listener, among others, if the JVM executes an instruction, if it creates a new thread, if an exception is thrown or if a thread exits. Other events are related to synchronization, initialization and loading of objects or garbage collecting.

The *search listener* handles events related to the state space traversing. The JPF notifies the listener if a new state is reached or if the JPF backtracks into the previous states.

The "*started*" notification is the first notification that the JPF sends. The event comes when the JPF starts execution of the program. The listeners typically initialize internal structures here.

The "*finished*" notification is the last notification that the JPF sends. The event comes when all traces are processed and all work is done. The listers typically calculate overall statistics, print results and release used resources here.

The "*advanced*" notification comes when the JPF has executed one transition and it has reached the next state. If the reached state is a new state then exploration of its descendants will occur later. If the reached state has been processed before then the exploration ends, and the backtracking takes place. The listeners typically test changes of the program state here and update their own internal state.

The "*processed*" notification takes place when all descendants of the current state have been explored.

The "*backtracked*" notification comes when the JPF makes a step back into the previous state. This event takes place after the "processed" event on the descendant state. The listeners typically restore internal state.

There are other notifications defined in the search listener interface, but they are not that important and our checker tool does not use them.

3 Checker

This chapter introduces the algorithm used for checking of Java code of a component against the TBP specification. It covers the description of the checker structure and the checking techniques. Description is done in an implementation independent way as much as possible.

The main task of the checker is to verify whether the implementation of a given primitive component in Java satisfies a TBP specification. We verify that a component behaves in a way described by the TBP for any allowed usage. Figure 3.1 illustrates how the checker tool works. The 2 steps - generating of an environment and running the JPF make the main and the most interesting part of the checking. Environment creation and its purpose

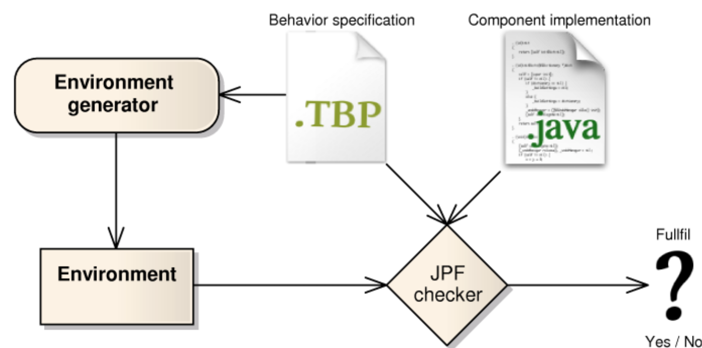

Figure 3.1: Checker structure

is described in the next chapter 3.1 in detail. The compilation is only the necessary technical step. The last step is running the JPF with our special protocol checker, that is special JPF listener. In comparison with other JPF listeners, which check relatively low level properties, our listener checks high level abstraction and it is more complicated. The listener observes the behavior of the component and picks interesting events. The events are then passed for further processing. They are processed in the same order as they take place in the program being executed. The events are separated by thread and type. Then the state of the checker is changed in accordance with the incoming events - representatives of new threads are created or position in the TBP is changed etc. An error is reported if an unexpected event (method call which is not permitted by the TBP) takes place.

3.1 Environment creation

Model checkers need closed systems to be able to check, however a single component is not that case. Closed systems are necessary by principle, because the checkers try to execute the tested system. The problem can be divided into two parts - missing main method and interaction with the environment. The way we generate the testing environment with a main method is described later in this chapter. Non-revertible interaction with environment (such as network communication, user input, ...) is a different kind of problem. Various approaches can be found in the model checkers - denying such behavior or user modeled expected interaction. We use the first approach - we suppose that components do not interact with the environment in any way. It is user's responsibility to remove such problematic parts

from the implementation of the component. It is not possible to remove these interactions automatically, because changing the component may break its semantics.

The test environment is generated from the provisions part of the TBP, because provisions encode the permitted use of the component. The environment can simulate all allowed uses of the component. The environment in cooperation with the JPF, that executes program in all possible ways, ensures, that the checker verifies all permitted uses of the component. The checker observes behavior of the component and verifies that the behavior is permitted by the TBP specification for each usage simulated by the behavior.

The provisions section is important for environment generation. We do not use any representation of the provisions section later at runtime. The provisions are encoded in the generated environment so that they affect the state space of the component that the JPF walks through. If the TBP specification contains more than one provision then the component is checked separately for each provision and has to pass all checks to comply with the specification.

Three basic problems have to be solved when an environment is generated:

- How do we obtain the values for method parameters?
- How do we establish relations between the generated environment and the provisions?
- How do we generate all possible traces of events?

3.1.1 Parameters of generated calls

The TBP specification is a high level abstraction of communication. It does not deal with parameters of methods used in the code. The type system and method parameters used in the TBP have no (mandatory) relation to the parameters in the source code. We need values to pass as actual parameters to be able to generate an environment that calls methods. All possible values of parameters should be used optimally. This approach leads to state explosion caused by data choices. The explicit state model checkers (as the JPF) cannot group states together, so it is necessary to examine all possible values one by one. We use more pragmatic approach. The user provides us with a database with values, that should be used as parameters. The generated environment tries all combinations of parameters.

3.1.2 Relation between environment and provisions

The relation between the generated environment and the provision is necessary to support parameter values in provisions. At runtime events like method calls do not contain information related to the parameters used in the TBP specification, but the information is necessary to invoke the reactions correctly.

There are a lot of ways how to store the relation. Information can be encoded e. g. in the annotations, in external tables, etc. We use the second approach. For each generated environment there is a hashtable, which associates the line of code, where the event takes place, to the actual event and its parameters in the provisions. This hashtable is used during the checking. Provided call events contain the line of code where the call takes place, the checker then finds the call parameters used in the provisions in the hashtable. Automaton for the called provided method is then selected according to the parameters.

3.1.3 Generating of the environment

Our task is to create a source code that will generate all call traces permitted by the provisions and will not generate any call traces which are not permitted by the provisions. The expected behavior is required to be fulfilled in cooperation with the JPF. We explain the behavior of the environment when it is executed in JPF using JPF's API for non-deterministic choice (`Verify`). The created environment relies on scheduling and data non-deterministic choices. The JPF systematically explores all possible non-deterministic choices thus it explores all call traces.

The previous paragraph describes the necessary condition. On the other hand generated environment should extend state space of the checked component as little as possible, because added extra states make checking more time and memory-consuming.

We describe how the code, that fulfills the above mentioned needs, looks like for each operator. Code generating is done in a recursive way. Substitute *A* or *B* by code generated by the corresponding operands in the code examples.

The s*equencing operator ';'* is the most simple case. We generate code for the second operator after the first one. The generated code for the *A ; B* expression is shown in figure 3.2.

```
A ; // Code for A operand
B   // Code for B operand
```

Figure 3.2: Sequence operator code

The a*lternative operator '+'* is converted into the if statement where condition is represented by the data choice. The JPF tries to execute both branches during the checking phase. Expression in form A + B is transformed into the code in figure 3.3.

```
if (Verify.getBoolean()) {
     A ; // Code for A operand
} else {
     B ; // Code for B operand
}
```

Figure 3.3: Alternative operator code

The *repetition operator '*'*. Repetition should be represented by the "while" cycle where condition is the same as in the alternative operator case and the body of the cycle contains code of the operand. This is the most precise approach. The JPF examines all possible repetition counts of the cycle. Generated code for *A ** expression is in figure 3.4. In

```
while (Verify.getBoolean()) {
     A ; // Code for A operand
}
```

Figure 3.4: Repetition operator code

our current implementation we unroll the "while" cycle into the "for" cycle where the maximum repetition count is limited by a constant. The checker tool currently uses 3 iterations of the "for" cycle. There are two reasons for our solution. It reduces the state

explosion, that can be caused by the extreme number of repetitions of the "while" cycle. On the other hand the loss of precision is not so significant because most of the errors are found in a few first iterations.

The *and-parallel operator* '|' operator is not as clear as the previous ones. The checker tool generates the code for each operand into a separate thread class. Both threads are started and the environment waits until both threads end. Interleaving of the events is generated by a scheduler, which plans the threads. Example of generated code is in figure 3.5.

```
Thread t1 = new Thread() {
     public void run() {
          A ; // Code for A operand
     }
};
Thread t2 = new Thread() {
     public void run() {
          B ; // Code for B operand
     }
};
t1.start(); t2.start();
t1.join(); t2.join();
```

Figure 3.5: And-parallel operator code

Note that the *(A ; B) | (X ; Y)* and *(A ; B ; X ; Y) + (A ; X ; B ; Y) + (A ; X ; Y ; B) + (X ; A ; B ; Y) + (X ; A ; Y ; B) + (X ; Y ; A ; B)* expressions as presented in chapter 2.1.1 are not interchangeable. Even though both expressions permit the same sequences of calls, which is important for the composition correctness verification, semantics of the expressions differ. The first expression *(A ; B) | (X ; Y)* requires to test parallel usage of the *(A ; B)* and *(X ; Y)*, we test all interleavings of instructions in the implementations of the methods. In the second case, no parallel usage is required, a single thread sequentially calls the methods of the expression. It is obvious that the second approach is much faster but only the first approach is able to test race conditions.

All other operators (or-parallel '||', unlimited '|?' and limited '|*n*' reentrancy) are syntactic sugar and can be expressed by the previous ones. More information on the implementation of the environment generator and the optimizations we have done in the generated environment can be found in chapter 4.2.

This paragraph describes the basic provisions - NULL and call. The *NULL* expression is converted into a comment. No executable code is generated. *Calls* are converted into a call of a method named after the protocol event on the component or on its provided interfaces. Values for parameters are obtained from a given value database (*envValues*) at runtime and the return value of the called method is ignored. Code example can be found in figure 3.6.

```
// Component comp = new Component()
// EnvValueSets envValues = ...
comp.Method(
     envValues.getInt(..));
```

Figure 3.6: Call expression code

3.2 TBP checking

This chapter describes the way how the checking is done. We focus on some basic ideas and then we will discuss selected technical topics in more detail.

Let us be reminded of our task. We have the TBP specification for a given component and an implementation of the component in the Java programming language. We want to ensure that the component behaves as described in the specification for each permitted use. In this context, the term "*behaves*" means that the component calls proper methods on required interfaces in the order permitted by the TBP specification. For the purposes of this chapter the term "*event*" means all calls and returns on the provided and required interfaces. This is our task described in the terms of our input, but the input (the TBP specification) has been processed by the TBPLib before it is used by the checker tool. The TBPLib transforms the reactions and the threads sections of the specification into nondeterministic automatons. These automatons have actions (like method calls, variable assignments, conditions, etc.) associated to edges. Automatons are described in more detail in chapter 2.2.2. All permitted uses are encoded into the generated environment.

Now we reformulate our task in terms of the input for the JPF checker plugin. We want to ensure that for each possible execution of our program, which consists of the component and the environment, the automatons accept events generated by the execution. The term "automatons accept" means there exists a path in the automaton from the start state to the final state such that the path generates the same events as the program. Such path in the automaton will be referred to as a *witness*.

The TBP specification contains two "orthogonal" approaches - automatons and variable evaluations. One automaton represents events that can be generated by one execution of the method. This means that for each method invocation (mainly on provided interface) we have to find an automaton associated with the called method and then try to follow events in this automaton to make the witness. If we have more then one thread, which means that each thread can invoke its own methods, then we need to separate automatons for each thread. To sum up, automatons are closely related to method invocations and they are separated by threads.

