Saarland University
Faculty of Natural Sciences and Technology 1
Department of Computer Science

Bachelor thesis

Efficient Typestate Verification for Java

submitted by

Nikolai Knopp

on

November 24, 2009

Supervisor

Prof. Dr.-Ing. Andreas Zeller

Advisor

Dipl.-Inf. Valentin Dallmeier

Reviewers

Prof. Dr.-Ing. Andreas Zeller,
Jun.-Prof. Dr. Sebastian Hack

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, September 29, 2009

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, September 29, 2009

Acknowledgements

First of all, I deeply thank my family and my friends for all their support before and throughout my studies. I especially thank you, Susanne, for cheering me up whenever things seemed to grow above my head.

I thank Andreas Zeller for providing me with this interesting topic, which allowed me to investigate typestate analysis all the way from its background to the difficulties in implementing it.

Thanks go to Valentin Dallmeier for his feedback, his guidance and all the good ideas which helped me a lot in writing this thesis.

Furthermore, I want to thank Christoph Mallon for his continuous work on jFirm and, together with Sebastian Hack, for hours of discussions on data flow analysis and lots of nerves.

Also, I thank Klaas Boesche for his cooperation on finding suitable interfaces between our programs and for enlightening me about points-to analysis.

I am very thankful to Markus Rabe, Michael Stoll and Donal Stewart for proofreading large parts of this thesis and bearing with me for I could only give them little time to do it.

Holger Hermanns has my gratitude for granting me the deadline extension I badly needed, because the preparation and the early start of my semester in Edinburgh cost me a lot of time.

Abstract

In this thesis, we investigate the feasability of supporting software development by static program analysis. We present an algorithm for partly interprocedural context-sensitive typestate analysis on Java programs which builds on jFirm, a Java implementation of the SSA-based intermediate language Firm [LBBG05], and uses finite state machines as typestate specification.

We implemented the algorithm as a Java application, jFTA. Given the bytecode of the programs to be tested and a set of typestate specifications, it reports typestate violations to the programmer and provides automatically derived fixing suggestions. It uses a jFirm-based library [Boe09] for points-to resolution and approximates variable aliasing.

We use the automated usage model miner ADABU [DLWZ06] to gather the required typestate specification, and by this assess the usability of mined models as specification for typestate analysis. A case study using real world examples from Columba [col] unveils interface problems but no major conceptional flaws. We furthermore evaluate the speed of jFTA and the quality of the reported violations, showing that both are so far only satisfactory for nearly intraprocedural analysis of simplistic test cases. Also, the results suffer from limitations imposed by jFirm and the points-to library which are in early development stages.

Contents

1 Introduction

During software development, programmers spend a lot of time on testing and debugging their code, trying to keep their product bug-free. Bugs are caused by programming errors which are classified as compile-time or runtime errors. While the former are hard to miss as the compiler will complain about them, the latter may not be discovered because they only occur under rare circumstances. Such a bug in a very rarely used subroutine might cause software to crash unexpectedly after running for a long time.

The causes of runtime errors are manifold. Prominent examples include failure to sanitise user input values, correct range checking for internal variables or violation of (implicit) invariants. In this thesis, we want to address a subset of these last items. The specification of a type often introduces invariants which define different states for instances, depending on which only a subset of all the available methods may actually be called (are ***enabled***). ***Disabled*** methods may not be called as they can produce errors. As a result of a method invocation, the object's state may change and therefore enable other methods. For example, such invariants are used to ensure that only opened files are read or that an iterator's next element is only acquired after explicitly checking for its existence.

Ensuring compliance with all given invariants becomes harder as programs grow. Unexpected exceptions will likely not be handled correctly and can therefore crash the whole program. The programmer does not get any warning of this before the program is run, because the aforementioned invariants are only – often implicitly – given in the type's natural language specification. Compilers cannot automatically extract these invariants and therefore do not detect if they are violated.

1.1 Tasks

In this thesis, we want to develop an automatic invariant checking system named jFTA which shall warn the programmer of bugs of this kind. To accomplish this, we have several challenges to address:

1. Analyse how objects are used in a Java program which might still be in development.
2. Extract the invariants to which the program must comply.
3. Check the observed usage patterns from step 1 against the specification from step 2 and inform the user of any violations.

Using jFTA should result in a reduction of both the amount of undetected runtime errors and also the time the programmer spends on debugging. In addition, we want our system to integrate as seamlessly as possible into the programmer's work-flow, keeping the additional work required to use it at the minimum. Therefore, we aim to gain the needed specification automatically instead of having the programmer extract it by herself.

1.2 Thesis structure

The rest of this thesis is organised as follows. We explain how invariant violations can be detected using ***typestate analysis*** [SY86] and discuss related

work in Section 2. We introduce the intermediate program representation language ***Firm*** [LBBG05] and its Java implementation ***jFirm*** in Section 3. The invariant extraction will be realised using the automatic usage model miner ***ADABU*** [DLWZ06], which we present in Section 4. We discuss our analysis algorithm in Section 5, describe its implementation as a Java program in Section 6 and evaluate it in Section 7. We conclude in Section 8.

2 Problem overview

In this section, we use a short example to clarify the nature of our target problems before we analyse their features in more detail. After presenting the basic techniques needed to detect these problems, we give a brief overview about related work.

2.1 Example

In our example, we want to check a remote IMAP mailbox for new mail. This is done by the Java method `mailAvailable()` given in Figure 2.1, which uses an implementation of the IMAP Protocol [IMA, ris]. In order to receive the wanted information, the specification of the protocol tells the programmer to have the program perform the following steps in this order:

Step	Client action	Server response
1.	Connect to server	Authentication query
2.	Provide credentials	List of available mailboxes
3.	Select wanted mailbox	Information about selected box
4.	Close the selected mailbox and terminate the connection.	"Goodbye" message

The order of the commands being executed is important, because the client's session with the server can be in different states, e.g. connected or authenticated. The valid sequence of protocol state transitions resulting from execution of the example method can be illustrated as the finite state automaton in Figure 2.2.

Violating the specified order may result in different error situations. The server will, for example, deny the client's request for selecting a mailbox if the session is not in `AUTHENTICATED` state. Thus, the login request has to be executed prior to any mailbox requests. If omitted, the invocation of `select("inbox")` in line 5 would throw an `IMAPException`. Depending on whether the method calling `mailAvailable()` handles exceptions correctly or not, the program would either report communication problems with the server (best case), or crash unexpectedly because of the unhandled exception (worst case). In any case, this flaw would prevent this method from ever working correctly, probably resulting in a severe program failure after the program was running for some time.

```
1 public boolean mailAvailable() throws IOException,
      IMAPException, AuthenticationException {
2   IMAPProtocol imap = new IMAPProtocol(host, port);
3   imap.openPort();
4   imap.login(username, password);
5   boolean newMailAvailable =
        imap.select("inbox").getFirstUnseen() != -1;
6   imap.close();
7   imap.logout();
8   return newMailAvailable;
9 }
```

Figure 2.1: A Java method which checks an IMAP inbox for new mails.

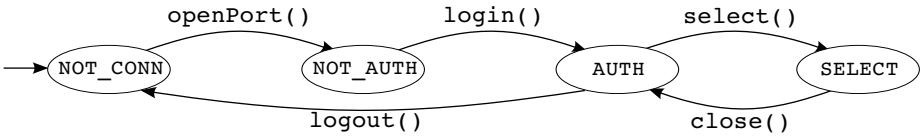


Figure 2.2: A valid sequence of method invocations on an instance of `org.columba.ristretto.imap.IMAPProtocol`, assuming that the provided user credentials are correct and that the network connection is stable.

It is desirable to be informed about such errors as soon as possible, especially before the program is executed. The more quickly the programmer notices the error, the easier it is for her to retrace the functionality of the part in question and to find a solution. Ideally, the programmer would receive a warning directly after writing erroneous code lines. This could be achieved by running an appropriate analysis right after compilation and using an integrated development environment like Eclipse [ecl], which constantly compiles in the background.

2.2 Static program analysis

The problem with runtime errors is that their occurrence depends on actual values of variables and memory locations, which are unknown prior to program execution. Checking for this type of errors in advance requires to somehow estimate possible memory values and identify problematic configurations. The research area dealing with this type of problem is called ***static program analysis***, and is often integrated into compilers. The main idea is to not only let the compiler transform code from a higher to a lower language level, but also interpret it to check certain qualities or invariants by approximating its behaviour when executed. Examples include checking for liveness of variables and pointers or assuring that pointers are only dereferenced after initialization. This can be achieved by enhancing the compiler with a set of rules which build and operate on an abstract representation of the program during or after compiling the code.

Most static analyses generate an over-estimation of the program's behaviour. They never ignore a value or execution path in the program which could occur with a probability greater than 0, but most likely also consider situations which can never happen in an actual program execution. This guarantees that the analysis will never miss any potential program behaviour, which makes it ***sound***. Without soundness, the results of the analysis would obviously not be safe to use for performing code optimization or program verification. On the other hand, results of an analysis can be safe but useless if they contain a lot of superfluous information resulting from over-approximation.

An optimal analysis would neither over- nor under-approximate at any point. This would correspond to running the target program on every possible input, also including data entered by the user or received via network sockets. Doing so in a finite amount of time would be equivalent to solving the Halting Problem [Tur36], which is why such an analysis does not exist. The challenge lies in finding the right compromise between approximation quality and analysis complexity, i.e. getting usable results in as little time as possible.

2.3 Dataflow analysis

A method which supports these requirements with adjustable level of detail is dataflow analysis. As the name suggests, it models the way data flows through a program by a directed graph [NNH04]. Every node in this graph is assigned an ***input*** and an ***output data set***. The actual logic of the dataflow analysis is encoded into the ***transfer function***. It takes a node n plus its input set and generates the new output set for this node, which is then merged into the input sets of all successors of n by a ***merge operation*** $\sqcup$.

The analysis begins with the specified first node in the graph, which is the only node required to have a pre-defined input set in the beginning. It advances to the other nodes by following the graph's edges, merging output into input sets and applying the transfer function to the successors. By requesting the transfer function and $\sqcup$ to be monotonic with respect to a finite, partial ordering on the input/output sets, the algorithm is guaranteed to terminate by reaching a unique fixpoint for the data in all sets even if the graph is cyclic. The nodes' output sets then contain the results of the analysis.

The merge operation in combination with this termination criterion actually leads to the aforementioned over-approximative results of the analysis. As we will see in Section 3.1.1, there are graph views on programs which support dataflow well through their structure.

2.4 Typestate analysis

As initially mentioned, the error discussed in our example is caused by invoking a method disabled by the current state of the affected object. This class of problems can be characterized as ***typestate violations***. Typestate [SY86] is an extension of the concept of types. While the type of an object determines the set of all operations which can be ever performed on an object, typestate only enables a subset thereof, based on the current state of the object. Execution of an enabled operation can change the typestate of an object, which might disable or enable other operations. Whenever a disabled operation is executed, the object switches to an ***error state*** to express a typestate violation.

In the example, `select("inbox")` is disabled unless the `imap` object is in `AUTHENTICATED` state. If the invocation of `login(username, password)` would be left out by error, `imap` would be in `NOT_AUTHENTICATED` state and calling `select("inbox")` would be a typestate violation (see Figure 2.3). As compilers in typed languages like Java only perform type-checking and do not know about typestate, the example method will compile fine whether `login()` is invoked or not; the latter resulting in possible runtime errors.

Detection of these violations can be implemented as a dataflow analysis. It begins at the starting point of the program and processes the instructions in the order which they would be executed in when running the program. From seeing every single program instruction, the analysis can identify those which would create or manipulate objects. However, as the program is not really executed, there are no ***instances*** (concrete objects) which the analysis could work with. Instead, it has to create and use ***abstract objects*** of which each represents one or more concrete objects. The needed abstraction is realized through the typestate, as it categorises instances by the values of their fields, which actually define an object's state. Thus, one abstract object – only providing the type and

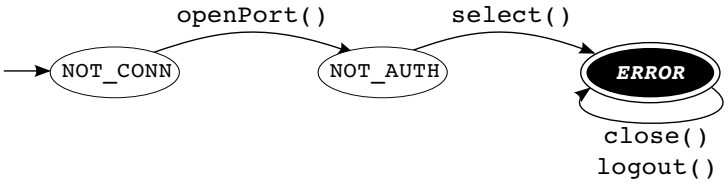


Figure 2.3: Typestate transitions leading to a violation. Once reached, the error state cannot be left.

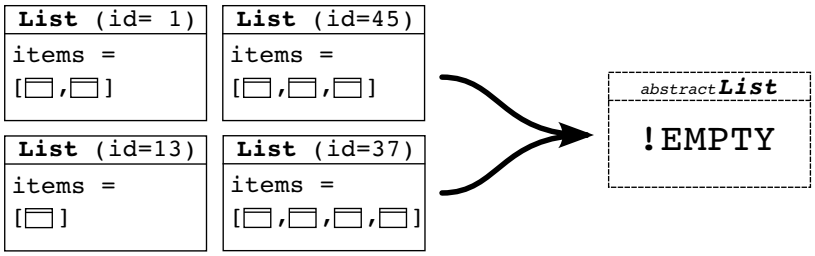


Figure 2.4: Many different concrete List objects are represented by the same abstract object by identifying object configurations with abstract states.

current typestate – can represent a potentially infinitely large set of instances (see Figure 2.4). If, in real execution, the state of an instance eventually changes as a result of a method invocation, this may result in a transition to another typestate during analysis. Defining those abstract states and transitions is not canonical but must be done individually for every type. This will be covered in detail in Section 4.

2.5 Accuracy vs. complexity

Finding instructions which create objects (***allocation sites***) is quite straightforward in Java, as objects are always created by a **`new`** statement. Tracing the objects through the rest of the program is what can quickly become very hard. If, for example, the program contains conditionals or loops, their branch conditions normally can only be evaluated approximatively, if at all. The analysis must therefore always assume that a condition might evaluate to both **`true`** and **`false`**, and investigate both resulting paths, unless it can exclude one option soundly. This can lead to an immense amount of paths whose analysis can take a significant amount of time and memory. By making the analysis insensitive to some aspects paths are merged together, thus trading off precision against runtime by reducing the complexity.

Usually, dataflow analyses are classified by four such aspects.

Flow-sensitivity Flow-sensitive analyses respect the order in which instructions are executed in parts of the program (usually on a per-method basis). Flow-insensitive analyses ignore the order and can thus only accept parts

where none of the operations might be conflicting (i.e. they are all enabled for every reachable typestate).

Interprocedurality If an interprocedural analysis reaches a call to another method, it will continue in the body of the called method. If performed intraprocedurally, it will not follow the invocation but proceed with the next line in the current method. Because the effects of the called method are then unknown, it must be assumed that the method might have cause every possible effect on the current data, resulting in a drastic loss of precision.

Context-sensitivity A method can be called from multiple locations such that typestate and/or parameter configurations differ for each invocation. An analysis is context-sensitive if it considers each of these ***calling-contexts*** separately and propagates the results only to the actual caller. Context-insensitive implementations merge all potential configurations and return the same result to all calling sites.

Path-sensitivity There are many situations in which execution paths can split, for example at conditionals, in loops or if exceptions are thrown. A path-sensitive analysis will follow each of them separately, while insensitive analyses will merge them again when they reach the next shared instruction (e.g. at the first instruction following the loop/conditional).

Also, there is the problem of references ***aliasing*** each other, which occurs if multiple references $\texttt{ref}_1, \ldots, \texttt{ref}_n$ point to the same object. The analysis must ensure that a change to the state of an abstract object `obj`, accessed via reference $\texttt{ref}_i$, is also visible if `obj` is later accessed via one of the other references $\texttt{ref}_j$ with $\texttt{j} \neq \texttt{i}$. Additionally, the approximating simulation can cause one variable to possibly contain references to more than one abstract object, for example if objects are retrieved from arrays or, for path-insensitive analyses, when paths are merged. A technique for finding all possible targets for a variable is ***points-to analysis*** [Wei80]. At runtime, this of course is not necessary, because a reference contains exactly one memory address.

These are typical issues with dataflow analysis, which will we address in detail in Section 5.