The "variable evaluation" term means such a mapping that to each variable defined in the variables section of the TBP protocol one of the permitted values is assigned. There can be more "*evaluations"*, which differ in the values assigned. The evaluation represents the internal state of the component. In contrast to the automatons evaluations exist from the start of the execution of the program being tested to the end of execution. Evaluations are global, so all events generated by the automatons use and modify the variables in parallel. In order to traverse the automaton we need to have an evaluation for each path (including the witness).

Our design decisions make the previously described procedure more difficult. We process more positions in the automatons in parallel.

In the following chapters we will cover the checking process and related topics in more detail. We will discuss how the automatons should be handled, consequences of parallel automatons processing, consequences of the way the JPF traverses the program state space, how to extend the state of the program with the state of the checker, how to handle threads, etc.

3.2.1 Automatons handling

In the preceding text we have discussed the automatons very briefly. How does the checker tool handle automatons? Positions in the automatons are independent for each thread and different threads may be processing different methods, so events from each thread may be handled by a different automaton. Automatons describing the reactions are non-deterministic. A pair of an automaton state (position) and an event may generate many automaton states, where the expected event takes place. The checker tool processes more states in parallel, therefore the checker tool does not store only one automaton state, but it stores a set of automaton states. Finding *the witness* in this case means finding at least one path in the automaton such that generated required events on the path correspond to events generated by the program. So for each thread we store a set of associated automaton states. The term "state *associated* to the thread" means that the state is stored in the set for a given thread. Parallel processing of the automaton states needs a special way how to handle *variable evaluations*. We cover this topic in more detail in section 3.2.2.

The JPF controls the way how to work with automatons. As it is mentioned in chapter 2.3.1, the JPF executes the program in the *transitions*, a transition is a list of executed instructions, instructions in the list may represent events (invoke/return). To leave things simple, imagine that the JPF makes only forward steps and no backtracking takes place. This situation corresponds to the execution of the program in normal JVM, from its start to its end. The JPF backtracking and the way how to support it is covered in more detail later in section 3.2.3.

What happens if the environment calls the component? This is the case of *a call to the provided interface*. In this situation the checker has to find an automaton to use. The automaton is found according to the interface and the method name and according to the values of the parameters in the provisions. There are no automaton states associated with the thread when provided call from the environment takes place, because the checker does not use any automatons to represent the generated environment. Finally the checker associates the start state of the automaton to the thread that generates the provided call event and updates

```
aut = foundAutomaton(interfaceName, methodName,
     methodParameters);
associateState(currentThread, aut.startState())
setMappingsAllStatesAsCall(currentThread, aut.startState());
```

Algorithm 1: Provided call

variable mappings. The pseudo code in Algorithm 1 shows main operations that handle provided method calls.

During processing of the provided call, the component typically invokes methods on the required interfaces. What should the checker do in case of *a call to the required interface*? At the time of the call automaton states associated to the calling thread exist. The checker needs to update the automaton states to reflect the new call. The checker takes all the associated automaton states and processes them one by one. While it processes the state the checker tries to find all possible paths in the automaton from the old state to the new states that represent the call of the expected required method, and associate all found states with the thread. Algorithm 2 shows pseudo code that handles required method calls.

```
oldStates = getAssociatedState(currentThread);
clearAssociatedStates(currentThread);
for each(oldState in oldStates) {
     newstates = foundNewStates(oldState,
          interfaceName, methodName,
          methodPatameter);
     associateStates(currentThread, newStates);
}
if (!isAnyStateAssociated(currentThread)) {
     report_error();
}
```

Algorithm 2: Required call

How does the checker search for the *new states*? It traverses the graph created by edges and states of the automaton. There is a lot of ways how to traverse the graph - depth-first search, breath-first search, heuristics, etc. The way used to traverse the graph is not important as long as the following rules hold.

- All paths in the automaton are processed
- Path contains at exactly one call. The call event ends the path. The called method is the expected one. The paths leading to the call of a different method are ignored.
- If a path contains a guard, then the condition has to hold. If the condition does not hold the path has to be skipped.
- If a path contains an enter into a synchronized section, then the mutex used to protect that section has to be unlocked.
- Path contains one state only once. This condition protects against cycles in the automatons such that no method call event takes place on them. Such cycles can generate infinite number of paths.

Figure 3.8 shows the automaton generated from the provision 3.7. We use the automaton to illustrate generated paths in case of the provided call to the "lookupStockCard" method. The states associated to the thread before the call are yellow filled (fully/partially). They are the start states of the paths. The dashed lines represent the paths being generated. The red paths are ignored, because they do not point to

```
01 : IShopMgmt.reserveItem() {
02 :   !IInventory.getWarehouses() ;
03 :   while(?) {
04 :     !IWarehouse.lookupStockCard()
05 :     /* Warehouse doesn't contain the item */
06 :   } ;
07 :   if (?) {
08 :     sync(itemReservation) {
09 :       !IWarehouse.lookupStockCard() ;
10 :       /* Warehouse contains the item */
11 :       !IWarehouse.blockStock()
12 :     }
13 :   }
14 : }
```

Figure 3.7: Reaction used to generate the automaton

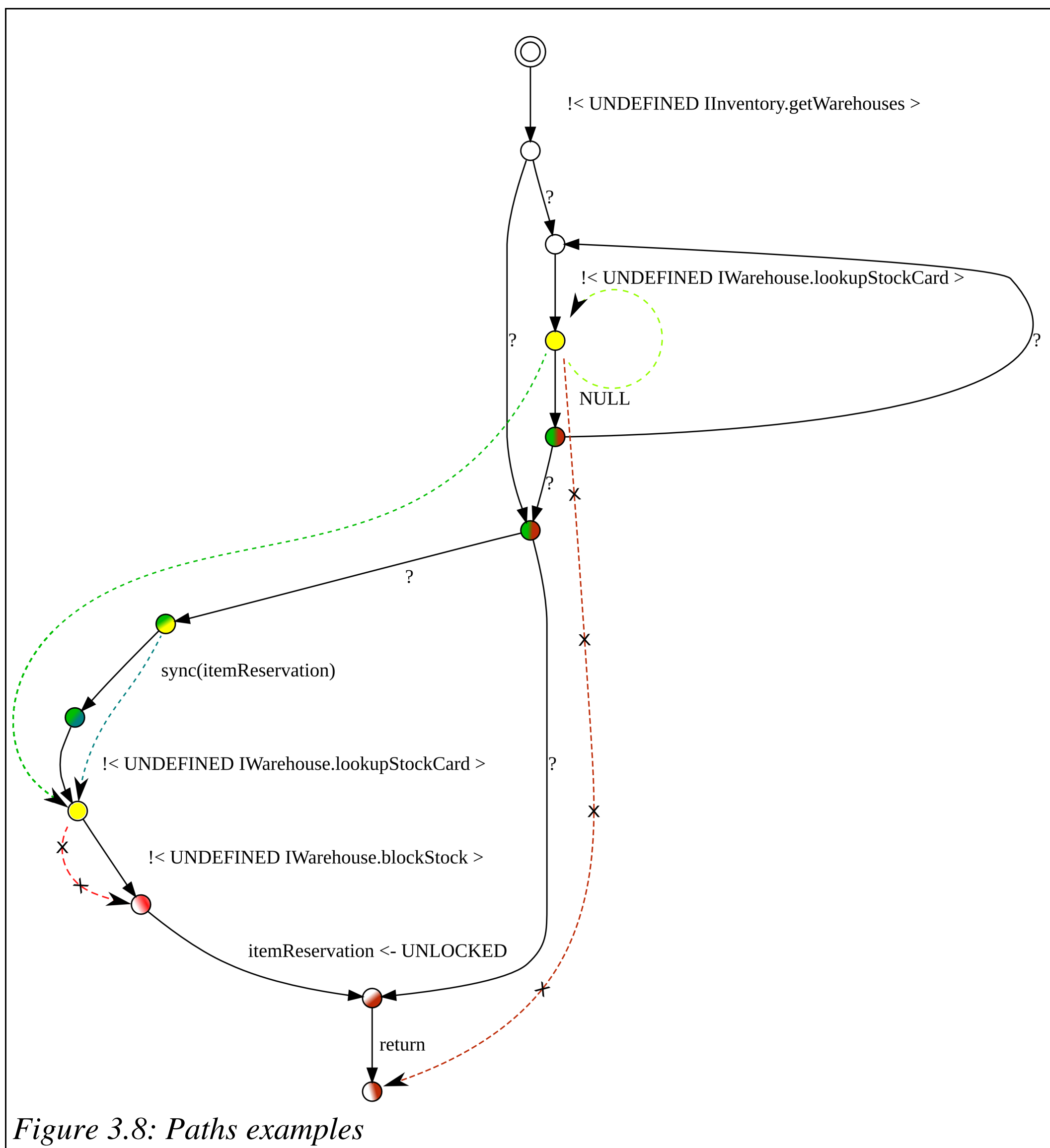

Figure 3.8: Paths examples

the "lookupStockCard" calls. The green and blue paths are correct and the target states of these paths will be associated to the thread after the provided call event.

If no state is associated with the current thread after all old states have been processed, then we have *found an error*. There is no path in the automaton which can produce the current trace. Last event cannot be generated by the automatons. That means that we have found a *bad activity* error. If we think about the path in the automaton, we have to consider all previously processed events. They may affect used variable evaluations which may disable some edges (guards). The disabled edges may cause that the current event cannot be generated.

When the processing of called method on the required interface is done, the return event takes place. What does the checker do in case of *a return on the required interface*? If we consider only the generated environment, then the situation is a bit easier. We know the behavior of the called required method, because the method is generated by the environment generator. The checker has to ensure that the return event is associated to the call event, that means that return is on the same method and in the same interface as the call event. If we focus on the communication in real component systems and not only on the single component, then the situation complicates a bit. The called required method is not generated, thus implementation of the called required method can invoke other calls. In this case the checker tool should observe these calls, stack these events to be able to recognize the return event related to our required call. The checker traverses the graph after a successful test of return event correctness. But the rules for traversing are modified a bit, paths leading to any method call or return event are considered. The automaton state is associated to the thread if call, mutex enter or guard edge is reached.

The last event type is *a return on the provided interface*. This event takes place when the provided method called from the environment has done all its work. After this return event the environment typically generates a new provided call. What should the checker do? The checker has to find whether the return event can take place. So the checker processes all associated states to see if there is a path to the final state. The search has to conform to the same rules as in the case of the call to the required interface. If no such path exists error has to be reported. The TBP protocol expects some other events before the return event, so *no activity* errors are captured here.

If the tested program reaches the end then no special action is necessary in this case. The component currently does not process any method, so there are no automatons to checks.

The previous paragraphs describes the situation for the provisions. Now we describe how threads are handled from the automatons point of view. The thread means a thread created by the component which is described in the special section of the TBP protocol. In general, the automatons which describe behavior of the thread are the same as the automatons used in the reaction section. But there are some differences as to the event handling. Processing of the required method calls and returns is the same as in the reaction case which has been discussed before. But processing of the provided method calls is different and is more similar to the approach used for the required methods. For the events generated by the environment there are no automatons used to check events. In contrast to the environment,

threads are described in the TBP specification and automatons define possible call sequences which the threads could generate.

When a new thread is created by the component then the automaton for the thread is selected. This event has to be handled similarly as the provided call in the reaction. Thread can call methods on the required interface only. Calls on the provided interfaces are prohibited. When a thread ends then a check if a final state is reachable takes place. It is a similar situation as the return from the provided method in the reactions.

Until now we have not analyzed the situation when a method from the provided interface calls another method on the provided interface. We will focus on how to handle this situation from the automatons point of view. The checker tool has to remember automaton states in the calling method and associate the start state of the called method to the current thread. After the return from the called method, the checker associates previously saved states from the calling method with the current thread and the processing continues from the saved state. This idea leads to a stack that copies the stack of the current thread. Called automaton may return value, so the checker has to track the relation between the original state from the calling automaton and the new states from the called method. This relation is necessary to be able to handle the return events. Different return values may be stored into the variables and the checker needs to know which value to use with which state in the calling automaton. This is also useful in a situation described in the following paragraph. Notice that the TBP specification does not support when the method from the provided interface calls another provided method. Current implementation reports an error if such behavior takes place.

Automatons for required methods may present the same problem with the same solution. These automatons have to conform to the behavior of the generated stub implementation. Stubs do not call any other methods, so these automatons cannot generate any events, but can be useful for the modification of the TBP variables (modification of internal state). The TBP checker supports such automatons.

3.2.2 Variables evaluation

In the previous section we have introduced the terms *witness* and *evaluations*. Now we will cover the handling of evaluations in more detail. A witness consists not only from a path in the automatons but from the evaluation on which the path operates.

To be able to interpret the path in an automaton the checker needs context. This context is represented by an evaluation. Evaluation is necessary to be able to decide whether condition (guard) holds, thus the edge is enabled and a path can go through the given edge. Processing of the assignment action needs the evaluation to be able to execute the action and change the evaluation. Internal state of a mutex is also stored in evaluations. Evaluation is not constant, it changes during traversing the automatons.

Evaluation should represent an internal state of the whole component. It is obvious that this state is global and shared among all threads. At first the checker uses a mapping that uses the initial values of the variables section of the TBP specification. While events in the automatons are handled, events of only a single thread are considered and interleaving with another threads events is ignored. In contrast to the automaton handling, all threads change the evaluations (internal state) in parallel, thus interleaving of the events from

different threads is important. Term "in parallel" means the order in which the events appear on the trace. We may say that in automatons we ignore interleaving events of different threads, but in reality they affect each other by global context (by evaluations).

We have mentioned that an evaluation is required to be able to traverse an automaton. If we have a witness, we have one evaluation on which the path through the automatons operates. But when the checker tries to find the witness it explores all paths, because it is not known which path is the correct one. For each path it is necessary to have a different evaluation, because each path may contain different internal state changes, different assignments.

Paths are generated iteratively, which means that the checker extends the paths when a new event occurs. At runtime each path is represented only by the last automaton state on the path. If we know only the final state it is not possible to reconstruct the witness, but for our purposes only the existence of the witness is important and not its actual instance (exact path through automatons). In this case the evaluation is assigned to the automaton state. We will discuss mapping of the automaton states to the evaluations later in this chapter.

In this paragraph we consider the situation in one automaton only. Because of parallel processing of multiple paths the checker has a set of automaton states and it maintains the evaluation for each state. Imagine that a new event comes, the checker processes each such pair (automaton state and evaluation) one by one. The checker tries to find all succeeding new automaton states according to the rules defined in the previous chapter 3.2.1. While it searches for new states the checker modifies the original evaluation according to the actions on the edges. If there is more then one succeeding new automaton state then the checker divides one evaluation into more separate instances of the evaluation. Note that it is not enough to find only one path from the start state of the automaton to the end state. It is necessary to process all the paths that lead to the final state, because they may create different evaluations. The checker does not know which one from these evaluations will be needed later, because the paths in the automatons are influenced by the evaluations. So we need to have all feasible evaluations to capture all permitted uses.

In the previous paragraph we have illustrated the situation from a one thread view (from automaton point of view). This paragraph will show a global view on the evaluations. We use similarity with the witness. The witness has the one evaluation to which all the threads made changes. That means that the automaton states on the path for each thread are associated with the evaluation (share the evaluation) and changes to this evaluation are immediately visible to the other threads. So the evaluation has one automaton state mapped to each thread at any time. Note that if a thread does not have any automaton that is currently processed (thread does not process any provided or required call) then no state is associated, otherwise there is exactly one state for the thread. The *mapping* should be unique, there is no reason to handle two same mappings because the result will be the same for both of the mappings. This has the following consequence - for one thread, automaton state and evaluation pairs are not necessary unique, the difference can be in mapped states for other threads. Current implementation processes identical pairs one by one, even if it is obvious that for a given thread they discover the same automaton states with the same evaluation, but these

evaluations will have different mappings. There is place for optimization, we could possibly group such pairs together and process as one bundle.

In this section we will discuss how different events affect *mappings* on evaluations. If a *call to the provided interface* takes place then no evaluation has been mapped yet to any automaton state by the thread which caused the call. In this case we update the mappings of all evaluations in such a way that for a given thread they point to the first automaton state.

If a *call to the required interface* takes place then the thread, which generates the call, is processed. In this case the checker searches for all possible succeeding states in the automaton. Searching involves modification of evaluations and changes to the mappings. As we have mentioned before assignments modify the evaluations and this change is visible to the other threads. There are three possibilities when searching for succeeding automaton states - no state is found, one state is found, two or more states are found.

In the first case there is no path for a given pair of automaton state and evaluation in the automaton, thus the given pair cannot act as a witness. In this case the evaluation is discarded. This affects other threads and their sets of automaton states, because we remove the evaluation mapped to the automaton state. In the worst case it is the last evaluation mapped to the automaton state and in this case the automaton state may be removed from the set of states of the affected thread.

In the second case if only one succeeding state is found we may reuse the updated original evaluation, but we have to update the mapping. Only mapping for the thread which caused the call event is changed. The old mapped automaton state from which the checker started traversing the automaton, is replaced with the new one that the checker found.

In the last case, when the checker finds more succeeding states, it duplicates the evaluation, so that the checker creates a new evaluation for each found succeeding state. This evaluation is changed according to assignments on a path from the old automaton state to the new one. Mappings for duplicated evaluations are the same except for the thread that caused the event. Mapping for the thread that is responsible for the event is changed in the same manner as in the second case, the old automaton state is replaced by the new one where the path ends.

No action is related to the evaluations when a *return on the required interface* takes place.

Processing the *return on the provided interface* action is similar to the situation when the call to the provided interface takes place. The checker searches for succeeding states but in this case for return (not call) states only. The checker needs to store the return value if any, this will be discussed later in this chapter. And finally the mappings of all evaluations are updated. In this case the situation is complementary to the provided call, the checker leaves the automaton and no automaton is assigned to the thread which caused the return event. Mappings of all evaluations are modified so that no automaton state is assigned to the thread which is responsible for the return event.

3.2.2.1 Example

Following example 3.9 illustrates the idea of variables evaluation to automaton states mapping. The component being checked has two threads, each of them executes a different provided method, thus each thread has associated states from a different automaton. The filled

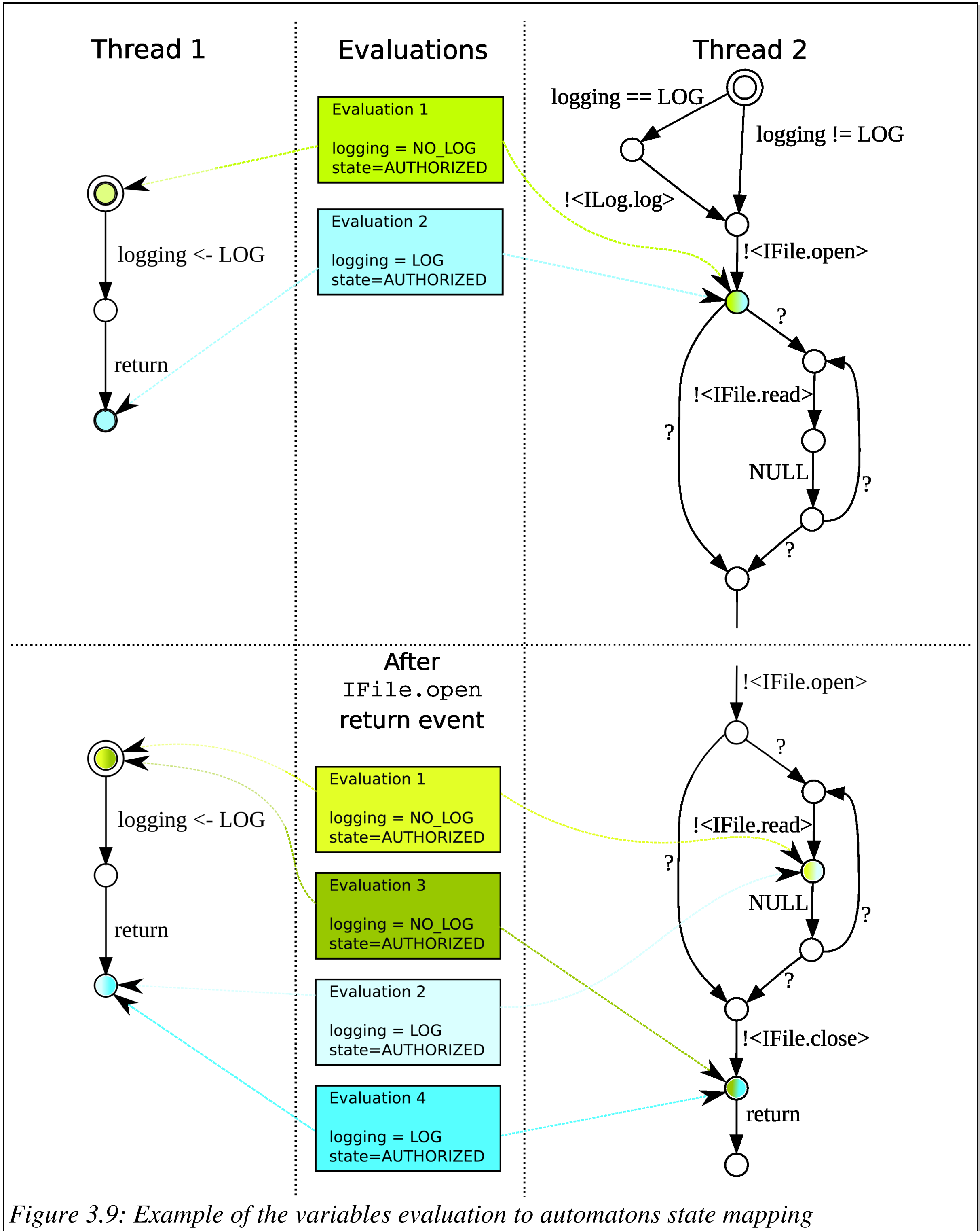

Figure 3.9: Example of the variables evaluation to automatons state mapping

automaton states represent the states associated to given thread. In the middle column all evaluations and their mapping to automaton states is depicted. The TBP specification defines two variables "logging" and "state". The top of the picture shows the situation before return from the required "IFile.open()" method in the "Thread 2", the bottom depicts the situation after the event is handled.

The checker searches for succeeding automaton states during the return event processing. It choses one pair from the automaton states (the marked state after "IFile.open" edge) and the evaluations (let us say "Evaluation 1"). The checker traverses the automaton and searches for next events. It chooses the right outgoing edge (without limiting the generality) at first and finds the state after the "IFile.read" edge. Then the checker maps this automaton state to the "Evaluation 1". The evaluation 1 is duplicated into the "Evaluation 3" (values of the variables and associated automaton state for Thread 1 are the same) and the checker continues in automaton traversal. It examines the left outgoing edge of the start state and finds the state after the "IFile.close" edge. It associates the "Evaluation 3" with the state. There are no more paths to process.

Then the checker processes the second pair (the marked state after "IFile.open" edge and the "Evaluation 2") in the same manner.

We have illustrated the way how to handle the return from the required method, we show the mapping of the variable evaluation to automaton states and we have presented the reason to duplicate the variables evaluations.