2.6 Requirements

To perform static typestate analysis, a suitable representation of the target program as well as typestate specification for the used objects input are required. The way in which this information will be provided can have a fundamental impact on the usability and utility of the analysis, so it should be considered carefully. Ideally, obtaining the required input data should be as little extra work as possible for the user. For that reason, it is common practice to take the source code of a program as its representation, and let the analysis extract the object behaviour by running dataflow analysis on it.

While the program representation can potentially come for free (source code is always available for programs which are in development), specifying the correct usage of objects is more complicated. For most types, information about its possible states and legal operations are only given in the documentation. Not

only that natural language can be hard and expensive to parse; information is often given only implicitly, which renders the potentially existing specification useless as input for an analysis. Thus, the specification often must first be generated by the programmer. This involves thinking about the possible states an object could be in, the state transitions which operations might perform, and when operations are enabled or not. The more complex the type in question is, the harder it can be to capture every possible case and to have the specification actually match the type's behaviour. Because the programmer's time is valuable, it is desirable to provide her with helpful assistance to keep this task as simple as possible, or better, to automate it completely.

2.7 Related work

There has been a lot of research on typestate analysis in the last two decades. The concept of typestate was initially introduced for pointers in Pascal programs in 1986 [SY86], where it was used to detect potential null pointer errors and avoid memory leaks. It could only differentiate between two basic states: $\bot$, meaning that a pointer is not initialized, or $\top$, if a pointer was initialized and assigned to a valid memory location. This simplifies the generation of specification drastically, as there is only one basic specification for pointers to basic types. Typestate for combined types (records, variants) can then be recursively generated by individually considering the typestates of the contained fields. The basic specification can easily be integrated into the analysis as it is fixed, and the derivation for combined types can be performed automatically. Because the analysis is integrated into the compiler, it can directly work on its internal representation, like the ***abstract syntax tree*** of the program. This is quite elegant, because a programmer automatically profits from the analysis by only using a different compiler.

Since then, implementations for different platforms, e.g. .NET [DF04], compiled machine code [XRM00] or Java [FYD+08, War, GYF06], have been written. The objectives of the implementations vary between the two extremes of either giving most precise results or being fast. Some approaches impose strong restrictions on their input programs to simplify the analysis, like limiting the depth of inter-object references [FGRY03]. This of course limits their general utility.

A promising implementation of scalable and sound typestate verification for Java, which also considers aliasing, was presented by Fink et al. in 2006 and revised in 2008 [FYD+08]. Their analysis actually consists of four stages with increasing precision. The first stage is intraprocedural and flow- as well as context-insensitive, and it only checks if some of all operations performed on an object could possibly be conflicting. If not (for example, all called methods are enabled in every state), this particular object does not need to be considered in the later, more precise and expensive stages. The last stage is an interprocedural, flow- and context-sensitive verifier, which combines typestate and aliasing information for very high precision. This stage is required for objects which are referenced by multiple methods, stored in collections or cross loop boundaries. The analysis does not impose considerable restrictions on the programs under test, except for the commonly excluded Java reflections and concurrency features. The achieved execution speed of between one up to ten minutes for

projects consisting of 200 and more classes is impressive, especially for the precision of the analysis, but still not bearable for continuous background typestate checking. In addition, Fink et al. only checked for very few typestate properties, and it is not explicitly mentioned how further properties could be added. We assume that increasing the amount of checked properties could noticeably affect the runtime, as it would need maintaining more data for more types. Therefore, we try a different approach which trades off analysis precision for speed and flexibility.

To our knowledge, currently existing typestate verifiers all either have the user create the needed typestate verification by hand (for example, as finite state automata encoded into source code annotations), or they come equipped with a fixed set of verifiable properties. In this thesis, we will present a verifier which is designed to take finite state automata as a separate input, making them completely independent from the sources or binaries of their target types. This interface enables us to use automatically mined object usage models [DLWZ06] as specification, passing a considerable part of this task over to the machine.

3 Program representation

In this section, we present the terms in which the behaviour of a program is usually described. We then present the intermediate compiler research language Firm [LBBG05] which displays the required features as ***explicit dependency graphs*** [Tra01]. After outlining their structure, we briefly discuss their usability for our analysis purposes. performance of an analysis

3.1 Basic concepts

To analyse a program's structure, it is necessary to have access to some of its features. The basic ones required for typestate analysis are the types of the objects created and the signatures of all methods ever invoked. Those allow for a flow-, path- and context-insensitive typestate analysis. The precision is increased if the analysis also takes the following aspects into account [Tra01]:

Control flow gives an ordering for the instructions in a program. As programs can jump between operations, this order usually is not strict or even total. It also relates the results of branching conditions to their possible jump targets, e.g. for conditional branch instructions, `select`/`switch` statements and exceptional `return`s. It is required for flow- and path-sensitivity.

Data flow provides information about where data is created in the program, and by which operations it is used. If an operation o_{src} creates some data element which another operation o_{user} requires for execution, o_{user} has a ***real data dependency*** [Tra01] on o_{src}. Information about data flow is necessary for flow-sensitivity.

Type hierarchy describes the inheritance relations between the types for languages supporting type inheritance. It is essential to correctly locate the affected member fields and methods for respective accesses. The type hierarchy is also needed to derive dynamic type information, which is required to handle language features like subtyping and interfaces.

Call hierarchy gives all locations from which a method is called, as well as all methods called from a location. There are different levels of detail, and they depend on the definition of *location*. Simple call hierarchies just represent whole methods, while more detailed ones may display their instructions. The call hierarchy is needed to disambiguate calls to overwritten or supertype methods and thus for context-sensitivity. It can be reconstructed from the control flow and the type hierarchy if not given.

It is important to provide this information as compactly and efficiently as possible for high analysis performance. If it is not explicitly available it must first be extracted from some other source prior to the actual analysis. This can introduce considerable overhead, especially if the source also contains a lot of additional information irrelevant to the analysis.

Static analyses which are integrated into compilers can simply use the internal representations built during the compilation. Stand-alone implementations usually either read the program's source code or its compiled binaries to build the required data structures. Source code must first be parsed to build the abstract syntax tree (AST), from which other required structures can then be

constructed. Parsing is not necessary for compiled binaries, as they already contain instructions in the required execution order. In addition, they will include performed compiler optimizations in the analysis – which would be invisible in the source code – and possibly contain fewer language artefacts. On the other hand, variable names or branch target labels may have been replaced by addresses, which can complicate giving informative feedback to the programmer.

We want to introduce the aforementioned features to illustrate the analysis' perspective of its target program.

3.1.1 The control flow graph

The control flow of the program is usually visualized as a graph (B, F, s, e) with

- a node $b \in B$ for every ***basic block*** in the program,
- edges $(b, b_{succ}) \in F = B \times B$,modeling the ***flow*** from b to its ***successor*** b_{succ}, and
- two special start/end blocks s,$e \in B$. [SPS99]

Basic blocks contain a set of program instructions which are strictly ordered according to their execution sequence. Usually, this order is also total and only allows this exact sequence, but there are representations where this is not required and which can express that two operations do not depend on each other (see Section 3.2). The first instruction in a basic block is always a branch target to enable jumping into the block, and the last one must be a branch instruction which jumps into one of its successors. Branching instructions are forbidden in any other position in basic blocks. Additionally, as e marks the end of the graph, it does not contain an outgoing branch instruction.

On execution, the program would begin with the first node of the start block s and traverse the instructions in the order given in the basic block. The control flow graph (CFG) is cyclic if the program contains loops, in which case there will be infinite paths. In any other case, paths are finite and must end at the e block. If a block $a \in B$ lies on all valid paths from s to a Block $b \in B$, we say that a ***dominates*** [Tra01] b.

A basic block can have multiple successors, in which case the last instruction must be a conditional branch choosing one of the outgoing edges. This connection must be encoded into the CFG for path-sensitivity. A possible solution would be to extend the domain of the edges to $F = B \times B \times V$ with V being the (probably infinite) set of all possible values. For boolean expressions, those are only `true` and `false` whereas `switch` statements can return a finite but unlimited amount of different values. An example CFG is shown in Figure 3.1.

3.1.2 Dataflow with variables

Values in programs are either directly passed from one operation to another, or stored in and retrieved from variables. As a variable v can be defined multiple times, it may contain different values at different points in the program. For any location l, the variable v always contains the value of its most recent definition, which ***kills*** all previous ones. To trace the flow of data in a program, one must find every such definition for all variable uses in a program (***Reaching Definitions Problem*** [NNH04]), a prime example for the application of dataflow analysis. The flow-sensitive but path-insensitive version works as follows:

```
public int faculty(int i) {
  int fac = 1;
  while (i > 0) {
    fac = fac * i;
    i--;
  }
  return fac;
}
```

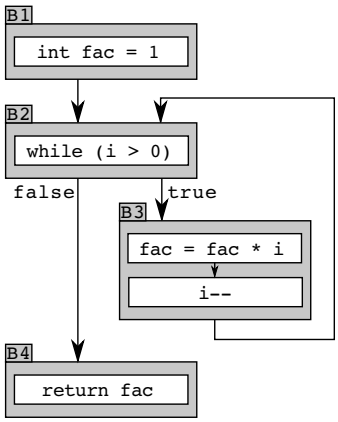


Figure 3.1: A Java code snippet and the corresponding control flow graph (CFG) with $s = B1$ and $e = B4$.

- The required graph (N, E) is obtained by taking the CFG and transforming every instruction inside a basic block into a single node. The edges between the basic blocks are mapped to their respective last and first instruction nodes. New edges are introduced to connect the new nodes in flow direction.
- We define the finite set of all variable names in the program as V. A definition of a variable $v \in V$ by the instruction of node $n \in N$ is identified as (v, n). Thus, we define the domain for the input/output sets as $2^{(V \times N)}$.
- The transfer function is simple: The resulting output set is equal to the input set, except for the case that the current node defines a variable v. In this case, the new definition replaces all previous ones and all entries $(v, \_)$ are removed. The only remaining definition for v in the output set is (v, n). An undefined variable u can be either expressed by not giving any entry for u or the special entry $(u, ?)$. The former is usually done for real implementations because it saves memory, but the latter is formally more consistent.
- The merge operation $\sqcup$ is the set union operation $\cup$: When two different definitions (v, n_1) and (v, n_2) reach an instruction n, the path-insensitive analysis cannot distinguish them and thus must assume both definitions to be ***live*** from that point on.
- The analysis starts at the first instruction of the program, which has the input set $((v, ?)|v \in V)$ because all variables are undefined at first. When the fixpoint has been reached, every node is annotated with all potential variable definitions live at that point.

The algorithm is easily extended to be path- and context-sensitive by adding the path and calling context to the value domain. Special care must be taken however to guarantee fixpoint convergence for programs containing infinite paths (e.g. loops, recursion).

	Code	Output sets $\subseteq 2^{(V \times N)}$
1	`Object o,u,v,w;`	`{ }`
2	`o = new Object();`	`{(o,2) }`
3	`u = o;`	`{(o,2),(u,3) }`
4	`v = u;`	`{(o,2),(u,3),(v,4) }`
5	`w = v;`	`{(o,2),(u,3),(v,4),(w,5)}`
6	`v = new Object();`	`{(o,2),(u,3),(w,5),(v,6)}`
7	`w.someOp(v);`	`{(o,2),(u,3),(w,5),(v,6)}`

Figure 3.2: Java code snippet with output set of each line for the Reaching Definitions Problem. After execution of line 6, the first definition of `v` is overwritten. To find the allocation site referenced by the use of `w` in line 7, four lookups are needed; `v` only needs one lookup.

As mentioned earlier, abstract objects are identified with the allocation site at which they were created. To find the object which is affected by an operation *op* on a variable v, the analysis must know the allocation site which is referenced by v. It can be found by looking up the instruction which defines v, using the information from the dataflow analysis.

This is often used directly as representation for the dataflow. But it has a conceptional flaw if used for typestate analysis: It gives definitions and uses for variables, but not objects. Although variables can be used to locate objects as described above, the implicitness of this information is impractical. If a variable is defined using other variables, finding the affected allocation site means walking up the whole definition chain. This is unnecessarily expensive, as illustrated in Figure 3.2. We would favour a different representation, where operations are directly connected to the affected objects, making the data dependencies explicit.

3.2 Explicit dependencies with jFirm

Explicit data dependency graphs [Tra01] (EDG) combine control and data flow and model the dependencies as graph edges. We use ***jFirm***, a Java implementation of Firm, to acquire EDGs for Java programs. In this subsection, which is mostly based on the dissertation of Martin Trapp [Tra01], we describe their structure together with the Firm language.

3.2.1 Language structure

As an intermediate language, Firm defines its own instruction set. The transformation from Java bytecode (or other languages supported by different implementations of Firm) to this language hides syntactical features which are irrelevant to the program's behaviour. In addition, Firm does not contain instructions for explicit memory management because it is independent of any architecture. As it is intended to be usable for analysis and optimization, every instruction performs a single, atomic operation. The absence of combined instructions allows the detection of instruction level parallelism and reduces the required amount of rules in analyses.

A distinct feature from the previously discussed representations is the non-existence of variables. In Firm, operations are directly connected to the outputs of their required operands and thus render the concept of variables obsolete. The number of inputs varies depending on the operation and its actual use, but all operations have one reserved input for connecting it to its containing ***block***. Similar to basic blocks in control flow, Firm blocks group instructions which must be always executed together. Additionally, a block may not be left before its instructions have been executed. Apart from that, they do not impose any further restrictions. As a result, Firm blocks only end when the control flow either splits, or when it reaches locations with more than one control flow predecessor. The only constraints on the execution order inside a block are set by the explicit data dependencies between the instructions.

Firm distinguishes between ***data*** and ***jump*** instructions. While the former generate data as result of their execution, the latter represent (un-)conditional branches. Conditional jumps are dependent on their input, but they cannot be used as operands for instructions; only blocks can specify control flow dependencies to them. Because all operations in a block must be carried out together, jumps must obey the following rules:

1. As jump instructions are only succeeded by blocks, all data instructions in a block must be executed before taking the jump.

2. For the same reason, there is only one jump instruction per block. If a block contained more than one jump, only one could be taken before leaving the block.

Note: This only holds if no operation inside the block can throw an exception. We address situations in which this is not the case in Section 3.2.5.

A block left by an unconditional branch therefore only has one successor. As conditional branch instructions have one output for each possible outcome of the condition, the respective block can have up to as many successors. It is legal to have different result values lead to the same successor (e.g. fall-through **`case`**s in **`switch`** statements).

In EDGs, the inputs of instructions and blocks are numbered. For instructions, input i is the i-th operand to the instruction, and for blocks it is the i-th control flow alternative, by which the block can be reached. If an instruction has more than one output, it generates ***projection nodes*** for each of them. Semantically, they represent one data value at the output on their parent instruction node.

3.2.2 Static single assignment form

The Firm representation of a program is in ***static single assignment*** (SSA) form. This means that a variable in a program is only defined once, and all following accesses see the same value. Redefinitions of the same variable get a different name and cannot overwrite the previous definitions. In Firm, this concept is applied to the result values of operations. Once an output value is defined, it will not change as long as the definition is live.

Figure 3.3 shows the EDG for the code snippet in Figure 3.2. Apparently, it does not contain any variable definitions. Instead, the call to the `someOp(Object)` method is directly connected to its two arguments: Input 1

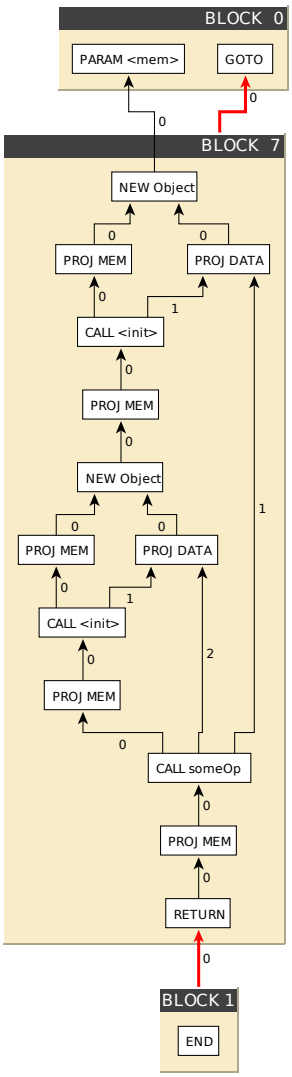


Figure 3.3: The explicit dependency graph jFirm generates for the code snippet in Figure 3.2. Operations with multiple outputs generate projection nodes which each represent one value. The invocation of `someOp` is directly connected to the data projection nodes of the **`new`** instructions, and all local variables have vanished.
The thick edges indicate control flow; the others are data dependencies. Their orientation is inverted with respect to the execution order of the commands, as they point from uses to definitions. The numbers give the index of the respective input of the using instructions.