3.2.3 Checker state and backtracking

This section describes how the checker cooperates with the JPF in traversing the state space. The checker has got states. The automaton states and variable evaluations were covered in the previous section. Now let us move to the higher level of abstraction. This chapter focuses on the states of the entire checker. The *checker state* is composed of two mappings. The first mapping holds a set of automaton states for each thread. The second mapping associates variable evaluations to automaton states and it was described in the previous chapter 3.2.2 (the global view paragraph). In this section the term state refers to the whole checker state.

The way how the JPF traverses the state space is described in chapter 2.3.1. It is important to know that the JPF works in steps. It makes forward and backward steps. Processing of the forward steps was described in previous sections - the checker observes events and changes the states accordingly. The backward step takes back the actions of the previous forward step. The backward step should restore the checker state to the state as it was before the actual forward step.

There are two different approaches how to restore the checker state. We can *revert actions* of the last step or *store the last checker state* and then load it.

Revertible actions are essential for the first approach, but this is not our case. The changes the checker has made are not revertible at all. Used automatons are not deterministic even in the original direction nor when the direction of edges in automatons is reverted. We have not discovered any usable way how to solve this problem with action payback (revertibility). This method requires less memory than the second approach and time requirements will be probably similar for both approaches.

The "store checker state" approach is straightforward. The checker makes an unmodifiable snapshot of the checker state after each JPF step. These snapshots are stacked. When backward step takes place the checker pops the last saved snapshot and starts

using it. This approach needs a lot of memory to save the snapshots. On the other hand the creation and restoring of the snapshot is relatively fast.

The checker uses the second approach – states storing. It is necessary to ensure that stored states cannot be modified in the future. We fight the modification in two different ways. The checker uses immutability as much as possible. If the immutability concept is too expensive due to too frequent changes the other approach is applied. In this case the checker uses mutable classes and makes copies if needed. The checker uses optimizations on mutable instances to reduce copying. The checker counts the JPF steps. Mutable classes store a *time of creation*. The *time of creation* is the JPF step when an instance has been created. Instance can be modified only in the JPF step it was created in. In all succeeding JPF steps they are logically considered immutable. Therefore the checker creates a new copy before any change in the newer JPF steps and changes are applied onto this copy. The copied instance uses the actual time of creation of course.

3.2.4 Program state extension

In this section we focus on the extension of the program state [8]. We describe why it is essential and how can the extension be done.

If a program reaches the same program state again (state that has been processed before) the JPF stops testing the current branch, backtracks and tests a new execution path. This behavior is necessary to process loops. It reduces needless computations too - all possible paths from the given state were tested before.

Standard program state, as it is recognized by the JPF tool, consists of the stacks of all threads, static areas and the heap. But for our purposes the checker needs to extend the program state with position in the TBP specification. The position in the TBP specification is encoded in the checker state.

The tested program always generates the same traces of events from the same program state. But the program state does not include previous actions - the history of method calls, so even if the program reaches the same state again, previous actions may differ. Position in the TBP comprises of previous events (the history of calls). Therefore even if the program reaches the same program state again the checker can be in a different checker state. It is a one to many mapping. The behavior of the program can be permitted by some position in the TBP protocol, but it may not be permitted by another protocol position. Thus if the JPF ignores the checker state and always backtracks when the same program state was reached again, then the checking does not work correctly and the checker have may missed real behavior errors.

The following example illustrates the problem. Consider java code fragment in figure 3.10 and the corresponding TBP reaction from figure 3.11. Looking at the source code it is obvious that the variable '*b*' always holds *true* at line 8, so the program state is the same after the execution of each 'if' branch. If the JPF checks such piece of code it traverses one branch of the 'if' statement first (consider b is true), then processes both *mC* and *mD* calls. After certain time the JPF backtracks and tries to process the second branch of the 'if′ statement (the 'else' branch in the program), but then the JPF visits previously explored program state (line 8) and stops the processing and backtracks again. This is a problem because the checker

```
01 : void m1() {
02 :     Boolean b = Verify.randomBool();
03 :     if (b) {
04 :         ri.mA();
05 :     } else {
06 :         ri.mB(); b = true;
07 :     }
08 :     ri.mC();
09 :     ri.mD();
10 : }
```

Figure 3.10: Java code fragment - The State extension problem

```
01: reactions {
02:    p1.m1() {
03:        if (?) { !ri.mA() ; !ri.mC() ; !ri.mD() }
04:        else   { !ri.mB() ; !ri.mC() ; !ri.mE() }
05:    }
```

Figure 3.11: TBP Fragment - State extension problem

is in a different position in the TBP protocol. When the JPF has reached the 8th line for the first time the checker position was in the true branch of the reaction (line 3). If the JPF reaches the 8th line for the second time the checker position is in the else branch of the reaction (line 4). We do not know if the new protocol position permits events generated by the skipped piece of code. It is necessary to check lines 8 and 9 against the new TBP protocol position (in the else branch of the TBP reaction). In our example the new protocol position permits different calls *mC* and *mE,* so the implementation violates the specification. But the standard JPF without state space extension will miss this problem. That is why we have to the extend the JPF state.

The extension brings two problems. The first problem is the way how to extend the JPF state to contain the checker state. Fast and easy comparison of the checker states is the second problem.

There are different ways to extend the JPF with the checker state - integration of the checker state into the program state or coordination of backtracking of the JPF.

The integration of the checker states involves important changes in the JPF internals. There is no interface prepared for extending the program state in the JPF. Therefore these specific changes will not be integrated into the JPF core, and further JPF versions may break them. Extending the JPF in this way will introduce complex changes.

The backtracking coordination approach includes creation of our own JPF `Search` class. It means that the user can not specify his own search algorithm and has to use the provided one. The class will virtually extend the program state. The search class has to maintain a mapping where for each program state there is a set of the processed checker states stored. If the JPF decides to continue in forward steps then the search class updates the mapping. The class adds current checker state to the set of processed states for the actual

program state. If the JPF wants to backtrack, because it reaches a previously processed program state, our class determines whether the actual checker state has been processed by the current program state. If the checker state has been processed then the search class permits to backtrack, otherwise the aforementioned problem takes place and the search class instructs the JPF not to backtrack and continue in forward steps.

During the design stage we think about 2 possibilities to compare states - snapshot comparison and events comparison.

The *snapshot comparison* approach is precise. In this case it is necessary to store all snapshots that the program generates during its execution. The snapshots consume a lot of memory and they are composed of a lot of small objects, so the comparison is slow. This approach can be improved by hashing. It is possible to precalculate hash code from the snapshot content. Hash code can be used to filter out different snapshots. If we abandon the precision (leave safe side), we can store only hash codes to represent the processed checker states. This brings fast comparison with lower memory consumption. The JPF tool uses the same approach (hashing) to remember program state.

The second *event comparison* approach is based on the idea that the checker state is determined by processed events. Thus the checker state is represented by a list of all events generated by the current trace. Two states are equal if and only if both lists of all events are the same. This approach is imprecise because different events may generate the same checker state. In this case we miss the opportunity to optimize and the checker tests the same trace again even though it is useless work that gives the same result as the previous check. But we are still on the safe side. This approach is relatively simple to implement and consumes smaller amount of memory. Imprecision causes problems with cycles. Each cycle iteration produces new longer event trace thus it is necessary to bind the number of repetitions in the '*' operator.

We have implemented all the approaches mentioned here - snapshot, hashcodes of snapshots and the event list comparison approach. The checker uses hash codes of snapshots by default.

3.2.5 Handling of threads

In the previous sections we have partially touched the threads. In this section we summarize these information and cover the given topic in more detail. In previous versions of BP there was no explicit support for threads, thus older BP checkers need not solve this problem.

When a new thread is started a representative for the thread is created. This representative exists until the thread is terminated. The representative handles all events of the thread and it holds all automaton states of the thread. It also reports errors in case of an unexpected event or an unexpected thread end. All representatives should be destroyed when the program terminates.

3.2.5.1 Thread type identification

There are many possible sources of threads in the tested program. The checker has to be careful because it does not have any guaranties about the code provided by the user. Thread may be created

1. by the component in testing when the component is created or started (TBP threads)
2. by the component in testing during reaction to any provided event (no event threads)
3. by the environment when it simulates the use of component (environment threads)
4. by the environment when the checker prepares sets of parameters, etc. (no event threads)
5. by a previously created thread (thread type inheritance)

We recognize different thread types according to the aforementioned creators of threads - *TBP threads*, *environment threads* and *no event threads*. Thread type can be easily recognized by the stack of the base (creator) thread.

The environment threads are created by the generated environment thus context of the stack is well defined. There can only be a call of the 'main' method followed by a call of a method with environment on the stack. Second possibility to create an environment thread comes with thread type inheritance, in this case the stack contains only the 'run' method and the base thread is also the environment thread.

It is a bit complicated to recognize TBP threads. The context of the stack is not so clear as in the previous case. The thread is considered a TBP thread if the stack of the base thread contains a call of the constructor of given component, which is not followed by any method call to provided or required interface. Also threads created by TBP threads while they are not in any provided or required method are considered to be TBP threads.

All other threads are the no event threads. Notice that provided methods must not create any threads. This limitation is important to limit the reentrancy.

Different thread types need different handling. If no event thread generated an event then an error is reported. The environment threads call provided methods first and then the required method calls are permitted. Handling of the TBP threads is a bit complicated. When a new TBP thread is created we assign the start states of the automatons representing the thread section to the TBP thread representative. If the TBP thread terminates we check whether we have reached the final state or not.

3.2.5.2 Mappings threads onto specification

The TBP specification describes the threads, which the component creates, and their behavior. On the other hand there is not enough information to precisely match the threads from the specification (TBP thread) onto the threads from the implementation (java thread). If a component creates a new java thread there is no information to specify which TBP thread from specification to use.

If we analyze this topic we find three possible solutions - annotations, brutal force search and nondeterministic automaton choice.

Currently, the source code *annotations* are heavily used to define meta-data. We can add missing mapping into annotations of the java thread classes. This approach requires

support from programmers, they have to create the annotations. That is why we do not support this approach.

The second way is to try all possible mappings - the *brutal force* search. The idea is relatively simple on the other hand the mapping counts grow exponentially. It might cause state space explosion and elongate the time of checking.

The third way was presented in the previous section. We map the java thread with all TBP threads and check if there is any that describes its behavior. In comparison with previous two precise methods this approach may not detect all errors. We do not check whether there is exactly one java thread for each TBP thread. Thus some TBP threads may not exist in the implementation. Also there may be more java threads for one TBP thread. On the other hand, this approach is fast and it does not contribute to state space explosion.

3.2.6 Synchronization

In this chapter we focus on the mutex type and the sync statement (see figure 2.5). In the original version of BP there was not support for synchronization or internal actions. The sync statement is suitable for the *critical section*. Other synchronization primitives (like *rendezvous*) have to be emulated.

There is a big difference in handling the synchronization elements (*semaphores*, critical sections, rw-locks) in programming languages and in the TBP. In the programming languages synchronization affects how the threads are planned, they modify the behavior of the program. On the other hand the TBP checker only passively observes the behavior of the program. The checker has to analyze events and ensure that there are not two threads in the sync sections at the same time associated with the same mutex. This requirement is fulfilled by the way the checker handles mutexes and synchronized statements.

Mutexes are implemented as standard variables with two states - LOCKED and UNLOCKED. The locked state means that a thread which is in the sync section exists. In the automatons there are two edge types related to the sync statement. The first edge type is related to the enter into the synchronized section, the second one represents the end of the sync statement. The *sync enter* edge is considered to be a guard and an assignment in one statement. The guard checks whether the given mutex is not in the locked state. While traversing the automatons, we can walk through the sync enter edge only if the mutex is in the unlocked state. The checker sets the mutex into the locked state if it walks through the sync enter edge. The *sync exit* edge sets the mutex into the unlocked state.