(**`this`** reference, was w in the code) uses the first object created; input 3 (the first formal parameter, originally v) uses the second one. Both objects are created by **`new`** instructions.

3.2.3 Explicit modeling of side effects

The 0-th parameter called `MEM` represents the current state of the programs heap memory. Some operations – like function calls, field or array accesses – read or manipulate data in the heap and therefore have a data dependency on it. As they might change its state, they must return a new node `MEM'` representing the new memory version as required by the SSA constraints. The next operation, which uses `MEM'`, therefore depends on this new node and cannot be executed prior to the operation generating it. So by explicitly expressing the presence of side effects, the order of these operations is retained.

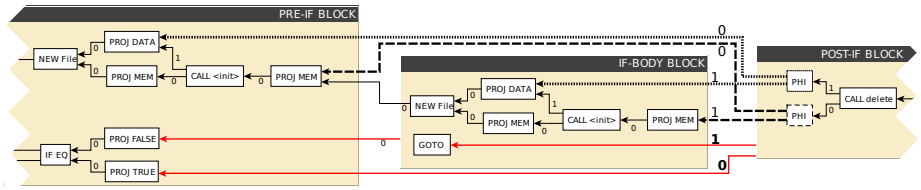


Figure 3.4: jFirm graph for the code snippet in Section 3.2.4. When control flow is joined after the conditional, a ϕ (`PHI`) operation selects the appropriate variable (thick dotted edges) and memory (thick dashed edges) definition which corresponds to the entry path into the block (bold numbers).

In the current state, jFirm is only capable of having one memory node represent the complete heap state. If a set of operations worked only on separate data in the heap, their relative execution order would not be relevant. The Firm language supports this by offering the possibility to create multiple memory objects for separate heap locations. If there finally is an operation which crosses the boundaries of the existing memories, a `SYNC` operation merges all needed memories.

3.2.4 Value selection

Although the SSA representation prevents redefinition of variables, there are many cases where multiple definitions of a variable can reach an operation. Consider the following example:

```
1 File f = new File("hello");
2 if (<condition>) {
3   f = new File("world");
4 }
5 f.delete();
```

Depending on whether the condition is true or not, the `delete()` operation will be linked to the file `world` or `hello`. On a real program execution, only one definition of the variable `f` can be live in line 5, which depends on the path taken before reaching this line. In Firm, this is modeled by ϕ (`PHI`) selections as shown in Figure 3.4. They have as many inputs as there are control flow predecessors for the block containing the ϕ. Each input is indexed with the number of the path to which it corresponds. Path-sensitive applications therefore still see exactly one definition of the variable at that time, while path-insensitive ones get all existing definitions by treating all of the ϕ's inputs as *potentially live*. The ϕ then represents the value selected from the different inputs. Thus, operations using such a variable take the ϕ itself as the appropriate input and still only "see" one single value. As the heap is treated as a variable in Firm, there are ϕ nodes for `MEM` as well. It is worth emphasising that ϕ only select from existing values and do not create any new ones. Moreover, they are a feature of the representation and thus do not represent program instructions.

The selection always takes place before the instructions in the corresponding blocks are executed. If conditionals or loops are nested, it is likely that ϕ have other ϕ as input. They can even be connected to themselves to express that a variable has not been redefined on a specific path. All ϕ update simultaneously, meaning that every ϕ represents the old value while the new selection is performed, and only update to the new one after all selections have been performed.

For consistency with the graphical representation we will refer to ϕ as `PHI` for the remainder of this thesis.

3.2.5 Exceptions

In Java exceptions can be handled explicitly and therefore do not cause the immediate termination of the program. For program analysis purposes, we need to be able to identify where exceptions can occur, if and how they are handled, and where the program execution is continued. It is not trivial to represent this information without unnecessarily blowing up the control flow graph, which makes analyses imprecise [Tra01].

Firm supports such explicit exception handling in the target language using two primitives:

Secured blocks Parts of the program with defined exception handling are enclosed in special secured blocks (Java: **`try`**).

Exception handlers If an exception is thrown inside a secured block, control switches to a specific block of the EDG (Java: **`catch`**).

The beginning and the end of a secure block is indicated by `BBegin` and `BEnd` nodes, respectively. They read `MEM` and define a new one, which is then read by the first instruction in the secured block. They also define an abstract `Exception` variable which we can ignore for our purposes, as it only ensures that instructions cannot be reordered to lie outside the secured block. Instructions which can possibly throw an exception form a third type of nodes which we call ***data-ex***. It is basically a combination of both data and jump instructions: If the operation executes normally, it is just treated as a normal data node and the next operation is processed. In the exceptional case, it directly jumps to the exception handler block. Note that, in contradiction to the jump node rules mentioned in Section 3.2.1, there can be multiple data-ex nodes in one block. In the exceptional case, some operations in the current block will therefore be skipped if exception handling is invoked before their execution.

The exception handler must know the memory state when it is invoked. At the beginning of its block, the outgoing `MEM'` of all data-ex nodes leading to this handler are combined into a single `PHI` which chooses the one corresponding to the entry path taken. After the execution of the handler, control flow always jumps to the successor of the `BEnd` node of the secured block. Figure 3.5 gives an example of a secured block containing two data-ex instructions and a one-instruction exception handler. Due to the memory selection at the beginning of the exception handler, analyses can also handle exceptions path-sensitively without losing precision, as previous representations often required merging of all handler entry paths [Tra01].

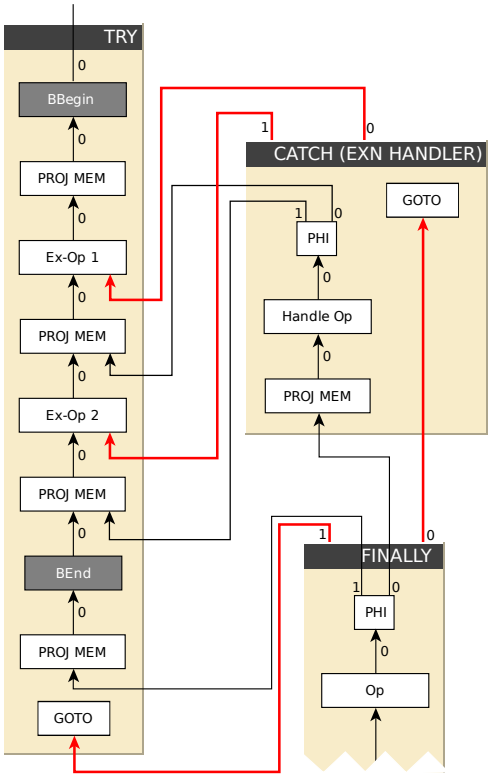


Figure 3.5: Example for exception handling in Firm. The secured block (**`try`**) contains two operations possibly throwing an exception. The exception handler (**`catch`**) selects the appropriate `MEM` and performs a single operation before jumping to the successor of the secured block (**`finally`**). If no exception occurs, the secured block is processed normally and also jumps into the **`finally`** block.

As Java supports different types of exceptions, the handler must also be provided with information about the exception which caused its invocation. The description of how this is realised in Firm is left quite vague as it is not explicitly described for any specific language [Tra01], and in the current version of jFirm, exception handling unfortunately is not implemented yet. One imaginable solution would be adding an output to every data-ex node representing the corresponding exception when the handler is invoked, and which is undefined in the non-exceptional case.

3.3 Expected benefits and problems

In this subsection, we explain the benefits of using jFirm as program representation for jFTA. The high suitability of EDGs for dataflow analysis drastically simplifies the design of the typestate analysis algorithm, as will become apparent in Section 5. In addition, we expect jFirm to become a platform for other static analyses as well, which then can easily be combined with our analysis because of the shared basis. A first example for this is the points-to analysis library we used

for our implementation of the algorithm which we describe in Section 6. Finally, it saved us the time needed for developing our own intermediate representation, which would have been a very complex task itself.

Except for issues like missing functionality related to the early development stage of jFirm, we do not see any negative aspects in using it. When thinking about the effect on the analysis performance, it is possible that building the graphs takes a considerable amount of time. We believe jFirm to be implemented with strong emphasis on its efficiency and speed. As it also performs semantically equivalent graph simplifications which we do not investigate in this thesis, this extra work will probably pay off during the analysis phase. We expect that the ability of sharing graphs with the points-to library results in a noticeable relative speed-up of both analyses' set-up phases.

4 Specification as finite state automata

We have already mentioned that, in contrast to previous approaches, we want to use automatically mined specification as reference, against which our analysis checks the actual usage patterns found in programs. In this section, we explain the format of our specification, quickly outline the mining process using ADABU [DLWZ06] and discuss expected problems.

4.1 Definition

The finite state automaton (FSA) $spec_T = (S_T, T_T, \texttt{start}_\texttt{T}, \{\texttt{ex}_\texttt{T}\})$ which specifies the correct usage of instances t of type T is defined as follows:

- S_T is the finite set of all abstract states for this type.
- M_T denotes the set of all methods for T.
- $T_T : S_T \times M_T \rightharpoonup S_T$ defines the labeled edges giving all specified typestate transitions. As T_T is not necessarily total, some transitions may be undefined and must be treated separately in our algorithm.
- $\texttt{start}_\texttt{T} \in S_T$ denotes a special state representing the typestate of t directly after its creation, before any methods (including constructors) were called.
- $\texttt{ex}_\texttt{T} \in S_T$ is the only terminal state which expresses that the typestate of t is undefined as result of calling a disabled method.

For our analysis, the abstract start and error states of all types convey the same meaning. An instance in state $\texttt{start}_\texttt{T}$ is not initialised yet, but only its memory has been allocated. A constructor must be called to perform initialisation, which will take the instance to a different (now initialised) state. As the $\texttt{ex}_\texttt{T}$ state represents an illegal and/or undefined typestate, it can never be left again. Therefore, we only consider one single error state `ex` which replaces all $\texttt{ex}_\texttt{T}$. Transitions which take the instance to `ex` represent typestate violations. It is those transitions which we want to detect and report to the programmer.

4.1.1 Transitions and non-determinism

If an instance t is in state s_T and the method m_T is invoked, the outgoing edges of s_T labeled with m_T define the new typestate of t after the method invocation. A state can have multiple outgoing edges with the same label if the instance can end up in several different states after the invocation of a method. An example for this is the simplified specification for a `Stack` class in Figure 4.1. As transitions to multiple target states are non-deterministic in FSA, we must assume that the instance may be in any of the states after the transition is fired. The over-approximation performed by our analysis becomes very clear at this point: We consider the instance to be in multiple abstract states at the same time, but in real execution, it obviously only has one concrete state which is represented by a single abstract one.

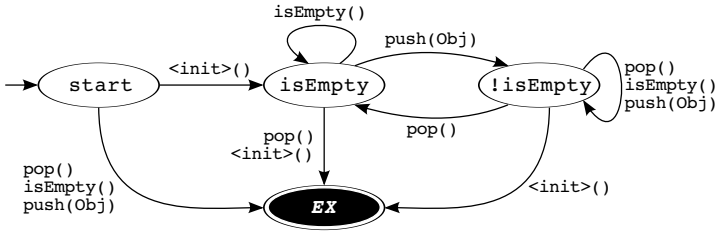


Figure 4.1: Example specification for a `Stack` Java class. If fired in the `!isEmpty` state, the `pop()` operation non-deterministically either stays in it or switches to `isEmpty`.

4.2 Automatic specification mining with ADABU

For most typestate analyses, the specification FSA are created by hand. As this can consume a lot of valuable time, we want to use automatically extracted usage models as specification automata. These models are acquired from existing programs which are believed to be using the types in question both correctly and intensively, such that the observed usage covers all valid cases. Otherwise, the generated specification can be too sparse and thus fail to capture the effects of rarely occuring but still valid use cases. As the analysis can only be as precise as the specification automata, its results will suffer badly if they are insufficient.

The models we will use are generated by ADABU [DLWZ06], a usage model miner for Java programs. First, it statically analyses the code base to find methods which only report the state of an object without altering it, called "inspectors". All other public methods are considered to be "mutators" which can change the state. The current state of an object is then defined as the set of all its inspectors' return values. Where applicable, ADABU abstracts from the real values through classification to keep the number of states and thus the returned models small. For example, integers are only classified as `>0`, `==0` or `<0` and booleans as either **`true`** or **`false`**. In the last step, the code base is instrumented to call all inspectors before and after each invocation of a mutator. If a combination of a certain pre-state, mutator call and post-state is observed several times, it will be added to the resulting usage model.

4.3 Expected benefits and problems

A positive aspect about using FSA to encode typestate specification is their small footprint. The information we need to represent – a set of states and connections between its elements - almost canonically defines the corresponding state machine with hardly any overhead. In most cases, only the outgoing transitions of the current state are of interest which perfectly matches the structure of the FSA. In addition, this is a very simple interface for providing specification and therefore models from many kinds of sources can be used.

Mining such FSA using ADABU obviously takes a time-consuming task away from the programmer, which is one of our main goals. However, she is presented with a different problem, namely providing a suitable code base for model extraction. This is rather easy for common types like `java.util.Vector`, `java.util.List` or `java.io.File`. With decreasing popularity of the type,

finding appropriate and valid example programs becomes harder; and the automatic mining is finally impossible for custom types which are only used in the program which is just developed. If analysis of their use is desired as well, the programmer must create the specification by hand. On the positive side, specification could easily be included into a "specification package" once created, and distributed along with the analysis due to the disconnection from the program code under test.

Apart from the problem of acquiring the specification, we also see some issues with the quality or expressiveness of mined models. As they are built only based on observed behaviour, they do not have any connection to the actual natural language specification from the documentation, and we have already pointed out that they can be sparse and miss specified but rarely observed behaviour. Even worse, forbidden behaviour could be captured which, by chance, did not cause any failure in the mining runs. The generated specification should therefore be at least manually checked before use.

4.3.1 Information loss through over-abstraction

Another problem arises from the abstraction in the concept of typestate and specification, because it introduces non-determinism in the models. This becomes visible when considering the `Stack` specification in Figure 4.1 again. As it only classifies the stack as being empty or not, it is impossible to distinguish between stacks containing only one or hundreds of elements. For this reason, the `pop()` operation in `!isEmpty` is non-deterministic: It stays at this state to account for stacks with more than one element, but it also switches to `isEmpty` for those with only one. Two subsequent *pop*()s will therefore always be reported as a problem, as the first one reports the stack as possibly empty which makes the second one illegal. Problems like this will occur for every type whose states include counter-like values, and we expect this to cause a considerable amount of imprecision. To a limited degree, the impact of this restriction could be lowered by "unrolling" counting states $k \in \mathbb{N}$ times, which could be achieved by replacing the `>0` class by `1`, `2`, ..., `k` and `>k` in the ADABU abstraction rules (similar for `<0`). However, the models will grow exponentially with k in the worst case due to the increased amount of states and combinations. We will not address this issue in this thesis, but it will be important future work.

Currently, ADABU does not extract ***deep models*** [DLWZ06], meaning that it does not analyse the state of objects returned by the inspector methods. This is problematic if a type's state is only defined through the states of objects stored in its member fields, because ADABU would then be unable to detect any state transitions. However, many complex types provide a `getState()` method or similar which returns an enum representing the current state. As ADABU does not abstract from enums but identifies their concrete values, it could distinguish the states via the enum values. Unfortunately, this fails for types returning integer state-constants instead of enums for the reasons discussed above.

5 A dataflow analysis algorithm

In this section, we present an algorithm for intraprocedural, flow-sensitive, path-insensitive and – through some interprocedural extensions – partly context-sensitive typestate analysis. As input, it requires a set of specification FSA and the jFirm graphs of all classes relevant for the analysis. This includes classes which might be referenced by interprocedurally analysed method calls, field accesses and the like. We require the existence of two functions $aliases(node)$ and $pointsto(node)$. The former returns all nodes which possibly refer to the same abstract object as the argument, and the latter returns the set of allocation sites the argument node might refer to. The actual implementation of these will be discussed in Section 6.

Our algorithm performs an iterative dataflow analysis. We can directly run it on jFirm graphs and annotate its blocks with pre/post versions of the gathered information. Remember that we want to find out which state objects can have at a certain point in the program. We need to define a few names and symbols for this:

Value domain V
: Contains mappings from Firm elements (graph nodes or field identifiers) with added context information to sets of typestates.

Context
: A snapshot of the call stack as the attached Firm element was encountered.

$\bot$ **,** $\top$
: The empty set and the set of all possible mappings, respectively.

$\top_{typename}$
: The set of all typestates valid for an object of type *typename*

Transfer function $\tau : B \times V \to V$
: Contains the analysis logic. It takes a block and its input set and gives its resulting output set.

Merge operation $\sqcup$
: Needed at joining points in the analysis. It performs a merge of all mappings in its inputs.

Partial order for V: $\sqsubseteq$
: For $a, b \in V$, $a \sqsubseteq b$ iff for every mapping $((context, element), states)$ in a there is a mapping $((context, element), states')$ in b with $states \subseteq states'$.

In the remainder of this thesis, we will use the short form $element_c$ instead of $(c, element)$ to identify an element with attached context information.

5.1 Allocation sites and references

During analysis, an abstract object is identified by a Firm node *alloc*, which represents the allocation site creating the object. If this object has typestate *tstate* at one point in the program, the corresponding value set contains the pair $(alloc_c, \{tstate\})$. If the set mapped to $alloc_c$ contains multiple entries, the object could be in any one of the contained typestates at this point. The least

precise information for an object of type *type* is given by $(alloc_c, \top_{type})$, while $(alloc_c, \emptyset)$ expresses that no valid state is known for $alloc_c$. The ***state of an object*** is therefore defined as the set of all its currently mapped typestates. If we want to talk about *single* elements of an object's state, we refer to them explicitly as ***typestates***.

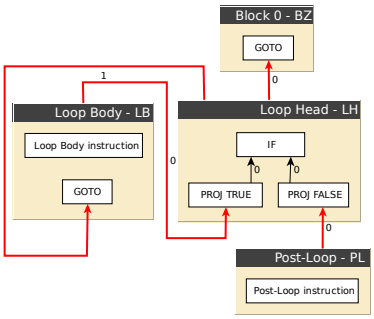
We do not only store states for allocation sites. As mentioned in Section 3.2.4, multiple definitions of a variable still exist even in SSA program representations (`PHI` nodes). To improve analysis precision, we also introduce ***references*** to model the effects of multiple variable definitions. Unlike allocations sites, these do not represent distinct abstract objects, but group already existing ones together. By assigning typestate information to a reference, it is possible to accurately reason about the state of *the one* object from the group that actually reaches the current location. This is comparable to the $focus$-operation introduced by Fink et al [FYD+08]. References are therefore always connected to a set of allocation sites with whom they also share state updates. We discuss this in detail in Section 5.3. During analysis, references are used at all places where a variable cannot be replaced by a dependency edge because it has multiple live definitions at this point. So we use references to store the state of `PHI` nodes as well as static and member fields.

5.2 Graph traversal

As typestate is defined as the states of an object's member fields, the algorithm must identify operations which might have effects on the memory state and consider them in the right order for flow-sensitivity. In terms of Firm, these are all operations which read and/or produce a `MEM` node. All other operations cannot have side effects and can thus be safely ignored (e.g. arithmetic operations, branch instructions). Remember that operations with side effects are ordered by their dependencies on the `MEM` variable (cf. Section 3.2.3). Therefore, the transfer function τ must follow the chain of `MEM` definitions, beginning at the user of the first definition. Irrelevant operations are automatically ignored, as they will neither use nor produce a `MEM` and thus will never be seen by τ.

The analysis is performed top down, beginning with the designated first block (later referenced to as "block zero") of the method's jFirm graph and with all typestate information set to $\top$. It represents the context from which the method was called: There is a `PARAM` node for every method parameter, a `CONST` node for each constant used in the entire method and one `MEM` node representing the heap state prior to the method invocation. Apart from those, block zero only contains an unconditional branch with exactly one control flow successor, which is the block containing the first real instruction.

Block zero is jFirm's solution for modeling the "world" in which the method is executed. The fact that jFirm builds intraprocedural EDGs makes every method graph have its own block zero. If the program was represented interprocedurally by one single (potentially huge) graph, the methods' `PARAM` nodes would be replaced by dependency edges and their first instructions would depend on the last `MEM` created prior to their invocation. There would only be one block zero for the whole program containing the `argv` parameter of the `main` method, all constants used in the program and the very first `MEM` (representing the empty heap).



	Worklist — block — Effect on input sets
1	`[BZ] -> BZ -> init LH`
2	`[LH] -> LH -> init PL,BL`
3	`[PL,LB] -> PL -> nothing`
4	`[LB] -> LB -> change LH`
5	`[LH] -> LH -> change PL,LB`
6	`[PL,LB] -> PL -> nothing`
7	`[LB] -> LB -> nothing`
8	`[] fixpoint reached`

Figure 5.1: An extremely simplified example demonstrating the worklist algorithm. The order shown here is an example for non-optimal successor selection, because the `PL` block (including all of its possible successors) is visited twice.

The traversal of the Firm graph is realized as a worklist algorithm. Initially, the worklist only contains block zero. The following steps are then repeated as long as the list is not empty:

1. Get next block b from the worklist
2. Set the output set of b to the transfer function's result: $out(b) := \tau(b)$
3. For every control flow successor b' of the branch instruction of b:
   (a) Propagate the values from the the output set of b to the input set of b': $in(b') := in(b') \sqcup out(b)$
   (b) Add b' to the worklist **only** if this changed $in(b')$.

The condition for adding a block to the worklist in step 3b is needed for detection of reaching the fixpoint of the algorithm, as described in Section 2.3. If $c = a \sqcup b$, it will always hold that $a \sqsubseteq c$ and $b \sqsubseteq c$. Thus, the input set of a block can never become smaller (with respect to $\sqsubseteq$) as a result of a merge operation. It cannot grow infinitely large either, because the sets of typestates, Firm nodes and contexts are finite – under the assumption that the program's call graph does not contain paths of infinite length (e.g. recursion). As this would be an unacceptable limitation, we limit the maximal context depth to a fixed number k. We discuss the effect of this limitation in Section 5.7.4. With this setup, the merge operations must eventually reach a fixpoint at which the resulting mappings are equal to the previously stored ones.

Note that the order in which the control flow successors are added to the list is not specified. Although it does not affect the correctness of the algorithm, this can be a performance issue. If a program contains a loop, the block containing the loop condition has two successors: One representing the branch into the loop body and the other one branching to the post-loop code. If the code following the loop is analysed first, the whole remaining program will be processed without considering the effects of the loop instructions. When the loop body block then is eventually analysed, it can cause a change affecting the entry set of the post-loop block, which is then put back to the worklist. So it and all subsequent blocks are re-evaluated (see Figure 5.1). Considering that, as mentioned in

the previous paragraph, an input set can never lose information as result of an update operation, all information in the first analysis iteration is also contained in the second one. Therefore, the first one could be skipped without affecting the correctness. So the loop body should be analysed before the post-loop block. Information giving an order for the execution of successor blocks can be gained by running a loop detection algorithm prior to the actual analysis. One example for this would be the loop detection algorithm of Robert Tarjan [Tar73]. We will not investigate this any further at this point.

5.3 Typestate manipulation

Some instructions in Java will cause an object to change its typestate, and these changes must be simulated by the transfer function. Depending on the situation in which typestate information is updated, this task is of varying complexity. Care must be taken when determining which abstract objects are affected by the update, because variable aliasing must be resolved and reference targets must be found if not updating allocation sites directly. Furthermore, the precision of the update operation may suffer from the existence of multiple targets. Depending on such circumstances, a state update can either overwrite or only add typestates to previous state information. While the former can sharpen the known information by reducing the amount of possible typestates, the latter usually blurs it (i.e. can only add but not remove any states). We speak of ***strong*** and ***weak*** updates, respectively.

An update consists of two main steps. The first one performs the requested update operation on the object itself. Depending on whether a strong or weak update must be performed, previously known state information is either merged with or overwritten by the new one. In the second step points-to information and potential aliasing is handled: a reference is only a place-holder for one out of the set of allocation sites it points to. Therefore, updating a reference effectively means updating one of its targets in real program execution. This one target is obviously known if the reference just points to a single allocation site, which enables us to perform a strong update on it. This knowledge is missing if the points-to set contains two or more elements, which is why the update must be propagated to all of them and must only be performed weakly.

Resolving reference aliasing means updating all references pointing to the same allocation site whenever this site's state is changed – either directly or as result of a reference update. In this case, the exact object causing the change in those references is the allocation site whose state was just updated. So the affected references must not[1] propagate this update back to their other allocation sites again.

Our resulting update algorithm is given in Figure 5.2 as well as a simple example in Figure 5.3.

[1] Expressed so strongly because doing so would unnecessarily blur the results, even though it would not affect the correctness of the algorithm.

Algorithm: $update(node_c, state, strong/weak)$

1. If there is a previous mapping $(node_c, oldstate_{node_c})$:
   - **Strong update** Replace it with $(node_c, state)$.
   - **Weak update** Merge both into $(node_c, oldstate_{node_c} \cup state)$.

   If not, create a new one as $(node_c, state)$.

2. Determine if $node_c$ is an allocation site or a reference.
   - **Allocation site** For every reference $r_c \in alias(node_c)$:
     (a) If $pointsto(r_c) = node_c$ and a strong update was requested, replace any old mapping for r_c with $state$ (like the strong update in step 1).
     (b) In any other case, merge $state$ into the existing mapping for r_c.
   - **Reference**
     (a) If $pointsto(node_c)$ contains only one allocation site and strong update was requested, then recur on it with a strong update to $state$.
     (b) In all other cases, recur with weak update to $state$ for all $alloc_c \in pointsto(node_c)$ which will also handle updating all $ref \in alias(node_c)$.

Figure 5.2: The algorithm for performing a strong/weak update to *state* on the typestate for node $node_c$.

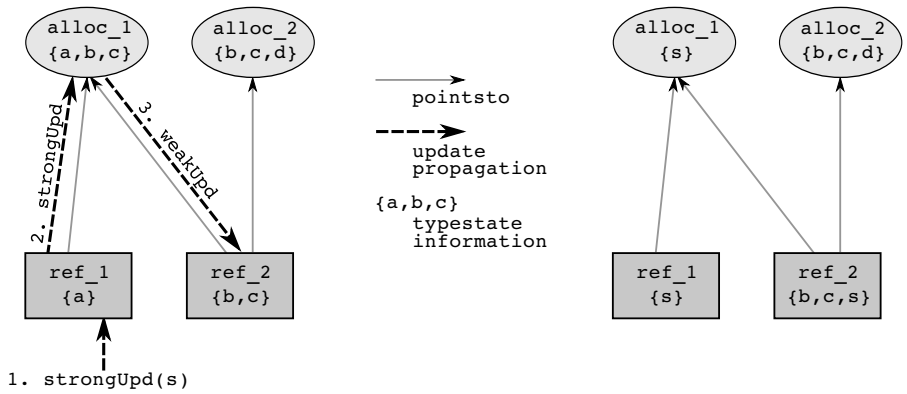


Figure 5.3: Simplified example of typestate update propagation.
Left: A strong update is triggered on reference ref_1 (1.). It only points to $alloc_1$ and thus propagates a strong update to it (2.). As ref_2 aliases ref_1 but also points to the different site $alloc_2$, it is only weakly updated (3.). Note that $alloc_2$ does not receive an update because ref_2 is updated indirectly.
Right: The typestate information after the update.

5.4 The transfer function

τ must be defined for every jFirm[2] node which can read or produce a `MEM`, because every one of those might be encountered while walking down the chain of memory definitions. Changes will be performed on a copy of the given input set to which we will refer to as the ***intermediate set***. This will be returned as the new output set after processing the last instruction in a block. We describe the various effects below, using the variable $node_c$ for the current jFirm node.

5.5 Starting at block zero

As already mentioned, block zero must be treated differently from the other, regular blocks. It contains nodes for parameters and constants, which are not part of the `MEM`-chain, and which must therefore be handled explicitly.

Constants (`CONST`) Constants can only be primitive values, `String`s or **`null`**. All of them are immutable; and as primitive values and **`null`** do not represent objects, typestate is not even defined. While `String`-constants are in fact objects, type and typestate are identical for them: On a `String` object, every method is enabled no matter which string it represents. The only kind of violation possible are exceptions thrown by some methods if invoked with illegal arguments. As those cannot be detected by our analysis, `String` constants are also ignored completely, and their state is simply defined as $\emptyset$.

Parameters (`PARAM`) For each method parameter, there is a `PARAM` node representing the referenced object. For the first method in the call hierarchy, those objects and their states are unknown, because there is no calling context from which this information could be taken. In this case, the `PARAM`s are treated as allocation sites, as from the analysis' perspective, they "generate" these unknown objects. Due to the pessimistic nature of the dataflow analysis, we must assume that an unknown parameter may be in every possible typestate. As a result, its state is stored as $(node_c, \top_{type})$ in the intermediate set. However, in interprocedural analysis, the calling context of most methods is available. In this case, the i-th `PARAM` node is substituted by the node connected to the i-th input of the `CALL` node belonging to the respective calling context.

After processing all nodes in block zero, τ returns the intermediate set which then contains all parameters. The state of all other objects (e.g. values of static and member fields) is still unknown. We could initialise them to $\top_{type}$ as well, because we can get a list of all fields of an object from jFirm, but this will only blow up the in- and output sets in the algorithm. We decide to store their states implicitly: Whenever a field is read and the intermediate set does not contain any information about it, the state $\top_{type}$ is assumed. Like this, the existence of fields is only discovered when they are first read or written. An entry for the field will explicitly be created the first time it is written.

`CONST` and `PARAM` nodes are not allowed in any other blocks. Control flow can never return to block zero from the method body, so τ will only find blocks which contain actual program instructions after the first step. Those always

[2]We explicitly refer to jFirm instead of Firm here because the former models the features which are specific to Java.

take a MEM node as input zero and generate a new one as output (PROJ mem). In the following subsections, only the inputs with indices > 0 are considered, such that speaking of the "first" input refers to input one and not zero.

5.6 Object creation

NEW nodes represent **new** statements in Java and thus are the source for objects. Every time this instruction is executed at runtime, a new object in the state $\texttt{start}_{\texttt{type(NEW)}}$ is created (cf. Section 4.1). Therefore, we classify $node_c$ as an allocation site and assign it the state start by adding the mapping $(node_c, \{\texttt{start}\})$ to the intermediate set. When the analysis encounters this NEW node for the first time this is straight forward.

It is more complicated if this allocation site is part of a loop, as its re-evaluation means that it creates another object distinct from the one(s) created earlier. For path-sensitive analyses, this distinctness must be maintained precisely as the results of *pointsto* and *alias* will depend on the current path. Instead of directly pointing to an allocation site, they will point to different instances stemming from it. As our algorithm is path-insensitive, it is not able to distinguish between single objects because it requires knowledge about the program paths taken. For this reason, it is sufficient to simply overwrite the allocation site's state with start but leave all references untouched. Like this, the state of objects previously created by this NEW node is still maintained in the references using them, but operations directly accessing the allocation site only see the last one created. Big imprecision is however caused by *alias*. As it cannot distinguish between users of the old and new objects, they will always influence each other, leading to a lot of weak updates.