4 Implementation

If you have read the third chapter, you should be familiar with internal processes such as events handling, moving in the automatons, mapping variables evaluations to automaton states. This chapter deals with implementation details. It connects the implementation to the idea of the 3rd chapter.

4.1 Checker overview

In this section we briefly describe the checker implementation. It should help you to understand the checker source files. This chapter is a supplement to the programming documentation. It outlines the purpose of checker's classes and their mapping to previously mentioned terms and problems.

All classes are stored in the `org.ow2.dsrg.fm.tbpjava` package. The `Checker` class contains the `main` method. It reads the configuration file then calls the TBPLib to process the specification, asks the environment generator to create an environment. It runs the JPF with protocol checker to finally test the component.

The protocol checker is stored in the `checker` subpackage. The checker is connected to the JPF by a listener - the `ProtocolListener` class. The listener checks 2 types of events. First type is related to threads, second one to provided and required method calls. Thread creation and thread end events fall into the first type. When a new thread is created the listener determines its type (according 3.2.5) and run-time representation of the thread is created. The listener tests whether the automaton for the given thread is in the final state when a thread end event occurs. The listener extracts all method calls and returns from executed instructions and then filters them. Only the interesting events (on provided or required interfaces) remain. Instances of the `EventItem` class are created for filtered calls and returns as a run-time representation of the event. Both event types (thread and method calls) are passed to the `EventParser` for further processing.

The `EventItem` class represents the method calls or the return events. Instances are immutable and hold necessary information to identify the event.

The `EventParser` class processes the events that come from the `ProtocolListener`. This class has two important aims. It separates incoming events by thread number and event type. Its second role is handling the JPF backtracking. This is done be making “snapshots” which are precisely described in chapter 3.2.3. Snapshots are stored in the `state` property. The `EventParser` passes the events for further processing into the proper `ThreadAutomatons` class.

Positions in the automatons for single thread are stored in the `ThreadAutomatons` class. The position in the automaton is represented by the `AutomatonState` class. Positions are stored in a set. The `ThreadAutomatons` classes for one thread form a stack. (They are connected into a linked list). The way how the stack is handled and its purpose are mentioned at the end of chapter 3.2.1.

The `AutomatonState` represents one state in an automaton. Instances are stored in the `ThreadAutomatons` class. The class holds a `State` from the automaton, an `Edge`, that

has been used to enter the state, the automaton, where the state and the edge take place, and a flag, whether action on the edge has been processed. The AutomatonState class is an immutable.

Traversal of the automatons is implemented in the AutomatonMoves class. This class is used by the ThreadAutomatons class, which provides the automaton states where the traversing starts. The AutomatonMoves class generates all reachable automatons states from a given set of start states. The generated states are filtered later. We filter out calls and returns unrelated to the current event. The way new automaton states are generated describes chapter 3.2.1 in more detail.

Values of all variables defined in the TBP specification are stored in the VariablesEvaluation class. The class is immutable. The most important methods are getVariableValue and getVariableType. You can use setVariableValue to change the value of specified variables. This method does not change the current instance, but creates a new one, where value for the variable is changed accordingly. The setVariableValue method cooperates with the StatesMapper, thus the created evaluation has the same automaton states associated as the original one.

The StatesMapper class is used to map the AutomatonStates and the VariablesEvaluations. The class forwards the mapping request into the current StatesMapperLayer instance. Main purpose of this class is to handle the backtracking of the JPF. This class makes clones of the StatesMapperLayer, which holds evaluations to states mapping as "snapshots". The "snaphots" are stored in the stack, which is synchronized with the one used in the EventParser. The EventParser does similar work for the AutomatonStates to threads mapping. Chapter 3.2.3 covers the "snapshots" in more detail.

The StatesMapperLayer class maintains the mapping of AutomatonStates to VariablesEvaluations. The way how the checker tool manages the mapping is described in chapter 3.2.2.

Each VariablesEvaluation and the associated AutomatonStates can be in two modes - mapped and processing. The *mapped mode* is intended for mappings that were processed and will not be modified later during processing of the current event. All mappings should be in the mapped mode when the event is processed. The *processing mode* is intended for mappings which are planned to be processed by the AutomatonMoves class. The mappings in the processing mode may change. The class holds the mapped mode mappings in a Set and processed mode in a HashMap. The mappedState set holds all mappings in the mapped mode. The StatesMapperLayer class contains two indexes over the mapped mode mappings (the automatons2evaluation and the evaluation2automatons maps) to speed up the access to the mappings. They hold mappings in both directions. The processingState map is an equivalent of the evaluation2automatons map but it holds the mappings in the processing mode.

The StatesMapperLayer class handles return values as well. Return value can be assigned to any AutomatonState, if the return event is processed, return values are passed to the parent AutomatonStates as a return value of the last call. The return value and the AutomatonState are stored together in the ThreadStateStack class.

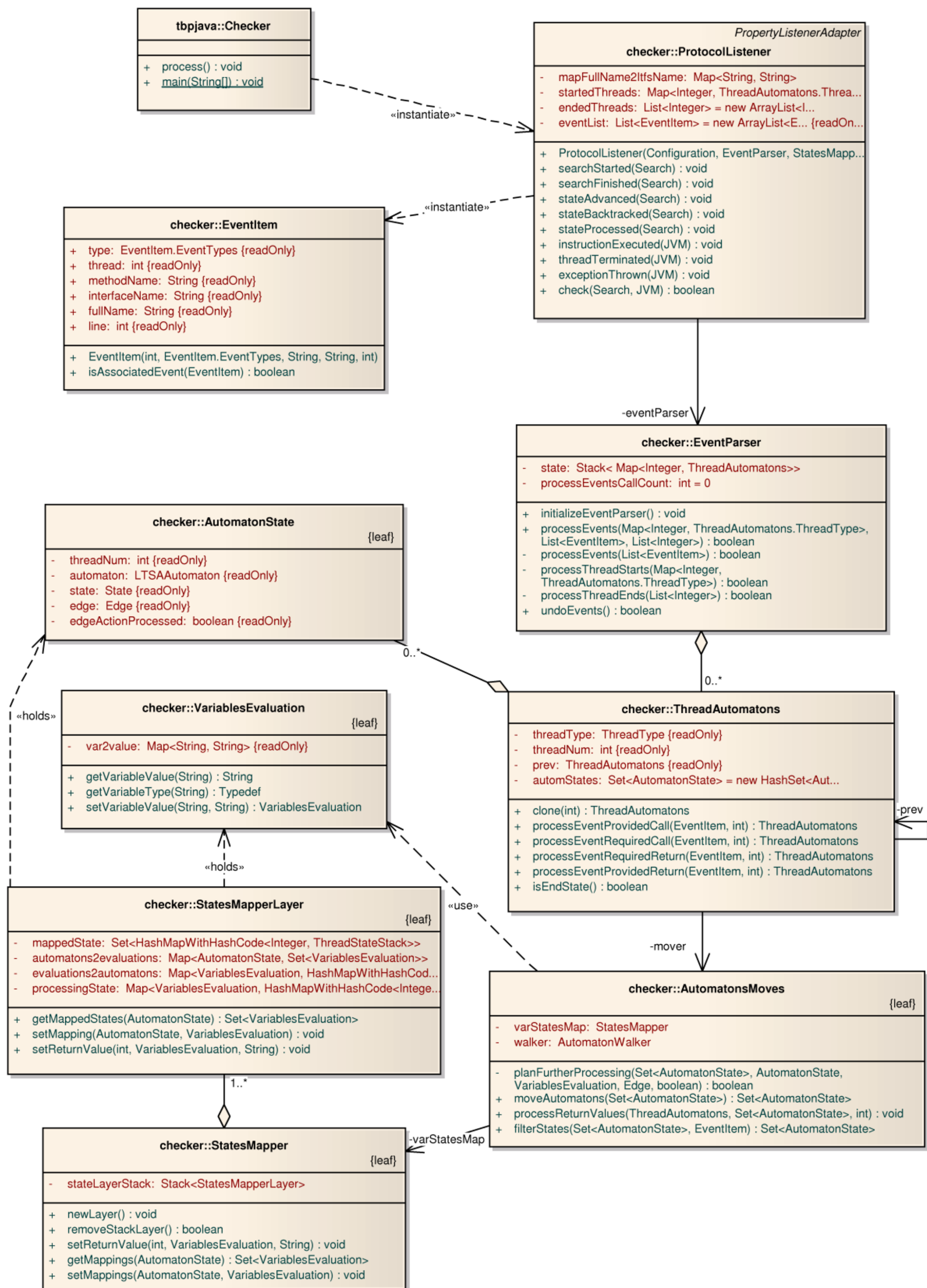

Figure 4.1: Checker internal structure

The ThreadStateStack classes are connected in linked lists, that are synchronized with the ThreadAutomatons lists.

It is possible to enable internal checks to ensure the stored mappings are consistent.

Figure 4.1 contains the structure of the checker part that has been described above. Note that only important methods and classes are shown.

The JPFSearch class is a bit modified standard JPF DFSearch. It traverses program decision tree in the depth first search manner. Program state is typically determined by a program counter, a stack for each thread and by heap content. The checker tool virtually extends the program state with a position in the TBP protocol. We do not extend the program state directly, because there is no easy way to do this in the JPF. We create the JPFSearch class instead of a program state extension. The JPFSearch emulates the behavior of the DFSearch with an extended program state. For more details about the extension of the program state see chapter 3.2.4. The JPFSearch uses JPFProgramStateMapping to test whether the current TBP position has been reached by the given program state.

The JPFProgramStateMapping is a common interface to map the JPF program states to positions in the TBP protocol. The position in the TBP protocol has to be mapped to the program state if the program has been in the given state and the checker tool was at the given protocol position at the same time. It is a one to many mapping. One program state is connected to a set of TBP positions. The JPFSearch uses wasProcessed method only, other methods are notifications of the ProtocolListener. The JPFProgramStateMapping implementors take care of the TBP positions stored in the stack too, because the program states in the stack are not considered to be processed thus they are not stored in the map. There are three different implementations of the interface - JPFProgram-StateMappingEventList, JPFProgramStateMappingTBPProtocolPositions and JPFProgramStateMappingTBPProtocolPositionsHash. They are described at the end of chapter 3.2.4 in more detail.

Diagram with classes related to the state extension can be found in figure 4.2.

The PropertyUncaughtExceptions class handles exceptions. This is the JPF property checking plug-in based on the standard NoUncaughtExceptionsProperty that comes with the JPF. The PropertyUncaughtExceptions class cooperates with the environment generator. If a method from the tested component ends with an exception, then this property is violated and an error is reported. See chapter 4.3 for implementation details.

The AutomatonToDot class helps with debugging. The class generates the ".*dot*" files for the automatons. Generated ".*dot*" files contain types and hash codes of edges and states. Generated output helps to understand created logs.

We considered two different approaches during the design stage of the checker tool - *parallel processing* of TBP states and *single TBP state processing*. We chose the parallel processing which is described in this work.

The idea of the single processing is simple. We have only one TBP position and one variables evaluation at a time. The principle of the work is similar to the JPF. If more choices are possible (more descendant states in the automatons exist) then representation for such

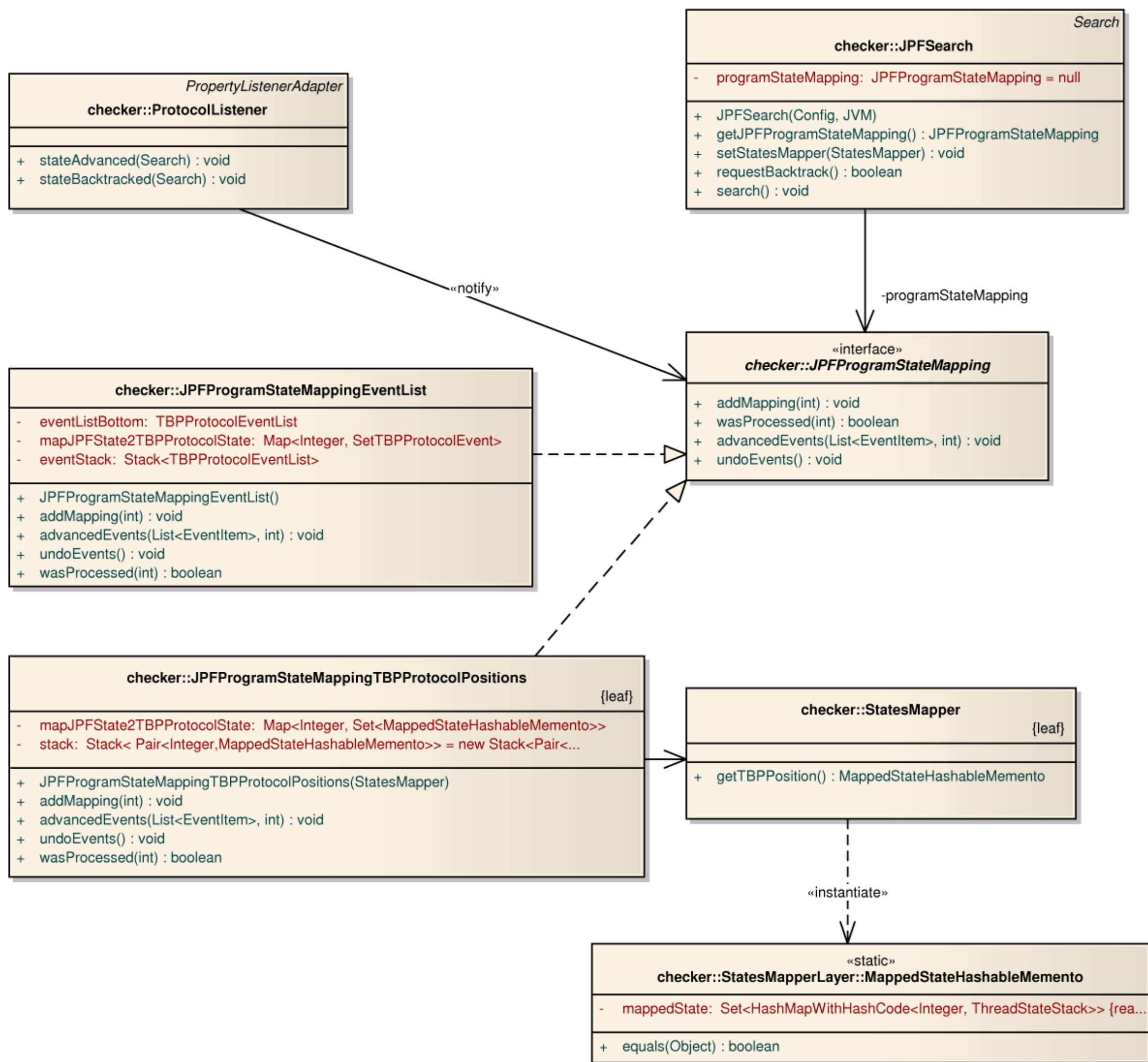

Figure 4.2: JPF state extension structure

choice is created (equivalent of `ChoiceGenerators` in the JPF). The checking can be done in the depth first search manner. If a new event comes, at first we try to go forward. If the expected state (representation of the expected event) is not reached then a backward step is executed and the next choice is processed.

We analyze attributes of both approaches. We suppose that the complexity of both implementations is similar. It is easy to obtain all positions in the TBP protocol in the parallel processing approach (PPA). It is useful for debugging. The PPA has better run time behavior, event processing time corresponds to the number of automatons states and variable mappings. Some steps can be processed very quickly, others can take a long time due to backtracking in the single state approach (SSA). Memory consumption should be slightly smaller in the SSA case. Both cases need to extend program state with processed positions in the automatons. On the other hand, the SSA case will represent current trace more effectively than PPA.

4.2 Environment generator

As we have mentioned before, the environment generator consists of three main classes. The `EnvGenerator` class is an interface of the environment generator, it parses the TBP specifications. This class generates the common parts of the environment files not directly related to the provisions, that means imports, constructor, properties, and the main method. Transformation of the provision into source code in the `runEnvironment` method is delegated to the `CodeGenerator` class.

The `CodeGenerator` class contains the logic responsible for translating provisions to the environment code. It uses a parsed tree of a single provision (which the TBPLib creates) as its input. It generates java source code of the environment as an output. The `CodeGenerator` class is implemented as a visitor of the parsed tree. Basic ideas to generate the environment are introduced in chapter 3.1.3. The implementation contains some improvements over presented straightforward approach, which are described later in this chapter.

The `ProvisionToString` class transforms the parsed tree of the provision into a single line string. Generated description of the provision (sub)part is used as comments in the generated code. Despite its compact form it helps users with orientation in the generated code.

The `StubGenerator` class is used by the `EnvGenerator` to generate representatives (stubs) for the required interfaces of the component. Created stubs are simple, generated methods ignore passed parameter values and the return value is taken from the given value database. This approach is similar to generating parameters for the provided calls. The most important method is `generateStub`. The `StubGenerator` is used by the `EnvGenerator` to create stubs for all required interfaces.

Supplementary class `Type2String` in the `utils` package helps other classes of the environment generator to process class names.

The `EnvValueSet` class represents a database with values to be used as parameters and return values of the generated environment. The user has to provide a customized descendant of this class to the checker tool. Empty database `EmptyValusSet` which is suitable for testing purposes is prepared. The descendant class should create required values and pass them to the appropriate `put*Set` method. The `putObjectSet` method is intended for all types of objects (Strings, HashMaps, etc.). The put methods take up to four parameters - type name, component name, interface name and method name.

The *type name* is used only in the `putObjectSet` case. Even though java provides introspection that can be used to determine type of the object at runtime, we cannot generate the type name automatically. There are two problems - erasure of generic types and child parent relation. Environment generator cannot automatically determine which one of the parent classes or implemented interfaces to use as a type name. Strings with type names are used to obtain values at runtime, the parent-child relation is not considered.

The *component name* specifies the component for which the values are prepared. One `EnvValueSet` instance may be used to test more components. If you use empty component name, then the given values may be used by all components.

The *interface method names* can be used to specify the location where the given values should be used more precisely. In the case of empty interface or method name all interfaces/methods use given values. The most suitable value set is used.

The `Component01.Comp01ValuesDb` in the tests directory is a good example of `EnvValueSet` descendant class.

Figure 4.3 contains classes of the most important methods of the environment generator.

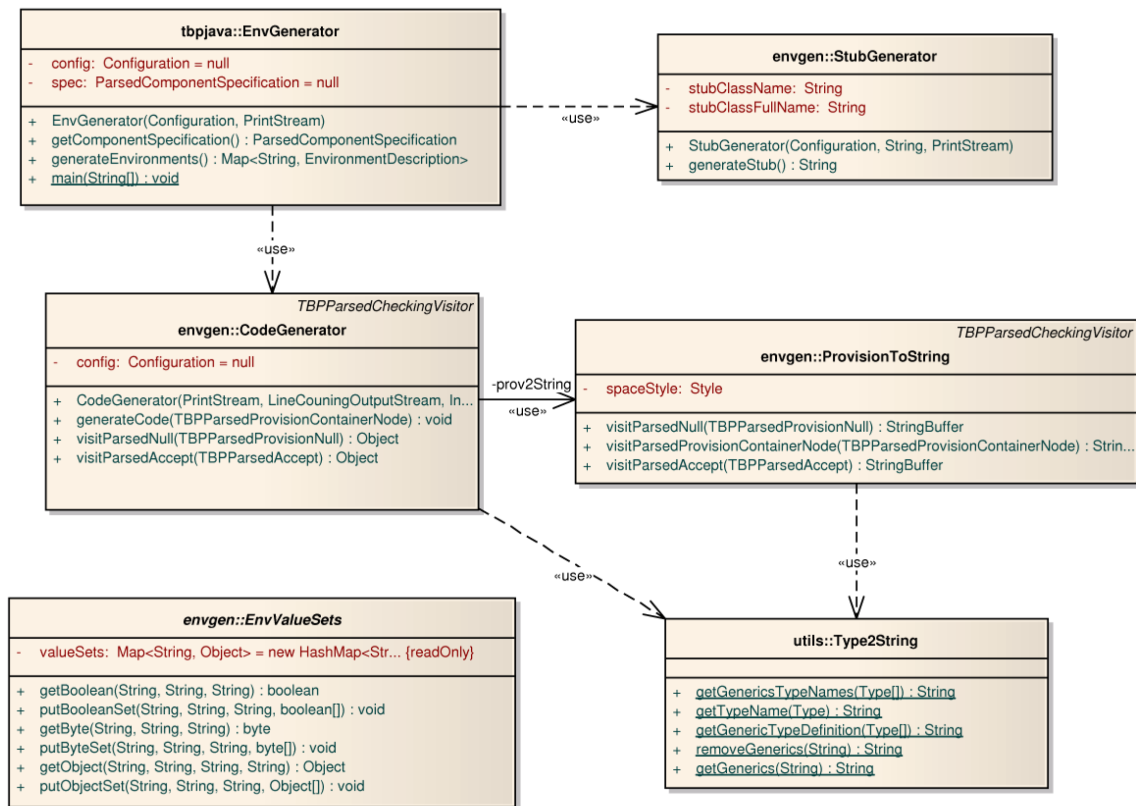

Figure 4.3: Environment generator structure

The code generator groups the same operators together ('+', parallel operators). They are processed in one block (as one n-ary operator). This approach generates more readable code and reduces the number of started threads.

Handling of the or-parallel and the limited reentrancy operators is different to the mentioned straightforward approach. The environment generator does not use the equivalence of these operators with the and-parallel '|' operator. The generator does not remove these operators from provisions in any special step, it generates code for them instead. In the or-parallel case the *A || B* expression is equivalent to the *A + B + (A | B)*, but the environment generator creates threads for each A and B operand as in the and-parallel case. These threads are started conditionally, at least one randomly selected thread is started. The example of the generated code for the *A || B* expression is in figure Code 1.

The generated code in the case of the limited reentrancy operator fits exactly to the code generated by an equivalent definition. Code for the operand is generated multiple times in this case. This is the only operator where the code for the operand is generated more

```
int parallel1 = Verify.random(2);
Thread t1 = new Thread() {
     public void run() {
          A ; // Code for A operand
     }
};
Thread t2 = new Thread() {
     public void run() {
          B ; // Code for B operand
     }
};
if (parallel1 == 0) || (Verify.getBoolean()) {
     t1.start();
}
if (parallel1 == 1) || (Verify.getBoolean()) {
     t2.start();
}
t1.join(); t2.join();
```

Code 1: or-parallel operator code

```
Thread t1 = new Thread() {
     public void run() {
          A ; // Code for A operand
     }
};
Thread t2 = t1.getClass()
     .getConstructor(this.getClass()).newInstance(this);
Thread t3 = t1.getClass()
     .getConstructor(this.getClass()).newInstance(this);

t1.start(); t2.start(); t3.start();
t1.join(); t2.join(); t3.join();
```

Code 2: Limited reentrancy operator alternative code

then once. This could possibly lead to exponential explosion in the size of the source code in case of the A |2 |2 .... |2 provision. But this is not the problem in real use. Different approach is also possible in this case. The code of the operand is generated only once and all other parallel threads are instances of the first one. The source code example for the A |3 expression is in figure Code 2. Nested operators make the implementation of this approach more difficult, but our generator is able to generate such code if set properly.