5.7 Method invocation

The actual typestate transitions are fired by method invocations. If the target method *meth* is called in a non-static context, it fires a typestate transition on its first argument (**this**). The transition's effect on **this** is given by the typestate specification for the object's type, in terms of providing a list of all possible typestates reached by invoking *meth*. If the outcome of this invocation is unspecified, we can either interpret this as an error transition or assume that the state did not change at all. While the first policy leads to a lot of false positives, the latter might result in false negatives. Which one is selected should depend on the quality (i.e. completeness) of the specification and the desired degree of correctness.

5.7.1 Typestate violations

In the case that *meth* is disabled, the list of states will contain ex to indicate the illegal invocation (see Section 4.1). Depending on whether the transition led to typestates other than ex or not, we differentiate between ***may-*** and ***must-violations***.

A may-violation indicates that *meth* was disabled in some but not all source states. This can occur by merging multiple paths together, of which some fail to prepare $this_c$ for *meth*. The may-violation then in fact indicates a program flaw. However, it is also possible that, due to over-approximation in the analysis,

the source state contains typestates which would never be included during real execution. If only those typestates lead to `ex`, this is actually not a real problem but a ***false positive***. This motivates us to avoid over-approximation whenever possible, because it increases the amount of unneeded typestates in the analysis. The more false positives are returned, the less usable the analysis is for the programmer, because it means spending more time checking for errors in places where there are none.

Invoking *meth* on an object in $\top_{type}$ state will therefore always produce a may-violation if *meth* is disabled in at least one possible typestate. As a result, when a method is analysed, nearly every first invocation on an unknown parameter will produce a may-violation. For this reason, we added an extra rule for those cases which ignores may-violations resulting from objects being in $\top_{state}$ state. This rule obviously makes the algorithm unsound, so it can be deactivated if desired.

The situation is different for ***must-violations***. As the name suggests, a must-violation is encountered if the only state reached by a transition is the error state. If a must-violation is detected, we know for sure that *meth* is disabled as there is no alternative valid outcome of the transition. Thus, as long as the specification is correct, calling *meth* will result in a program failure, which is obviously always valuable information for the programmer. Must-violations cannot be false positives, as over-approximation might only degrade them to may-violations by adding superfluous states in which *meth* is enabled.

5.7.2 Fixing suggestions

When a violation is detected, the fix to the solution will likely be one of the following:

1. Prior to calling *meth*, call another method $meth_{pre}$ such that *meth* becomes enabled, meaning that $meth_{pre}$ is a prerequisite for *meth*. For example, `open()` is often a prerequisite for some `read()` method.

2. Instead of calling *meth*, a different method $meth_{alt}$ must be called. This implies that *meth* is simply the wrong method for this situation, as it would be the case for the second of two consecutive calls to `File.open()`.

Both kinds of fixes require the programmer to think about what other methods are available for the object, which of these are enabled in the current state, and what possible outcomes they have. This normally involves looking up the type's specification and identifying appropriate methods. This can be done by the analysis as well, because all of this information is encoded into the typestate specification of the type. So whenever a violation is generated, the analysis finds the set of all methods enabled for every typestate in the state set of $this_c$. If this set is not empty, it offers them to the programmer, while it rates solutions for the first case (methods enabling the initially requested *meth*) higher than those for the second one (methods simply not causing a violation).

To take the fuzziness of the analysis into account, the programmer is also presented with a third category of methods: Those which will cause a may- but not a must-violation. For the same reasons discussed earlier, such a method might always be enabled and never fire a runtime exception. However, this certainty has to be validated by the programmer, which is why suggestions

from this third category should always be selected with care. Another downside of this solution lies in the analysis always producing violations for it. To avoid this, the programmer should have the possibility to manually tune the analysis in such situations.

5.7.3 Explicit typestate definitions

We enable the programmer to actively manipulate the intermediate state set to help the analysis where needed. Superfluous violations can then be avoided by manually removing invalid states resulting from over-approximation. This is especially needed in cases where dynamic state checks are used to ensure typestate compliance because the analysis is unable to interpret them. For this, the programmer can insert special method calls into the target program resulting in `CALL` nodes which the analysis recognises as ***typestate directives***. Their first input is connected to the node obj_c whose state information is to be altered, and the second takes a `String`-constant giving the name *tstate* of a valid typestate for the object's type. τ identifies those special nodes by the class and name of the called method, which must be unique to prevent misinterpretation of real method invocations. Depending on the method's name, *tstate* will be added or removed from the state set of obj_c. In addition, the programmer can add directives to instruct the analysis to ignore may-violations for calls which she verified to only produce false positives. She must use two special *begin* and *end* directives which enclose the wanted `CALL` node as it cannot be used as a parameter due to Java language restrictions.

This extension is very similar to other typestate analysers where the whole specification is encoded into the programs using annotations. We cannot use the annotation framework for jFTA because jFirm graphs currently do not contain annotation information. There are both advantages and disadvantages to using method invocations instead of annotations. The main advantages are:

1. The parameters are sanity-checked because the compiler treats the directives as regular method calls. This means that the first parameter must be a valid variable/object reference, and it must have been initialized.

2. The directives are usable in Java code written for Java versions earlier than 5.0 which do not have built-in annotation support.

On the other hand, there are also considerable disadvantages compared to annotations:

1. On normal program execution, the directives are executed like normal methods. Although their bodies are empty, this could affect the programs performance if, for example, many directives are put into a loop with many iterations. Furthermore, method invocations manipulate the stack and could also affect the efficiency of caches.

2. Depending on the degree of optimization performed by the compiler, calls to empty methods might perhaps be dropped. Typestate directives would then be lost.

Independent from their actual implementation, typestate directives are static and do not automatically adjust themselves when the surrounding code changes.

Code snippet with `Typestate` directives	*state*(s) after execution of line

`/* s = Stack with unknown state */`
`while (!s.isEmpty()) {`
`  /* dynamic check not detected */`
`  Typestate.remove(s,"isEmpty");`
`  s.pop();`
`}`
`Typestate.remove(s,"!isEmpty");`
	`TOP = [!isEmpty,isEmpty]`
	`[!isEmpty,isEmpty]`
	`[!isEmpty,isEmpty]`
	`[!isEmpty]`
	`[!isEmpty,isEmpty]`
	`[!isEmpty,isEmpty]`
	`[isEmpty]`

Figure 5.4: An example for a situation where explicit typestate definitions are required. Without them, the `pop()` in the loop body would cause a may-violation. After the loop, the stack is guaranteed to be empty, which is ensured by another directive.

They should always be used with caution, because obsolete directives might unexpectedly remove or add typestates or suddenly address different objects in the case that a variable's name changed. An example situation where explicit typestate definitions considerably help the analysis is given in Figure 5.4.

5.7.4 Interprocedural context-switching

The typestate specification only gives information about the effect of the method on $this_c$. As static methods are not called on any object and we do not consider typestate for classes, the steps mentioned so far are all skipped for them. However, the effect on *meth*'s formal parameters must be reproduced for both types of methods. This is achieved by either analysing the body of the called method in the current typestate context (as given by the intermediate set) or by safe over-approximation. The former results in context-sensitive, interprocedural analysis, yielding better precision but also higher cost than intraprocedural estimation algorithms. To some extent, our algorithm performs interprocedural method analysis to achieve much better results. However, this can quickly become extremely expensive, because the length of the paths in a program's call graph is generally unlimited and can also be infinite in case of recursion. To avoid the problem of infinite descent one could perform loop detection on the call graph. Paths can then still be of arbitrary length with unpredictably high costs. We address both these problems by simply limiting the maximum path length to some comparably small number d_{max}. Effectively, this means that the simulated call stack cannot grow bigger than d_{max} during analysis.

If the call stack has depth $< d_{max}$ when τ finds a `CALL` *meth* node, it will perform an interprocedural ***context-switch*** to *meth* and analyse it using the state information from the current intermediate set. Considering the structure and semantics of jFirm graphs, this is fairly straightforward. As mentioned earlier, the `PARAM` nodes of *meth*'s graph can be replaced by dependency edges to the nodes connected to the respective inputs of the `CALL` node $node_c$. Apart from this, a context-switch is equivalent to reaching the end of the current block. As the parameters need not be analysed anymore, *meth*'s block zero is skipped and the analysis can directly continue at its first successor block. Afterwards, the return value (if any) of *meth* is treated similarly: If the return type of *meth* is non-**`void`**, $node_c$ produces a `PROJ data` node for it. As a method

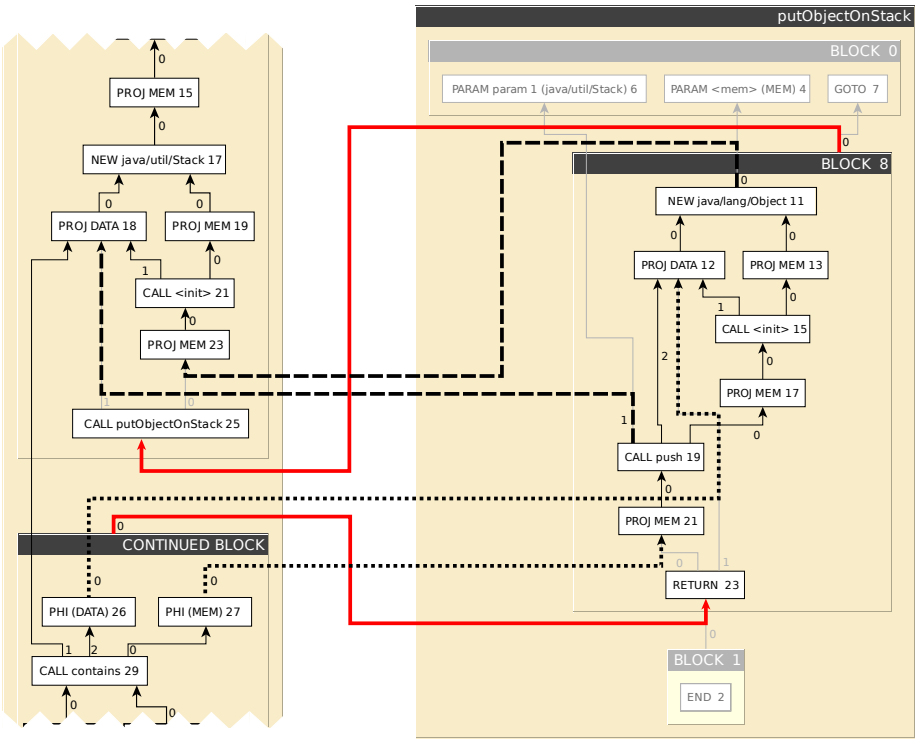


Figure 5.5: An example showing the modified jFirm graph when performing a context switch to a called method. Its PARAM-nodes are replaced by dependency edges (thick dashed edges), its returned value(s) are directly connected to the old projection nodes which are now used as PHI (thick dotted edges), and the block containing the CALL in the parent method has been split (new control flow by thick solid edges). Note that the PHI are only contained for clarity, as they would normally be replaced by dependency edges when they select from only one input. Greyed-out edges, blocks and nodes would be present if the graph was not modified, and are only given for comparison.

can have more than one RETURN instruction, there is possibly more than one value which is returned. So simply reconnecting all of the PROJ data node's users does not work. Instead, the node is treated as a PHI which is connected to all nodes referenced by a RETURN. Independent from the return value, the same is applied to the PROJ mem which then selects the appropriate memory state belonging to the RETURN instruction taken. When *meth* has been fully analysed, the analysis switches the context back to the next instruction in the parent method. A modified jFirm graph representing the changes is given in Figure 5.5.

In the case that the call stack has reached its maximum depth, no context-switch will be performed. Instead, as *meth* could arbitrarily manipulate the states of all its visible variables, the state information of all its parameters and return value is set to the respective $\top_{type}$. Furthermore, it is also necessary to

set the states of all member and static fields to $\top_{type}$ as well, because *meth* could also change their states either directly or by invoking other (static) methods. Obviously, a lot of information may be lost every time this happens, which is why we expect the setting of d_{max} has a strong impact on the quality of the analysis results.

5.7.5 Applying typestate changes

The new state for $this_c$ is applied last, overwriting possible state changes which occurred during the context switch. It is safe to do so because the specification ensures us that $this_c$ will definitely end up in this state after calling *meth* on it. Note that this also means that $this_c$ still has its old state during interprocedural analysis of the called method. We consider working with the new state nonsensical at this point because the sole invocation of *meth* does not change anything yet. It is the method's body which probably changes some of $this_c$'s fields and, as a consequence, its state.

The handling of `CALL` nodes is the core part of our algorithm. To sum up the ideas it incorporates, we present its description in Figure 5.6.

5.8 Object references: Fields and arrays

Apart from local variables, which can be replaced by dependency edges and `PHI` nodes, objects can also be referenced by fields or arrays. Both are different from local variables: The visibility of fields is not limited to the method itself and arrays can contain more than one object. While these properties do not have a direct effect on the state of the stored objects, it is important to correctly capture how object references flow through the program. We account for this by adding or modifying references accordingly.

We must differentiate between static and member fields. As the former belong to classes, there is only one instance of the field which exists throughout the whole program execution. On the contrary, member fields are defined, created and destroyed along with their respective parent object instance. As a consequence, fields cannot be identified only by their name: For static fields, the parent class must be specified; while member fields must specify the allocation site and context at which their parent object was created. Using these ***field identifiers***, we can add references carrying typestate information for fields to our analysis state.

5.8.1 Writing fields

Depending on whether the target field $field$ is static or not, `PUTSTATIC` or `PUTFIELD` nodes are generated to indicate a writing operation. For both types, the node itself contains the field name and its member class, as well as an input connected to the item to store in the field ($item_c$). The `PUTFIELD` has another input connected to the jFirm node determining its parent object ($target$). When τ encounters a put-operation, it gathers all allocation sites in $pointsto(item_c)$ and creates a reference connected to them with $state_f := state(item_c)$. If $field$ is static, there is exactly one instance of it. So a strong-update can always be performed by putting the mapping $((field_{name}, field_{class}), state_f)$ into the intermediate set.

Algorithm: $\tau(states_{interm}, \texttt{CALL}(meth, mem_c, param_{1,c}, \ldots, param_{n,c})\ )$

1. If *meth* is a non-static method: ($param_{1,c}$ is **`this`**)
   (a) Get the current state for $param_{1,c}$ from $states_{interm}$, which is $state_{old}$.
   (b) Prepare a new empty set $state_{new}$.
   (c) For every $tstate \in state_{old}$:
      - Add all typestates reachable from *tstate* by invocation of *meth* (as given by the specification for the type of $param_{1,c}$) to $state_{new}$.
   (d) If $state_{new}$ contains the error state:
      i. Generate a violation giving $state_{old}$, $state_{new}$ and *meth*.
         - If $state_{new}$ contains no other state, mark it as a must-violation.
         - If it contains other (non-error) states, it is only a may-violation.
      ii. Generate propositions to avoid the violation, if possible.
2. Consider the effect of invoking *meth* for all $param_{i,c}, i \geq 2$ and global variables in mem_c.
   - If the maximum call stack depth has not yet been reached and the jFirm graph for *meth* is available:
     (a) Acquire the jFirm graph for *meth*.
     (b) Link each of its `PARAM` i nodes to the respective $param_{i,c}$.
     (c) Recursively analyse *meth* using $states_{interm}$, starting at the successor block of its block zero.
     (d) Transform the `PROJ mem` node of $node_c$ into a `PHI` selecting from all memories used by a `RETURN` in *meth*'s jFirm graph.
     (e) If *meth* returns a non-void value, turn the `PROJ data` node of $node_c$ into a `PHI` selecting from all values used by a `RETURN` in *meth*'s jFirm graph.
   - If the maximum depth is reached or the jFirm graph is not available:
     (a) Set the states of all $param_{i,c}$ to $\{\top_{type(param_{i,c})}\}$.
     (b) Similarly, set the states of all fields to their respective $\top_{type}$, too.
     (c) If *meth* returns a non-void value, create an allocation site for the `PROJ data` node of $node_c$ and set its type to $\top_{type(meth)}$.
3. If *meth* is a non-static method, apply the new state for the **`this`** parameter: Put $(param_{1,c}, state_{new})$ into $states_{interm}$ (overwriting any changes from step 2).

Figure 5.6: Complete algorithm for handling `CALL` instructions in jFirm.

In case of PUTFIELD, this cannot be assumed, as *target* could refer to more than one object. Therefore, in addition to creating the reference as in the static case, τ must also get all allocation sites $alloc_c \in pointsto(target)$ and update $field$ for each one separately. At this point, we face the same problem as for PHI nodes: Of all candidate fields, only one will actually be updated in an actual program run, while all others remain untouched. Again, this candidate cannot be determined, allowing only weak updates to be performed on the fields by merging their old state information $(field_{name}, alloc_c), state_{old})$ with $state_f$. Strong updates are only enabled if $pointsto(target)$ contains exactly one allocation site.

5.8.2 Reading fields

Similar to writing fields, there are two different jFirm instructions for reading static or member fields. Both types again carry the field's name, and only GETFIELD has an additional input to a *target* jFirm node. Both produce a PROJ data node for the value they read from the field. In the static case, τ just creates a new reference linked to the allocation sites pointed to by $field$, carrying the state currently stored for $field$ and assigns it to the PROJ data node. In Section 5.5, we defined the state of $field$ as $\top_{type(field)}$ if it has not been written prior to the first reading access. In the non-static case, all affected fields are determined the same way as for PUTFIELD, and the union of all their states is assigned to a reference linked to all participating allocation sites. This is then stored for the PROJ data.

With a small extension of our update algorithm from Figure 5.2 we can considerably improve the precision when working with references from GETSTATIC and GETFIELD. So far, the fields themselves are only updated by receiving updates from their respective allocation sites, even if an update is directly performed on a GETSTATIC's or GETFIELD's reference. As soon as more than one allocation site is connected to the field, it will only receive weak updates. We circumvent this problem by enhancing the returned references with extra connections to the fields themselves. In the case that the referenced field is unique (i.e. has only one potential parent object), the update algorithm will perform a strong update on it instead of waiting for the weak one to be propagated. This is always the case for static fields.

5.8.3 Creating, reading and writing arrays

The array-handling in our analysis is very simple because arrays are hard to handle precisely in static contexts. While in principle they work similarly to fields, they can store an arbitrary large amount of different objects. Our analysis cannot differentiate between those objects, as every read or write operation on arrays also contains an integer address for the position inside the array which the analysis ignores. Thus, the analysis can never know if an existing object is overwritten or not when a new one is added via ASTORE, or which item will be retrieved by an ALOAD. So all operations carried out on elements of the array can only produce weak updates. For this reason as well as the fact that we do not maintain states for arrays, τ does not keep extra typestate information for them. Instead, whenever an ALOAD is encountered which refers to an array $array_c$, it creates and returns a reference linked to all allocation sites in $pointsto(array_c)$

with the union of their states. As a consequence, NEWARRAY and ASTORE nodes which create arrays and store objects in them are simply ignored by τ.

5.9 Unsupported or ignored nodes

In this version of our algorithm, there are some jFirm nodes which do not have an effect when they are encountered. The reasons for this are either that they do not provide any relevant information for the analysis, or that the operations are not supported at the moment. We briefly present those nodes and explain the reasons for ignoring them.

- **Exceptions (THROW)** Although one of the advanced features of jFirm (see Section 3.2.5), our algorithm currently does not handle those nodes because they are not implemented in the version of jFirm used in this thesis. This will be one of the major improvements for future development of the algorithm.
- **Typecasts (CAST)** In its current version, the algorithm can only handle static type information. Therefore, type casts are completely ignored at the moment.
- **Return (RETURN)** As we resolve the passing of return values to the parent method by re-linking the nodes when context-switching back to the parent node (see Section 5.7.4), no action must be taken when τ finds a RETURN.
- **Memory synchronization(SYNC)** The Firm language specifies splitting of MEM nodes to represent distinct memory areas. Whenever an operation might work on areas which are split, they must be merged (synchronised). As support for this is not implemented in jFirm, we also ignore this feature for now. It will however be a valuable addition for future work, because it can limit the amount of necessary overestimation.
- **Memory-phi (PHI)** When multiple paths lead to the same jFirm block, the MEM chain will contain a memory-type PHI node. We account for those by performing the merge operation $\sqcup$ whenever we update the input set of a block. For this reason, τ just skips them and continues with the next node in the chain.

In this section, we presented the concepts of our analysis algorithm. In the following one, we present our implementation of the algorithm as a Java application named jFTA.

6 Implementation

In our tool jFTA, we implemented the algorithm discussed in Section 5 as a Java application. In this section, we want to describe the challenges and decisions taken during the implementation. After briefly presenting its interface, we focus on performance and correctness aspects, as speed is one of our main goals. Therefore, we talk in detail about how we handle points-to and aliasing information with support of external libraries [Boe09], and how we try to keep the introduced overhead small. Lastly, we discuss aspects of the algorithm which are not covered in jFTA yet, therefore affecting the performance and validity of the results.

6.1 Analysis interface

The core of jFTA can be executed from the command line. As input, it requires pointers to directories containing specification FSA in GraphML [GML] format and compiled bytecode of all Java classes possibly visited during analysis (this includes classes which might be referenced by interprocedurally analysed method calls, field accesses and the like). This data will be used to analyse either all or just selected methods from the given classes. The results of the analysis are returned to the console in plain text format. This interface is mainly designed for externally benchmarking the analysis or checking programs which are not under development.

As the main purpose of jFTA is to support the programmer during development of an application, it also provides interfaces suitable for customized IDE-integration. We intended to develop a sample Eclipse plug-in to demonstrate the integrability of jFTA but could not finish it due to time restrictions (see Appendix A for a preview). Instead of reading the input data from files, specification FSA are read in JUNG-format [OFWB03] and the needed bytecode will be dynamically acquired from the program's classpath. Analysis results, comprising found violations and the fixing suggestions mentioned in Section 5.7.2, are returned after termination and can be further processed by the calling context. Like this, it is possible to run the analysis automatically in the background, display found violations directly in the source code via the Eclipse Marker-facility [mrk] and embed the fixing suggestions into the QuickFix mechanism. The usability of such an integrated approach heavily depends on how much time and memory continuous background analysis consumes. While the current implementation can be relatively fast depending on the parameters (see evaluation in Section 7), we expect this to change when it becomes more complete in the future. So instead of performing analysis after each automated compile, it might be reasonable to only run it after several compiles or a minimum amount of time to not interfere with the normal work-flow of the programmer.

6.2 Optimizing runtime performance

When implementing the analysis algorithm, there are a lot of choices to make which affect both speed and accuracy of the analysis, and often trade one off for the other. In this subsection, we want to explain where we had to make such decisions and which effect we expect to arise from them. Our intention is

to make the analysis as fast as possible, so that it does not become unusable because evaluation takes too much time. So when implementing the algorithm discussed in the previous section, we kept this requirement in mind and, as an example, generally accepted trading off memory for speed.

6.2.1 Problems with analysis results caching

As the level of interprocedurality in jFTA is limited, it does not always begin in the `main` method and crawl through the whole program. Instead, the user must specify either single methods, or entire classes whose methods will then all be checked. Analysis of each method is run independently such that no information from the previous run is passed over to the next one. Initially, as the analysis was expected to work purely intraprocedurally, we planned to implement a simple caching facility to reuse results from previous runs. The bytecode of the requested method would be loaded and compared to the last version seen in the analysis to detect if it has been changed. For this purpose, it would need to store a hash value or similar snapshot of the method and its most recent analysis results. If the information in the cache was found to be still valid, the results would be loaded from the cache and the actual analysis of the method would be skipped. In any other case (cache miss or if the method was changed) the analysis would be performed normally.

Caching is only implemented as a stub at the moment, as with the interprocedural analysis extensions it is insufficient to just check the method itself for changes. If referenced classes or methods changed, the stored results for all methods using them must be invalidated. To find these dependencies, the call graph of the program must be inspected. In addition, the different calling contexts from which the method was called need to be considered for maintaining context-sensitivity. Depending on the context-sensitivity depth, we consider it probable that maintaining such a caching structure can quickly become very complex, possibly slowing down the analysis considerably. However, we are currently unable to investigate this further, because jFirm does not support call graph construction yet. In addition, as caching techniques are not the focus of this thesis, they are completely disabled right now.

6.2.2 Fast points-to and aliasing resolution

In the discussion of our algorithm, we used the functions $pointsto(node_c)$ and $aliases(node_c)$ in our design without considering how they are actually calculated. For the implementation, we obviously need to address these problems properly.

Statically finding all possible targets a reference or pointer ***points to*** is a very hard problem in static program analysis. It has been investigated for nearly thirty years [Wei80] and thus even longer than typestate analysis itself. The problem of determining which variables all point to the same (***alias*** an) object is strongly related to this and thus of similar complexity. Handling those problems explicitly would by far exceed the scope of this thesis, which is why we decided to use an external library for resolving points-to information and, based on the information gathered from it, at least approximate the variable aliasing. We chose an interprocedural, context- and flow-sensitive points-to analysis library created by Klaas Boesche [Boe09] because it is written in Java

and also uses jFirm as program representation, allowing for shared use of jFirm graphs and derived data. In addition, it does not simply analyse the whole program and then keep all of the data in memory like most approaches, but it is ***demand-driven***. This means that jFTA can query it for single items, which then gradually builds up points-to information and thus only processes items which are really needed, reducing the amount of time and memory needed for points-to handling. As the library is in early development itself as of writing this thesis, its results most likely are less accurate than those of other, well-established points-to analyses. We willingly accept this limitation in favour of the aforementioned reasons and expect that further development will reduce the drawbacks in the future. The library is currently unnamed, for which reason we invent the name jPTS to refer to it for the remainder of this thesis.

From the results of the points-to analysis, we will derive information about reference aliasing. Whenever a points-to query for a reference ref returns a set of affected allocation sites $\{alloc_1, \ldots, alloc_n\}$, we register ref as a user to all $alloc_i$. So each allocation site builds up its own individual "pointed-to-by" set which can be used to propagate state updates to its users (cf. Section 5.3 and the example in Figure 5.2). If a reference is not used any more after some point, it must be unregistered at all its allocation sites. Otherwise it would still be always updated and never garbage collected, wasting both time and memory.

We create similar links from references to their allocation sites, as given by the points-to information by jPTS. As it is not path-sensitive, it will always return the same (maximal) points-to information for a node in a fixed calling context. Due to the path-insensitivity, it might contain allocation sites which our analysis has not discovered yet, like sites in loop bodies which have not been analysed so far. So we cannot link the reference to this unknown allocation site, but we must wait until we finally find it and process the current node again. As this can happen many times when we traverse the method graph, it is reasonable to cache the complete points-to set and avoid repeatedly querying jPTS for the same node. Therefore, we store the entire points-to set in the affected reference and reuse it to find the missing allocation sites on future requests. Figure 6.1 gives an example for such late allocation site discovery.

Caching of points-to and aliasing information obviously increases the memory footprint of the internal representations of allocation sites and references. Additionally, it requires careful bookkeeping to avoid breaking the references between them when they are copied or merged together as necessary for performing the $\sqcup$ operation on input-sets. We willingly accept this additional work and memory usage because we expect to save a lot of time by fewer points-to queries. This is especially important for the aliasing information which if not cached would be newly constructed each time it is required, which involves expensively intersecting a lot of points-to sets.

6.2.3 Improved jFirm graph traversal with two-level worklist

As we have explained in Section 5.2, we traverse the jFirm graphs using a worklist containing the blocks remaining to be analysed. While the order in which multiple control flow successors of the same block are put into the worklist does not affect the correctness of the analysis, it can immensely affect its runtime (cf. example in Figure 5.1). Analysing the jFirm graph to find loops before the actual analysis could give an optimal ordering, but is costly itself. We

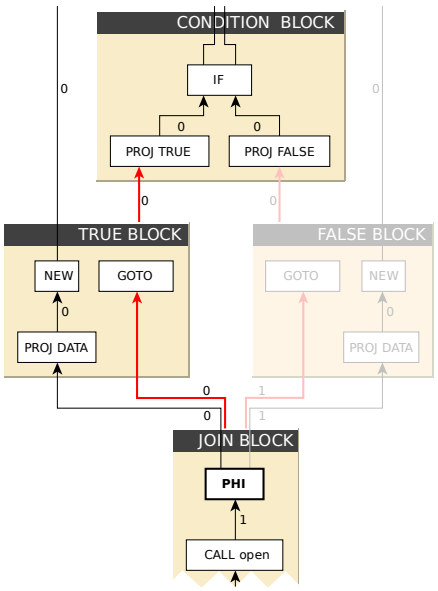


Figure 6.1: An example for late allocation site discovery. The points-to set for the `PHI` node contains two allocation sites, of which only one was discovered by the analysis so far. The other one (greyed block) will be ignored for the first evaluation of the join block, but the points-to information will contain the dependency for future iterations. For clarity, parts of the jFirm graphs have been removed.

consider evaluating different loop detection algorithms on big graphs worthwhile for future work, but do not address this issue for this thesis. Instead, we use two stacks w_1, w_2 as worklists instead of a single one and perform a very simple but surprisingly efficient successor prioritisation by separating blocks into two classes S and M: Those which only have one **s**ingle and those with **m**ultiple predecessor(s). The idea is that blocks in class M are joining points after control flow was split, which means that more than one program path will arrive at every such block b_M. We expect to reduce the amount of unneeded evaluations of b_M by allowing all other analysis paths to proceed as far as possible before actually analysing b_M. Those paths might either reach the end of the graph or stop at different M-blocks for the same reason, until all are blocked.

To achieve this, we add S-blocks to the stack w_1 and M-blocks to w_2. We traverse the graph by popping blocks from w_1 as long as it is not empty, in which case we move the first element from w_2 to w_1 and continue exploring as described above. The reason for taking stacks instead of lists is that their LIFO[3] behaviour leads to depth-first exploration of the graph. If the analysis reaches a loop body, it will suspend at its "head" block hb_M, which will always have at least two predecessors, and add it to w_2 if it is not already contained. The other paths continue and either reach the end or pause at some other block b'_M as well. In the latter case, all b'_M are stacked over hb_M and thus processed before it. Because we only keep one (the earliest) copy of hb_M on the stack, we can be sure that the analysis has explored every path to it from the beginning when it is eventually popped. Thus, its input data set contains maximal information available from those paths. This is not the case for all later encounters of hb_M, which explains why we do not store them on the stack. Finally, we pop hb_M and continue normal exploration.

Compared to randomly selecting the next block, this queueing algorithm can noticeably reduce the number of superfluous block explorations, but it can also

[3] "last in, first out"

perform worse in other situations. It works very well when the graph contains a lot of blocks where many paths join together (e.g. after conditionals, **switch**-statements), but it does not have any information on which successor of a loop head block to chose, for example. So it will often decide to first continue after the loop body, and then evaluate the loop body afterwards. The optimal solution would of course be the opposite case, and errors like this can quickly accumulate inside nested loops and similar structures. We will analyse the efficiency of this solution in Section 7.2.1.

6.3 Implementation limitations in jFTA

In its current state, jFTA can already detect typestate violations in programs which only use a limited set of Java language features:

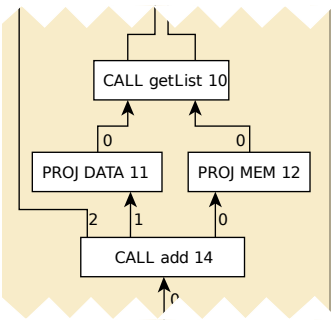
- Basic control flow patterns like conditionals, (nested) loops, **switch**, **break**, **continue**
- Accesses to static and member fields
- Calls to static and member methods
- Arrays (theoretically, see evaluation results and problems in Section 7.2.2)

As exceptions currently do not work correctly in jFirm, they are not supported by the analysis. This is considered future work and will be addressed as soon as jFirm can build correct graphs for methods throwing exceptions. Currently, graphs end abruptly at `THROW` nodes and thus prevent jFTA from analysing the corresponding methods.

6.3.1 Lack of dynamic type information

The analysis results suffer from the fact that jFTA currently only works on static type information. The analysis can therefore miss typestate transitions if the target program uses polymorphism, because only the methods belonging to the static type will be examined. To be more specific, jFTA currently does not handle or process type information at all, except for loading the appropriate classes when analysing method invocations. This was partly due to the lack of support for types in jFirm when we implemented the tool, and it also would exceed the time frame of this thesis.