4.3 Exception handling

Handling of the exceptions is closely related to the generated environment. Methods of the component in checking can throw exceptions, however the TBP does not support them. The TBP protocol does not specify behavior of the component if an exception occurs. If the called provided method ends with an exception then the checker supposes that there is a problem and reports an error.

This chapter describes the way how the checker tool handles exceptions. The generated environment contains threads which call methods of the component being checked. Called provided methods of the component can throw exceptions (declared in the method header), but the `run` methods of the `Thread` class cannot throw *checked exceptions*. We solve the preceding problem by wrapping the code of all run methods by try-catch blocks which catch all exceptions (checked and unchecked) and wrap them into the new unchecked `RuntimeException` that can be thrown by the thread body. The thread terminates with an exception as in the original case, but the exception is different than the original one.

The JPF tool comes with the `NoUncaughtExceptionsProperty` property listener, which reports an error if uncaught exception terminates the tested program. The checker tool uses the `PropertyUncaughtException` property listener which extends the `NoUncaughtExceptionsProperty`. Our property listener unwraps the original exception. The unwrapped exception is used to report an error only, thus it effectively hides the way how the exceptions are handled.

Another way to handle the exceptions is that a called method can throw declared exceptions and it is a legal return from the method. The exceptions declared in the method header can be considered part of that method's programming interface as its parameters and return value are. Any other exceptions than the declared ones (the unchecked exceptions) are illegal and stop the verification. The environment generator is able to generate such environments, where declared exceptions are caught and the environment continues, as is the case of the standard method return. If an exception is thrown, the JPF unwinds the stack and notifies the checker that a method is left. If the checker finds out that the exception leaves the provided or required method then it generates return events for them.

5 Evaluation

5.1 Experiments

In addition to basic development tests, we have evaluated the checker tool against two more real life, yet small, components. Both components use the fractal component model.

First component in the test is part of the *CoCoME* (Common Component Modeling Example) [9][10] system. The CoCoME's task is to create prototype of a trading system as it can be observed in a supermarket. The solution includes the typical procedures at a single cash desk like scanning a bar code, paying by credit card or by cash, as well as administrative tasks like products ordering and managing product database.

We chose the `CashDeskApplication` component to check. The component is responsible for management of single cash desk. The CashDeskApplication collects events from devices like a credit card reader and a barcode scanner and generates events for output devices like a receipt printer. The implementation of the component consists of 400 lines of code in Java. We have done single evaluation of the application. We do not modify the TBP specification or the component. Used EnvValueSet database provides single value for each type.

Second component in the test is part of prototype application developed in the CRE (Component Reliability Extensions) project in cooperation with France Telecom. The application [11] permits WiFi Internet access at airports. The whole application works as a provider of WiFi Internet, supports clients authentication, payments via a credit card, firewall and assigns and manges IP addresses.

We chose `IpAddressManager` which is part of the DHCP server. It is a simple database of IP and MAC addresses, which is responsible for assigning proper IP addresses to clients on the basis of MAC addresses. It handles client's requests for assignment of new IP address, for renewing or releasing assigned IP addresses, it also handles time-out for assigned and not renewed addresses. The implementation of the component consists of 240 lines of code in Java.

We have to modify the `IpAddressManager` component in order to be able to verify it. The component uses actual system time to specify time, when the assigned IP address expires. This is not a permitted interaction of the component with the run-time environment. Even thought the JPF is able to run such programs, this progress enlarges the state space a lot. If the current time is a part of the program state, then the program states which differ only in the time part of the state cannot be recognized as equal states. We created small database which is filled by actual time during start-up. Time entries stored in this database are used instead of the actual system time. It shrinks the program state space approximately 10 times.

We have done multiple evaluations to show impact of the complexity of the TBP and of the size of EnvValueSet data domains on checker performance. We use single values and dual value domains for each type and we remove the outer repetition operator in the short version of the TBP. Removed repetition operator cycles over parallel operator and thus it

gives good chance to rapidly reduce the state space. Extract of the used TBP specification is in figure 5.1, the red color marks removed operator in the short version of TBP.

```
01 : provisions {
02 :   main {
03 :     ?IDhcpServerLifetimeController.Start() ;
04 :     (
05 :       (
06 :          ?IDhcpListenerCallback.RequestNewIpAddress()
07 :          +
08 :          ?IDhcpListenerCallback.RenewIpAddress()
09 :          +
10 :          ?IDhcpListenerCallback.ReleaseIpAddress()
11 :       )*
12 :       |
13 :       (
14 :         ?ITimerCallback.Timeout()
15 :       )*
16 :     )* /* Repetition operator removed in short version */
17 :   }
18 : }
```

Figure 5.1: IpAddressManager - TBP extract - full / short version

CashDeskApp	States [count]	Time [H:M:S]	Memory [MB]
Standard JPF	632 776	00:10:54	142 MB
JPF + Checker	1 056 270	00:12:34	1076 MB

Figure 5.2: CashDeskApp component

IpAddressManager	States [count]	Time [H:M:S]	Memory [MB]
	full TBP / single value domain / time modeled		
Standard JPF	2 153 228	00:43:07	125 MB
JPF + Checker	3 680 375	00:50:33	1775 MB
	short TBP / single value domain / time modeled		
Standard JPF	56 661	00:00:56	121 MB
JPF + Checker	100 423	00:01:23	121 MB
	short TBP / dual value domain / time modeled		
Standard JPF	3 891 151	01:09:35	145 MB
JPF + Checker	9 227 647	02:26:42	2541 MB

Figure 5.3: IpAddressManager component

All tests were executed on Intel Core2 Quad Q9550, 2.83GHz, 3GB RAM, running Gentoo linux and Java HotSpot Server (version 1.6.0_20). Source codes of tested components (in the form used in the evaluation) can be found on the enclosed CD. Results of tests are shown in figures 5.2 and 5.3.

The checker increases the state space in a reasonable way. The duration of the checking grows together with state space. We consider this increase acceptable. On the other hand the memory requirements grow more dramatically. Memory consumption of more complex components or larger value domains exceeds our hardware limits. The checker needs approximately 1kb to store one state, which is unacceptable to us.

Two or more value domains cause extreme increase of the states count (time and memory) thus it is recommended to use single value domains only. Simplification of the TBP provisions may significantly reduce the checking time (and required resources).

5.2 Future work

As to future work, we want to lower the memory requirements by smarter extension of the JPF program state. This is our highest priority task.

Stubs generated by the environment generator do not throw exceptions, thus exception handling code in the component is not tested. We would like to extend the `EnvValueSets` to be able to throw exceptions. Behavior protocols lack exception support. Even thought a proposal for extending the original BP with exceptions exists [12], it has not been used and implemented due to its complex syntax. We would like to adjust the proposal to fit the TBP and update the TBPLib and the checker accordingly.

The TBP is a high level abstraction, thus relation between the TBP specification and the implementation is imprecise. We feel the necessity to remove the semantic gap and refine the relation. This topic covers more fields:

- mapping of the internal threads created by the component to the threads of the TBP specification
- mapping of the TBP internal state and the component internal state
- relation of the TBP parameters and the TBP return values to the parameters and the return values used in the environment

As we have mentioned before in chapter 3.2.5.2, the way the current version of the checker handles the threads lacks precise mapping of thread in the component to single thread section entry in the TBP. Even though the checker handles the threads reasonably, the used approach does not meet the TBP requirements. Current version does not check the counts of the threads of the component, it is possible to map more threads of the component to single TBP threads section entry now.

We have discovered two different solutions to this problem. The first approach is based on the annotations and requires user's action. The user specifies the name of the TBP threads section entry in the annotation of the thread class used in the component implementation in this case. The second solution can be automated and it needs not modify the implementation of the component or the TBP specification. There is no way to determine the proper TBP threads section entry when a new thread is created. On the other hand, if

the thread of the component terminates it is easy to find which thread sections entries correspond to the terminated thread. They correspond if the thread reaches the final state of the automaton of the entry. If the checker would store corresponding entries for each thread, it would be possible to add a new test to detect whether more threads of the component are mapped onto single TBP thread entry. It is possible to combine both presented approaches.

Even though the checker supports return values in provisions, the way the checker handles them is not the same as the intention of the TBP's authors. If there are more branches with different return values in the provision then according to the authors, the implementation of the component chooses which branch to use. The generated environment selects the used branch (expected return value) in the current checker version.

Large scope changes mainly in the environment generator part are necessary. The environment generator cannot generate the code directly from the annotated syntax tree of the provision as the current version does because the emits which differ in return values are not grouped together as it is needed. Related emit events should be grouped into a single emit event with multiple return paths (for each return value). The grouping is done during the conversion of the provision into automaton in the TBPLib, thus it is necessary to use this form of provisions in the environment generator. Note that for our purposes the automatons may be nondeterministic.

How to find out the return values used? We may obtain the return value from the automaton of the provided method called by the emit event, but this return value is chosen by the TBP specification event and it can be selected nondeterministically. The return value is not chosen by the implementation of component but by the specification in this case, so it is necessary to use a different approach.

Return value has to be gained from the implementation. We may get the return value of the called method easily, but this value is a java type value and in the TBP specification the return value has a different type from the TBP types section. So we need to map the implementation return values into the TBP ones. The easiest way to do this is to extend the `EnvValueSet` to store the TBP value for each entry that can be used as a return value. Even though this approach should solve the given problem, it is a very simplistic one. We plan to investigate how this problem is solved in other tools and provide more complex solution which refines the relation of the implementation and the TBP specification.

The creation of the value database (`EnvValueSet`) is a process susceptible to errors. It is easy to omit some value types. A tool, which generates the skeleton of the value database where all usages of given type on the provided and required interfaces are extracted, will help a lot.

6 Conclusion

We have created a tool that checks the code conformance of a primitive component written in Java against the TBP specifications. The tool is able to handle all parts of the TBP protocol except return values in the provisions. We have shown the way to handle the TBP threads and other concepts new in the TBP. Various techniques to represent positions in the TBP have been implemented. This work shows that it is possible to test whether the implementation fulfills the TBP specification.

We have also evaluated the checker tool on two examples. The tool checks both examples in a reasonable time, but it has been necessary to do some modifications and/or simplifications. Current version of the checker tool suffers from big resource requirements and thus it is not possible to verify real life size components. We plan to address some of these issues and limitations in future.

7 Bibliography

[1] Parizek P., Plasil F., Modeling Environment for Component Model Checking from Hierarchical Architecture, FACS'06, 139-153, Jun 2007

[2] Parizek P., Adamek J., Kalibera T., Automated Construction of Reasonable Environment for Java Components, FESCA 2009, 145-160, Oct 2009

[3] Kofron J., Poch T., Sery O., TBP: Code-Oriented Component Behavior Specification, SEW-32, IEEE, 75-83, Jan 2009

[4] Adamek J., Modeling Unbounded Parallelism Using Behavior Protocols, Tech. Report No. 2005/7, KSI MFF, Nov 2005

[5] Graphwiz, www.graphviz.org

[6] The Java PathFinder, http://babelfish.arc.nasa.gov

[7] JPF Tutorial, http://www.visserhome.com/willem/presentations/ase06jpftut.ppt

[8] Parizek, P., Plasil, F., Kofron, J., Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker, SEW-30, IEEE, 133-141, Jan 2007

[9] CoCoMe, http://agrausch.informatik.uni-kl.de/CoCoME/

[10] Bulej L., Bures T., Coupaye T., Decky M., Jezek P., Parizek P., Plasil F., Poch T., Rivierre N., Sery O., Tuma P., CoCoME in Fractal, Springer-Verlag, Aug 2008

[11] Internet access demo, http://fractal.ow2.org/

[12] Holub V., Enhancing Behavior Protocols with Exceptions, WDS 2005, 30-35, Jun 2005

Appendix A - TBP Grammar

Currently more versions of the TBP grammar exist. Our project uses the older version. In order to avoid confusion we attach grammar used in our checker.