This limitation obviously makes the analysis performed by jFTA arbitrarily unsound. A possible everyday example is given in Figure 6.2, showing that this is likely to happen in simple, common cases. We are aware that correctly implementing dynamic type information can lead to an explosion of runtime and memory usage. Instead of analysing the method *meth* in the class *c* designated by the static type of a `CALL`, it is then necessary to analyse the same method in all classes extending the type *c*, which can be an arbitrarily large number. Performance results from this version of jFTA are therefore likely to be a lot better than they will be in the future. On the other hand, if the simple version already takes more time than would be feasible for constant background evaluation, we can make assumptions on whether jFTA can ever suit the setting we intend it to be used in or not. Implementing correct resolution of dynamic type information is therefore considered one of the most important tasks for future work.




```
List lst = getList();
lst.add(obj);
```

Figure 6.2: Example situation in which dynamic type information would be needed to correctly analyse the call to `List.add(Object)`. The return type of `getList()` is the interface `java.util.List`. Returned objects therefore will always be instances of a subtype, whose implementations of the `add(Object)` method are different from each other.

6.3.2 Unsound treatment of $\top$ points-to sets

Another problem related to not correctly maintaining type information is correctly handling $\top$ values for points-to information. If a points-to query for a node $node_c$ returns $\top$ because the real targets could not be exactly determined, it means that every object with the same or a subtype could be a points-to target of $node_c$. So if this query was submitted to find the targets of an update operation, all allocation sites, references and fields with such type should be updated. Failure to do so might miss targets which would be affected in real execution. If we wanted the analysis to be sound at any cost and without having the needed type information, we would need to carry the operation out on every object currently in our scope. This would preserve soundness, but of course introduce enormous imprecision. Note that this also applies to the over-approximation we perform when we do not switch to a called method, but instead set the state of all possibly affected nodes $node_c$ to $\top_{type(node_c)}$.

When we implemented jFTA, jPTS returned a lot of $\top$ values because its results are sound but imprecise as interprocedurality and types were not supported yet. Considering this, we had to decide whether to make the analysis sound but probably unusable by handling $\top$ correctly or not. In common terms, this means allowing false negatives in the results in favour of considerably reducing the amount of false positives or not. For this thesis, we decided to accept the former in order to get an idea of what the results can be like when jFirm and jPTS have improved. Note that returning $\top$ does not necessarily mean that we do not get any targets at all, but instead only tells us that there might be more targets affected than the ones returned.

6.3.3 Completely unsupported language features

Apart from these important basic functionalities, jFTA does not handle advanced features like synchronization or calls to native methods. In the near future, these will remain unsupported and their effect on the results is unknown.

7 Evaluation

For the evaluation of jFTA we want to assess its required runtime and the achieved accuracy. As jFirm and jPTS – the core components on which we rely for our analysis – as well as jFTA itself are still in early development stages, these results are only first pointers to where this project might evolve.

For the runtime-assessment, we ran jFTA on a set of larger programs and measure the time effect of different analysis parameters. We also executed the test-suite with our two-level worklist algorithm disabled to analyse its effect on the runtime. All timings were measured on a 2.27GHz Intel Core2Duo processor with 3GB of RAM using 32bit Ubuntu Linux 8.10.

The accuracy was measured on a set of various small methods designed to isolate different Java language features, because some constructs are easier to handle accurately than others. This could not be clearly traced when using real-world examples as the effects are cumulative and not easily separated.

Additionally, to get an initial idea of how the early analysis copes with automatically extracted specification and if we already find any bugs, we analysed the usage of the type `IMAPProtocol` in selected classes of the Columba [col] project.

7.1 Test subjects

We performed the runtime evaluation using the command line interface of jFTA on a test set consisting of five classes as shown in Table 7.1. The first two were taken from the Columba application and make intense use of the `IMAPProtocol` class. `IMAPServer` handles all the IMAP communication in Columba and keeps one single instance of the protocol in a member field all the time. `IMAPProtocolTest` is a JUnit test suite which creates a new protocol object in every method. The next two are from jFTA itself and have been selected because they use methods of many other classes, which increases the impact of the selected maximum call stack depth. Generally, we considered these classes to be good representatives for classes with many lines of code which also have some amount of standard library use (subclasses of `java.util.Collection`; the JUnit framework). The last one was taken from jFTA as well and can be seen as being the equivalent to the `main` method of the program. Although it has only few lines of code, it calls all the methods which basically make up the analysis. For these reasons, we performed one single run with a high maximum call stack depth to get an impression of the runtime when the settings are similar to performing classical interprocedural analysis.

We performed multiple runs of jFTA on each of these classes to evaluate all 48 combinations of the following analysis parameters:

- $depth \in \{0..5\}$ – The maximum call stack depth
- $topPolicy \in \{allStates, firstOk\}$ – How to handle $\top$ states (Section 5.7.1)
- $unknownPolicy \in \{error, ignore\}$ – Treat unspecified method invocations as errors or ignore them (Section 5.7)
- $oneWorklist \in \{true, false\}$ – Use one or two worklists

Class	LoC	methods
`org.columba.mail.imap.IMAPServer`	1083	69
`org.columba.ristretto.imap.IMAPProtocolTest`	553	30
`de.unisb.prog.typestate.ir.FirmWalker`	309	11
`de.unisb.prog.typestate.ir.MethodMemory`	718	37
`de.unisb.prog.typestate.AnalysisRunner`	64	2

Table 7.1: Lines of code (LoC) and amount of methods in the classes used for runtime evaluation

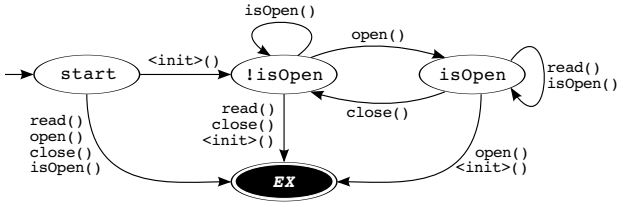


Figure 7.1: A simple specification FSA for an imaginary implementation of a `File` type. For simplicity, it can be only opened, closed or read from. It does not feature a `write(Object)` method or any exceptions (e.g. to symbolise reading while at EOF).

The synthetic test methods generated for the evaluation of the result quality contain several valid and invalid use cases of a particular language feature to measure both false positives and negatives. We used hand-made and very simplistic specification for an imaginary `File` implementation (see Figure 7.1) in this phase to avoid errors arising from ambiguities and misinterpretations in the automatically generated ones. Based on the results from this phase, we want to discuss if jFTA is actually able to detect some sorts of typestate violations in its current state. We divided the examined language features into two categories:

Object referencing and storing The different ways of storing and referencing objects through local variables, static and member fields, arrays or method parameters.

Control flow manipulation Manipulating the normal sequential control flow by using conditionals, **`while`**-loops (with **`break`** or **`continue`** statements) or crossing method boundaries by method calls.

The test set we have created consisted of eight method pairs, each one featuring a valid and an invalid sequence of operations on `File` objects. Five pairs only accounted for intraprocedural scenarios and three ones were interprocedural. We tried to capture many common cases (e.g. nesting loops containing **`break`** statements, moving objects back and forth between fields and variables) but these tests are not exhaustive. Combination of different language features easily makes tracking of points-to information harder and causes more strong updates to be only performed weakly, resulting in an increase of falsely reported may-violations. We investigated the amount of correctly reported must-/may-violations, false positives and false negatives.

7.2 Results

7.2.1 Runtime

While performing runs for all possible parameter configurations, we expected that the call stack depth and using only one worklist would have the biggest impact on the actual runtime. While this was proven to be the case, we also discovered that the other options only influenced the time very marginally. This was obvious for the classes from jFTA, because there was no specification available for them and thus the parameters did not have any effect. It was however surprising to see that this – with one single exception – also applied to the two Columba classes, for which a quite extensive specification was given. The difference in runtime mostly was below two per cent. For this reason, Table 7.2 only shows times for different maximum stack depths and compares between the two- and one-worklist algorithms. Please note that these are only analysis times which do not contain the time needed for loading jFTA and initialising jFirm. In all our testcases, this time was between 0.5 and 0.7 seconds and was therefore considered insignificant. However, this will probably change when jFirm and jPTS are further developed and provide more functionality.

The numbers show that our simple prioritisation algorithm with two worklists performed much better than pseudo-random successor choosing at higher depths. The speed-up factor was in the range between 5 and 10, and in one case it even was 45 ($depth = 5$, `FirmWalker`). For nearly intraprocedural analysis ($depth < 2$) there was hardly any measurable difference in the runtime, and there were even cases where the single worklist was slightly faster.

We did not measure the memory usage in detail, but all sample checks were in ranges from 100 to 500MB for runs with *depth* between 0 and 5. It did not grow above 820MB for the $depth = 10$-run on `AnalysisRunner`, which we consider tolerable though not perfect.

time (s)	*depth*						
one worklist	0	1	2	3	4	5	10
`IMAPServer`	1.8	3.9	9.6	31.9	123.3	539.8	n/a
`IMAPProtocolTest`	2.0	4.2	7.8	18.0	41.2	116.4	n/a
`FirmWalker`	1.6	4.1	23.9	128.2	778.1	8566.4	n/a
`MethodMemory`	2.2	4.3	7.7	30.0	82.2	125.1	n/a
`AnalysisRunner`	1.0	2.5	5.2	52.1	367.9	1701.2	n/a

time (s)	*depth*						
two worklists	0	1	2	3	4	5	10
`IMAPServer`	2.4	3.6	7.6	17.7	38.5	96.9	n/a
`IMAPProtocolTest`	0.9	2.0	3.7	8.3	15.6	29.1	n/a
`FirmWalker`	1.4	2.4	6.2	16.2	59.5	189.8	n/a
`MethodMemory`	1.7	2.9	4.3	9.6	20.7	31.3	n/a
`AnalysisRunner`	1.1	1.9	3.3	11.2	36.9	115.9	18721.6

Table 7.2: Absolute runtimes measured for analysis runs with different parameters for *depth* and $oneWorklistOnly$. Others were omitted because they had no significant effect. For time reasons, there was only one run with $depth = 10$ which also only used the two-worklist algorithm.

	Existing		Reported		False positives		False negatives	
Tested feature	must	may	must	may	must	may	must	may
intraprocedural scenarios								
local variables	2	–	2	–	–	–	–	–
member fields	4	–	4	–	–	–	–	–
static fields	4	–	4	–	–	–	–	–
control flow	1	5	1	8	–	3	–	–
arrays	4	–	–	–	–	–	4	–
interprocedural scenarios								
member fields	2	–	2	1	–	1	–	–
static fields	2	–	2	1	–	1	–	–
parameters	3	–	3	–	–	–	–	–
total	22	5	18	10	–	5	4	–

Table 7.3: The test cases, the number of must- and may-violations contained in each one and the results of running jFTA on it. Except for the array test case, which revealed severe problems with array handling both in jFTA and jPTS, no violations were missed. Falsely reported may-violations were results of weak- instead of strong updates (2), unhandled runtime checks in the program (2) and path-insensitivity (1).

7.2.2 Analysis quality

We ran jFTA on the eight method pairs using $depth = 1$ because the methods were not nested any deeper. The parameters *unknownPolicy* and *topPolicy* did not have any effect in this setting: The test cases only used methods from the dummy `File` specification and did not have unknown parameters which would have been set to $\top$ in the beginning. We will discuss the parameter's effect briefly in the last step of the evaluation, when we perform different runs on real-world examples. By choosing this simplistic test set together with the simple dummy specification, we can analyse how well jFTA can do under nearly ideal conditions. Therefore, the results should not be taken for granted in real projects but merely show the highest upper bound of how good jFTA can perform.

The results from analysing the test cases are shown in Table 7.3. Except for methods using arrays, jFTA did not miss any violations. The problem with arrays were due to the early development stage of jPTS, because it seemed to be unable to track objects through arrays. As a result, a request for points-to information for an `ALOAD` operation always returned $\top$ without any further details. In combination with our incorrect handling of $\top$ results (as discussed in Section 6.3.2), `ALOAD` instructions created references which were not connected to any allocation sites, and therefore did not carry any state information either. Therefore, jFTA was unable to do anything reasonable with arrays at that moment.

The test set generated five false positive may-violations whose causes we want to investigate. Two of them occurred due to weak updates on the states of two objects, which were both accessible via a local variable and a (member or static) field. A simplified example showing this behaviour is given in Figure 7.2. The state of the `File` is updated through the field reference from a helper method, for which reason jPTS returns $\top$ for the points-to targets of the field.

```
File f = new File();  // create new file and also store it in
this.file = f;        // member field. state(f)={!isOpen}

helper_method() {     // in here: pointsto(this.file)=TOP
  this.file.open();   // bad weak update on f after which
}                     // state(f)={!isOpen,isOpen}
                      // instead of state(f)={isOpen}

f.read();             // may-violation: read() in !isOpen
```

Figure 7.2: A situation in which jFTA performs a weak update due to imprecise points-to information, leading to a false positive may-violation. Note that for clarity, the body of the called method has been inlined into the code, so that it is actually not legal Java syntax.

```
File f = new File();
if (??) {
  f.open();
}                     // now state(f) = {!isOpen, isOpen}
/* FAILS */
if (f.isOpen()) {     // dynamic check ignored by analysis
  f.read();           // may-violation: no read() in !isOpen
}
/* WORKS */
if (f.isOpen()) {     // dynamic check ignored by analysis
  Typestate.set       // encodes effect of dynamic check
      (f,"isOpen");   // state(f) = {isOpen}
  f.read();           // OK: read() enabled in isOpen
}
```

Figure 7.3: Runtime state checks guard a possibly disabled method invocation. Unless their effect is made clear to jFTA by typestate directives, it will miss this information and likely generate superfluous may-violations.

We consider this being incorrect as from the calling context, `this.field` cannot possibly point to a different allocation site. So although a strong update would be possible in this observed case, a weak update is performed which leaves the old state in place and causes a may-violation.

Two other false positives were not surprising, as they could have only been avoided if the analysis would have parsed branch conditions, and furthermore have extracted the meaning of the return values of methods like `isOpen()` for the object's typestate. The actual situation is displayed in Figure 7.3: A `File.read()` call is enclosed by a check for `File.isOpen()`. While in real execution, the file will only be read if it is really opened, jFTA will ignore the branch condition and also analyse the read if `f` could only be closed in this situation. While this feature is considered important future work, we currently rely on the programmer to use the typestate directive-mechanism to encode this

```
[FAIL]Testclass@509, call stack [params_interproc_err]:
  Error transition on variable defined in line 507:
    [!isOpen] --close()-> [ex]
  Methods safely enabling the wanted method: [open()]
  Methods not changing state: [isOpen()]

[WARN]Testclass@396, call stack [controlflow_File_err]:
  Error transition on PHI  191:
    [!isOpen, isOpen] --read()-> [ex, isOpen]
  Methods not changing state: [isOpen()]
  Use Typestate.ignore(..) to ignore these states: [!isOpen]
```

Figure 7.4: Example fixing suggestions for a must- and a may-violation.

information into the program code. The last false positive occurred due to the path-insensitive treatment of a `NEW` instruction in a loop which we discussed in Section 5.6.

Apart from these exceptions, the handling of basic Java language features worked well. As mentioned in Section 6.3, exceptions were not supported due to problems with jFirm, so we could not test them. However, by using nested loops and conditionals which are exited from multiple levels by **`continue`** and **`break`** commands, we also tested jFTA on non-trivial control flow examples.

Our fix-suggestion algorithm coped very well with this simple setup. For each must-violation, it found and suggested methods which, if called before the disabled method, would have avoided the violation. This was not possible in the case of may-violations because our `File` specification did not contain a suitable method. Therefore, it suggested ignoring the state performing the error transition as a last resort via the typestate directive mechanism. Examples for both cases are given in Figure 7.4.

7.3 Case study: Usage of `IMAPProtocol` in Columba

Because of their different structure, we discuss the results for `IMAPServer` and `IMAPProtocolTest` separately.