Start symbol

```
TBP                 : 'component' IDF_NAME '{'
                          'types'       '{' TYPE*          '}'
                          'vars'        '{' VAR*           '}'
                          'provisions'  '{' PROVISION*     '}'
                          'reactions'   '{' REACTION*      '}'
                          'threads'     '{' THREAD*        '}'
                      '}'
```

Note: All sections are required, but can be empty.

```
TYPE                : IDF_TYPE    '='     '{' IDF_CONST ( ',' IDF_CONST )* '}'
VAR                 : IDF_TYPE    IDF_VAR    '='    IDF_CONST
                    | 'mutex'     IDF_VAR
```

Provisions

```
PROVISION           : IDF_NAME? '{' P_PROTOCOL '}'

P_PROTOCOL          : P_ALTERNATIVE
P_ALTERNATIVE       : P_SEQUENCE        ( '+'   P_SEQUENCE      )*
P_SEQUENCE          : P_PARALLEL        ( ';'   P_PARALLEL      )*
P_PARALLEL          : P_OR_PARALLEL     ( '|'   P_OR_PARALLEL   )*
P_OR_PARALLEL       : P_POSTFIX_OP      ( '||'  P_POSTFIX_OP    )*
P_POSTFIX_OP        : P_TERM            ( P_POSTFIX_OP1         )?
P_POSTFIX_OP1       : '*' | '|*' | ('|' DIGITS)
```

Note: Only one postfix operator at a time.

```
P_TERM              : '(' P_PROTOCOL ')'
                    | ANNOTATION? P_EVENT
P_EVENT             : '?' METHOD_CALL     ( ':' IDF_CONST )?
                    | 'NULL'
```

Reactions

```
REACTION            : ANNOTATION?METHOD_DECL      '{' R_STMT '}'
THREAD              : IDF_NAME?                   '{' R_STMT '}'

R_STMT              : R_STMT1 ( ';' R_STMT1 )*
```

Note: ';' is an operator not as ';' in ordinary programming languages. There cannot be ';' after the last event.

```
R_STMT1             : ANNOTATION? R_STMT1
R_STMT_TYPES        : R_WHILE
                    | R_SWITCH
                    | R_IF
                    | R_SYNCHRONIZED
                    | R_EVENT
                    | R_ASSIGN
                    | R_RETURN
                    | 'NULL'
R_SWITCH            : 'switch' '(' VALUE ')' '{'
                              R_SWITCH_CASE+
                              R_SWITCH_DEFAULT?
                         '}'
                    | 'switch' '(' '?' ')' '{' R_SWITCH_CASE2+ '}'
R_SWITCH_CASE       : IDF_CONST    ':'     '{' R_STMT '}'
R_SWITCH_CASE2      : 'case'       ':'     '{' R_STMT '}'
R_SWITCH_DEFAULT    : 'default'            ':'     '{' R_STMT '}'
R_WHILE             : 'while'  '(' GUARD ')'  '{' R_STMT '}'
R_IF                : 'if'     '(' GUARD ')'  '{' R_STMT '}' ( 'else' '{' R_STMT '}' )?
R_SYNCHRONIZED      : 'sync'   '(' IDF_VAR ')'      '{' R_STMT '}'
```

Note: Variable has to have mutex type.

```
R_EVENT             : '!' METHOD_CALL
R_ASSIGN            : IDF_VAR '<-' VALUE
```

Note: Parameters are not considered to be variables. It means you cannot assign new value to a parameter.

```
R_RETURN            : 'return' IDF_CONST

GUARD               : IDF_VAL '==' IDF_VAL
                    | '?'
```

Note: Comparison of two variables is not supported. At least one identifier has to be a constant. Only equality testing is supported.

Note: '**?**' means nondeterministic choice - any branch can be selected.

```
VALUE               : IDF_VAL
                    | '!' METHOD_CALL
```

Note: Called method has to be a function. Return value is needed.

Methods

METHOD_NAME : *IDF* '**.**' *IDF*

Note: Method name consists from *interface name* '**.**' *method name*.

METHOD_CALL : *METHOD_NAME* '**(**' *PARAM_LIST*? '**)**'

PARAM_LIST : *IDF_VAL* ('**,**' *IDF_VAL*)*

Note: Only constants can be used as parameters in the provisions.

METHOD_DECL : *METHOD_NAME* '**(**' *PARAM_DECL_LIST*? '**)**' ('**:**' *IDF_TYPE*)?

PARAM_DECL_LIST : *IDF_TYPE IDF_PARAM* ('**,**' *IDF_TYPE IDF_PARAM*)*

Note: Braces after the method name are mandatory.

Technical definitions

DIGITS : ('**0**' .. '**9**')+

IDF : ('**a**'..'**z**' | '**A**'..'**Z**') ('**a**'..'**z**' | '**A**'..'**Z**' | '**0**'..'**9**' | '**_**')*

Note: Identifier has to begin with a letter. National characters are not supported.

Note: Following IDF_* are here only to distinguish different semantics of identifiers.

IDF_NAME : *IDF*

IDF_TYPE : *IDF*

IDF_VAR : *IDF*

IDF_PARAM : *IDF*

IDF_CONST : *IDF*

IDF_VAL : *IDF_CONST* | *IDF_VAR* | *IDF_PARAM*

Note: IDF_VAL represents a value which can be used in reactions as a parameter or return value.

WS : (' ' | '**\n**' | '**\r**' | '**\t**')+

COMMENT : '**/***' .* '***/**'

Note: Singe line comments (//) are not supported.

ANNOTATION : '**@**' *IDF* ('**(**' (*IDF* '**=**' *IDF*)+ '**)**')?

Note: Annotations are not used by the checker now.

Appendix B - Checker configuration

You have to provide configuration (file) to the checker to be able to work. Configuration is related to the tested component. Thus each component has its own configuration. Among others the configuration maps names from the TBP specification to java classes/interfaces which contain implementation/definition. The configuration can be provided in two forms as a text file or as a checker command line parameters. Both ways use the same syntax and they are equivalent, so you can set the same configuration entries in both ways. The command line configuration entries override the values specified in the file.

First command line parameter of the checker contains the name of the configuration file and the following parameters (if exist) hold command line entries.

The configuration file is line oriented. Line starting with '#' is considered a comment. Each line contains one entry. Entries are case sensitive. They comprise three parts - a property name, the equality sign ('=') and a value.

You can set following properties:

- Property "*component.name*" defines the name of the component being tested. This name has to match to the component name defined in the TBP specification. Main purpose of this entry is to find errors when configuration is applied to a wrong component.
- Property "*component.implclass*" holds the name of the java class with implementation of the component. The class name has to be a "*fully qualified name"* with the package part.
- Property "*component.provitfs*" defines the provided interfaces of the component. Value part is a list of pairs separated by semicolon (';'). Exactly one pair exists for each provided interface. The *pair* contains an interface name, the minus character ('-') and a fully qualified name of the class where the interface is defined. See example below.
- Property "*component.reqitfs*" defines the required interfaces that the component needs. Format of the value is the same as in previous component.provitfs case.
- Property "*component.reentrancylimit*" specifies the number which is used in conversion of the unlimited reentrancy operators into the limited ones. This number should be calculated from the component composition. The default value is 2.
- Property "*env.valuesets*" defines the fully qualified name of the class with database where prepared values for parameters are created. This class has to be a descendant of the `EnvValueSet`.
- Property "*env.targetdir*" specifies the directory, where the checker generates the files. That means the environment and the .dot files with automatons. The default value is the current working directory. The directory is created if it does not exist.
- Property "*env.protocol*" defines the file name with the TBP specification for the given component. This is a vital part of the configuration.
- Property "*env.generate*" defines whether the checker should generate the environment or not. If classes are not generated it is expected that they exist (in the env.targetdir or

elsewhere on the classpath). This property is useful mainly for debugging. The value can by *yes* or *no*. The default value is yes.

All referenced classes (the component class, the provided and required interfaces, the values set database class) have to be accessible. That means that the CLASSPATH has to be set accordingly. Classpath need not contain the env.targetdir directory, the checker adds this path to the classpath automatically.

Properties *component.name, component.implclass, env.valuesets, env.protocol* are mandatory. *Component.provitfs* and *component.reqitfs* have to be defined only if the component provides or requires any interface. Typically you set the *env.targetdir* and *component.reentrancylimit* property too. Other properties are optional and have suitable default values.

In figure B.1 there is an example of a simple configuration file. The component has 2 provided and 2 required interfaces.

```
# Test component 01 configuration file
component.name=Component01
component.implclass=Component01.Comp01
component.provitfs=P1-Component01.PI01;P2-Component01.PI02
component.reqitfs=R1-Component01.RI01;R2-Component01.RI02
env.valuesets=Component01.Comp01ValuesDb
env.targetdir=./genEnv-Component01
env.protocol=tests/Component01/Comp01.tbp
```

Figure B.1: Configuration file example

Appendix C - User documentation

The source code of the checker tool is provided with the ant build script. The project contains all required libraries. The most important ant target is "*jar*" which compiles and packs the checker and required libraries into single *Checker-*.jar* file (where '*' should be replaced by actual date). You may use the "*java -jar Checker*.jar parameters"* command to run the checker. At least one parameter with the configuration file is expected. Format of the configuration file is described in appendix B. You has to set classpaths properly so that the checker tool should be able to access the tested component.

You have to have the JDK 1.6 or newer installed to be able to run the checker tool. Note that JRE is not enough to run the checker, because it lacks the javac compiler used to compile the generated environment.

We have created scripts (run.checker.sh, run.checker.bat) which can be used to run the checker easily. These scripts were used to run the evaluations. These scripts set system resource limits (in the linux case), compile provided source file, adds to the classpath compiled sources and 'jar" files in the "lib" directory (if present) and run the checker tool.

The first parameter of the script points-to the directory with sources to compile, other parameters are passed to the checker. The second parameter is typically the configuration file.

Execute "*run.Checker.sh src CashDeskApp.cfg*" in the /tests/CashDesk directory (on the attached CD) to start the checker. Note that you have to copy the content of the CD into write-able medium. Write access is necessary to be able to compile the sources of the component and to create the environment. It is necessary to set executable rights to the run.Checker.sh script ("*chmod u+x run.Checker.sh*") on unix based systems.

The checker processes provisions separately, so each provision has its own result. The result is standard JPF result extended by the event trace if an error is detected. Possible results are: No error detected, Tested component is not compliant with the protocol or Out of memory (Search constraint hit). In the last case the JPF was unable to check all possible traces because of limited resources. Number of visited states as well as required memory is also printed.

Appendix D - Content of attached CD-ROM

This thesis is accompanied by a CD ROM containing source codes, compiled checker and a set of examples. The structure of the CD ROM is as follows:

`/Thesis-Checker.pdf`

Electronic version of this document.

`/tbpjava`

The checker tool project directory. The directory contains ant build script, scripts to run checker easily on Windows and Linux systems, generated javadoc and precompiled checker.

`/tbpjava/src`

Source codes of the checker tool.

`/tbplib`

Library used by the checker tool to process the TBP file.

`/tests`

Tests used in the evaluation.

`/showcase`

Two smaller tests not used in the evaluation. Tests can be run by included scripts.

`/docs`

Various material related to master thesis document.