7.3.1 `IMAPProtocolTest`

The results of the analysis runs are given in Table 7.4. The used *topPolicy* did not have any effect as each test case method created its own instance of the protocol object. Therefore, no $\top$ states needed to be resolved. The effect of the *unknownPolicy* was only marginal because the test methods only used methods which were also contained in the specification. There were only three situations in which this was not the case: The specification has a state with two non-deterministically chosen successor states (see Figure 7.5), expressing the fact that the user might be asked to provide authentication credentials or not when logging in. As there were test cases for both situations, the first operation in each of these cases failed for the state representing just the opposite situation.

The other 60 may-violations were reported for method invocations which were actually valid according to the API. As `IMAPProtocol` relies on network

Policy settings		depth = 0		depth = 5	
top	*unknown*	must	may	must	may
allStates	error	0	63	0	63
allStates	ignore	0	60	0	60
firstOk	error	0	63	0	63
firstOk	ignore	0	60	0	60

Table 7.4: Reported violations for `IMAPProtocolTest` for different parameters and *depth* $\in \{0, 5\}$. No must-violations were found, and all may-violations occurred due to ambiguous transitions in the specification. The analysis depth did not have any impact on the results as the test case methods did not call each other.

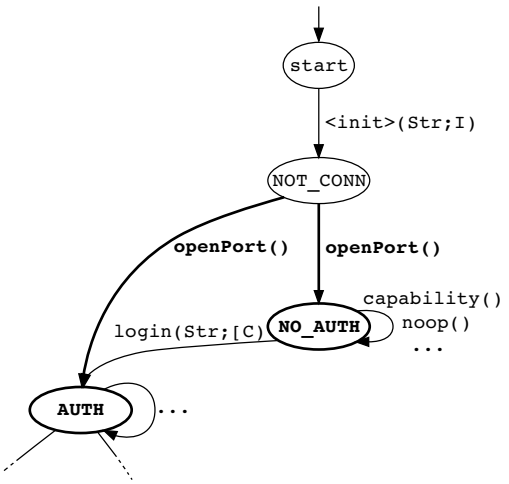


Figure 7.5: Non-deterministic case distinction because the client is not necessarily asked for credentials when connecting to a server. So after establishing the connection, the analysis must assume that the protocol is in `NO_AUTH` and `AUTH` at the same time. Depending on which situation the test case targets, one of the states will perform an error transition. As the specification does not explicitly model all invalid invocations, those are only discovered if the *unknownPolicy* is set to *error*. For completeness, the full state automaton is given in Appendix B.

communication, we found that many methods can unexpectedly throw exceptions (e.g. time-out, connection loss). From the typestate perspective, these exceptions could simply be ignored as they do not indicate a program flaw. Instead, it is important to know in which state the protocol will be when an operation fails, such that the analysis can also determine if the error handling in the application deals with this situation correctly.

This unveils the severity of the missing exception handling in jFTA as well as an interface problem concerning the automatically mined specification. Currently, the mined specification contains a transition to the error state as soon as a method invocation could result in an exception being thrown, including those

Policy settings		depth = 0		depth = 5	
top	*unknown*	must	may	must	may
allStates	error	17	33	243	52
allStates	ignore	0	40	(87) 0	(60) 147
firstOk	error	17	5	243	5
firstOk	ignore	0	3	(87) 0	(12) 99

Table 7.5: Reported violations for `IMAPServer` for different parameters and *depth* $\in \{0, 5\}$. Numbers in parentheses are given where manually removing two unreachable states from the specification changed the analysis results.

of the kind we just discussed. As a result, some methods in the specification also non-deterministically go to the error state. With the current limitations in jFTA and jFirm, there is no way to circumvent this issue except for perhaps completely ignoring certain kinds of exceptions in the specification. Once exceptions are supported, they could be indicated as extra transitions labeled with their name which can then be handled appropriately.

The remaining few may-violations were either caused by other ambiguities in or limitations of the specification. As an example, it is not possible to open another IMAP-folder after the previous one was closed according to the given specification, although the IMAP protocol allows it [IMA].

7.3.2 `IMAPServer`

As mentioned before, this class keeps one instance of the protocol in a member field which is used in all its methods. When starting the analysis of a method, the field's contents are unknown and therefore in state $\top_{\texttt{IMAPProtocol}}$. As a result of this, the chosen *topPolicy* had a noticeable impact on the results because it heavily reduced the amount of reported may-violations, as shown in Table 7.5. The effect of *unknownPolicy* was very strong in this test case, as `IMAPServer` uses many methods which are not contained in the specification. Therefore, using the *error* policy lead to a lot of must-violations. Most violations also occurred for the same reasons as for `IMAPProtocolTest`.

Surprisingly, setting *unknownPolicy* to *ignore* caused all must-violations to vanish in the beginning. By examining the results we found two isolated states in the specification which also had no outgoing edges. In combination with our $\top$-state handling, those two states were always added to state set of the field containing the protocol instance because they could never perform an error transition with the chosen *topPolicy*. Therefore, the state set always contained at least these two "valid" states after any transition (note that this did not affect the testing of `IMAPProtocolTest` because no $\top$ handling was necessary there). When we manually removed the two states, a lot of must-violations were found in the runs with higher depth (values in parentheses in Table 7.5). They were false positives as well and occurred because every method – to ensure that the connection is working – first calls another method which connects the protocol if it not already done so. Obviously, this is done by a lot of dynamic checks at runtime which the analysis does not detect. Therefore, from the analysis' perspective the connected protocol always connected for a second time, resulting in a must-violation.

7.4 Discussion

From the runtime measurements, we can see that the runtime increases drastically with the analysis depth. While the classes under test were not tiny, they only represented a fraction of the size most projects reach during their development time. Bearing in mind our idea of constant program checking during development, the maximum depth should probably not exceed 1 or 2. Even if the analysis is not run after each compilation, depths of 4 or higher will most likely be too slow. Though the current implementation still has some potential for improvement, this will probably not outweigh the runtime effect of correctly handling dynamic type information, as discussed in Section 6.3.1. For the desired field of application, the analysis will therefore most likely need to stay nearly intraprocedural.

The memory usage, though still bearable, seems to be already pretty high. However, jFTA still has a lot of potential to improve on this aspect. In its current state, it saves a lot of unusable data, because it explicitly keeps information about types for which there is no specification available. Without specification, the analysis could never detect any problems for these types and might as well ignore them. When there is only information for a small subset of the used types, this might have a strong effect both on memory usage and runtime. Optimizations like this are considered important future work.

While the analysis results in the optimized, small setting with handmade and simple specification seem to be very promising, the case study on parts of Columba showed that jFTA's limitations are severe and make the results practically unusable in complex situations. On the other hand, the `IMAPProtocol`-class may just be too complex at this early stage (988 LoC, 74 methods). Candidates for the future might be classes implementing the `java.util.Collections` interface (e.g. lists and queues) for they are likely to be less complex as they do not rely on I/O. In addition, the code base on which jFTA could be tested would be large because of the widespread usage of these classes.

Due to time reasons, we did not have any specification for other types and therefore could not inspect more real-world examples. The expressiveness of our evaluation therefore is very limited. Nevertheless, some problems in both the implementation as well as the general concept became apparent: While the implementation suffers from the inability to handle exceptions and dynamic type information, the concept of using FSA which are only labeled with (unnamed) states and method signatures is insufficient as soon as transitions become non-deterministic. In addition, the errors in the automatically mined specification made it clear that even slight incompleteness and/or non-determinism can flood the results with false positives. We can only say little about the general suitability of mined specification from this single example. The fact that some parts of it were malformed (isolated states) probably only exhibits bugs in ADABU and does not unveil a conceptional problem of the mining process. As `IMAPProtocol` holds references to input-/output streams and a socket, its specification was affected by the combination of strong abstraction and non-recursive state definition, as discussed in Section 4.3.1. We suppose that a more detailed representation of the states would have reduced the non-determinism contained in the specification, and thus probably reduced the number of false positives.

8 Conclusion

We presented jFTA, a new typestate analysis for Java which uses the intermediate language jFirm as representation language for the analysed programs and compares their behaviour against automatically mined specification. As the main components on which we built jFTA are still in the earlier stages of development, we could only achieve usable results on very small, restricted and artificial test programs. For real-world test subjects, speed and especially the precision of the analysis are unsatisfactory in the current state. We expect future work to greatly improve on the precision, surely at the cost of speed and memory. This cannot be avoided due to the immense complexity of the (unsolvable) problem for which we aim to find an approximative solution.

8.1 Performance

Our primary goal was to build a fast typestate analyser suitable for constant background execution during development. Therefore, unlike most other typestate analysis implementations, we accepted unsoundness and unsafe assumptions in exchange of achieving higher speed (e.g. ignoring violations occurring when $\top$ typestates are encountered). Also, we provide a set of parameters with which the precision of the analysis can be configured, the most important one being the maximal call stack depth allowed during analysis. Depending on the setting, the analysis is performed intra- or interprocedurally. In the current state, the increase of runtime due to higher depth is nearly unpredictable, as it not only depends on the size of the call graph for the classes under inspection, but is also influenced by the effectiveness of our node-scheduling algorithm (cf. Section 6.2.3). Method-based results caching, as discussed in Section 6.2.1, could perhaps decrease the impact of higher analysis depth, but efficiently building such a cache is probably a very complex task itself.

So for our targeted application area, jFTA should probably be only used with a very low level of interprocedurality and the least restrictive parameter combination ($depth \leq 1, topPolicy = firstOk, unknownPolicy = ignore$). Our evaluation showed that here the runtime is acceptable even for larger classes, and it – on very rough average – increases linearly to the LoC of a class. On the other hand, the analysis is basically intraprocedural then and thus concentrates only on violations with high locality. We could not perform any case studies to investigate whether errors based on typestate violations are usually local to a method or spread across the whole program. Therefore, we are unsure if this limitation is significant or not.

From our efforts so far, we draw the conclusion that we will likely not be able to constantly perform nearly interprocedural analysis in the background. However, (nearly) intraprocedural analysis seems to be possible already, and future work will hopefully increase the level to which we can at least approximate interprocedural analysis.

8.2 Design decisions

The choice of jFirm as an intermediate language turned out to be very suitable for our problem. Explicit dependency graphs provide a lot of simplifications and invariants, which enabled us to clearly structure our analysis algorithm

and keep the amount of special cases low. The used version is very fast and does not add much overhead to our application. On the downside, the fact that it was in the very early stages of development when we started our work has proven to sometimes be obstructive. jFirm however noticeably matured over time and we gave constant feedback about bugs and missing functionality. Unfortunately, some interesting features like exceptions were not implemented in time, but on the other hand, we are not sure if this could have then been correctly implemented on our side with the very limited time given for this work.

Using jPTS to resolve points-to information also worked very well, and we could almost seamlessly integrate it into our program. As development of jPTS started at the same time as jFTA, its results were weak in the beginning but improved a lot over time as well. The situation also proved to be a major advantage for both tools, as communication interfaces for both programs were designed in concert with the authors of jPTS. Using jFirm as a shared basis made any kind of translation unnecessary and allowed for very precise description of points-to information. In addition, it enabled us to use the same method graphs for both programs, basically halving the initialisation time and memory requirements of the program representation for each part. This will be especially useful when dynamic type information will be handled, as the whole type hierarchy is then only built once and can be shared as well. Lastly, jPTS is demand-driven and will for example support multi-threading in the future, which further supports our speed-requirements.

8.3 Suitability of automatically mined specification

We did not investigate the preparatory steps needed to build specification, but only concentrated on the expected results of this process. The one example we worked with showed that the mining can deliver mostly correct models but it suffers from over-approximation. The results are not universally valid as we only had one example for the investigation, but they suggest that some improvement on the model detail could be necessary to reduce the amount of non-deterministic transitions and the problems they cause. Apart from this, the experiments suggest that for less complex types, the level of detail of the generated models is probably sufficient.

8.4 Future work

Ongoing work on jFTA will primarily focus on implementing important missing functionality, though some of it will have to wait for support from jFirm and jPTS.

Exception handling should be addressed next because it allows for more detailed type specification as mentioned in Section 7.3.1.

Dynamic state check detection will help to reduce the amount of impossible program execution paths that are needlessly taken by jFTA. We expect this to prevent a lot of false positives showing up in the results, and probably also provide a noticeable speed gain due to reduced analysis space which is to be explored.

Dynamic type information As discussed in Section 6.3.1, we currently ignore the type hierarchy and features like inheritance or overwritten methods completely. The effects of this can be bad, as the analysis could for example try to analyse the code of an interface instead of the class implementing it.

Result caching techniques Motivated by the results of the evaluation, the idea of caching analysis results completely or in parts should be reconsidered. Although effective caching algorithms might be complex and therefore expensive in terms of time, it still might pay off especially for higher analysis depths.

Increasing model detail In Section 4.3.1 we investigated precision problems stemming from the amount of abstraction performed by ADABU. Different levels of abstraction shall be evaluated in terms of model size vs. information gain and thus improve the usability of the mined models as specification.

Integration The Eclipse IDE plug-in, which we have already started implementing, should be finished to get a first idea of how well the analysis can really integrate into everyday work flow.

In addition, the implementation should be cleaned and optimised to achieve smaller internal data representations, for example by implementing features like copy-on-write.

References

[Boe09] Klaas Boesche. Demand-driven pointer analysis on explicit dependence graphs. Master's thesis, Saarland University, October 2009.

[col] Columba email client, version 1.4. `http://sourceforge.net/projects/columba/` as of 22-11-2009.

[DF04] Robert Deline and Manuel Fahndrich. Typestates for objects. In *In Proc. 18th ECOOP*, pages 465–490. Springer, 2004.

[DLWZ06] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with adabu. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24, New York, NY, USA, 2006. ACM.

[dot] The dot language, graphviz graph visualisation software. `http://www.graphviz.org/doc/info/lang.html` as of 22-11-2009.

[ecl] Eclipse java ide. `http://www.eclipse.org/` as of 22-11-2009.

[FGRY03] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *In Proc. of SAS'03, volume 2694 of LNCS*, pages 439–462. Springer, 2003.

[FYD+08] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. volume 17, pages 1–34, New York, NY, USA, 2008. ACM.

[GML] Graphml, a graph file format. `http://graphml.graphdrawing.org/` as of 22-11-2009.

[GYF06] Emmanuel Geay, Eran Yahav, and Stephen Fink. Continuous code-quality assurance with safe. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 145–149, New York, NY, USA, 2006. ACM Press.

[IMA] Rfc 3501: Internet message access protocol - version 4rev1. `ftp://ftp.rfc-editor.org/in-notes/rfc3501.txt` as of 22-11-2009.

[LBBG05] Götz Lindenmaier, Michael Beck, Boris Boesler, and Rubino Geiß. Firm, an intermediate language for compiler research. Technical Report 2005-8, 3 2005.

[mrk] The eclipse marker interface. `http://www.eclipse.org/articles/Article-MarkMyWords/mark-my-words.html` as of 22-11-2009.

[NNH04] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, December 2004.

[OFWB03] Joshua O'Madadhain, Danyel Fisher, Scott White, and Yan-Biao Boey. The jung (java universal network/graph) framework. Technical report, University of California, 2003.

[ris] Ristretto mail api, version 1.0. `http://sourceforge.net/projects/columba/files/Ristretto/` as of 22-11-2009.

[SPS99] Dale Shires, Lori Pollock, and Sara Sprenkle. Program flow graph construction for static analysis of mpi programs. In *Parallel and Distributed Processing Techniques and Applications*, pages 1847–1853, June 1999.

[SY86] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[Tar73] Robert Tarjan. Testing flow graph reducibility. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 96–107, New York, NY, USA, 1973. ACM.

[Tra01] Martin Trapp. *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.* PhD thesis, University of Karlsruhe, Faculty of Informatik, Oct. 2001.

[Tur36] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[War] Adam Warski. Typestate checker. `http://www.warski.org/typestate.html` as of 23-11-2009.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 83–94, New York, NY, USA, 1980. ACM.

[XRM00] Zhichen Xu, Tom Reps, and Bart Miller. Typestate checking of machine code, 2000.

Appendices

A Preview: Eclipse integration of jFTA

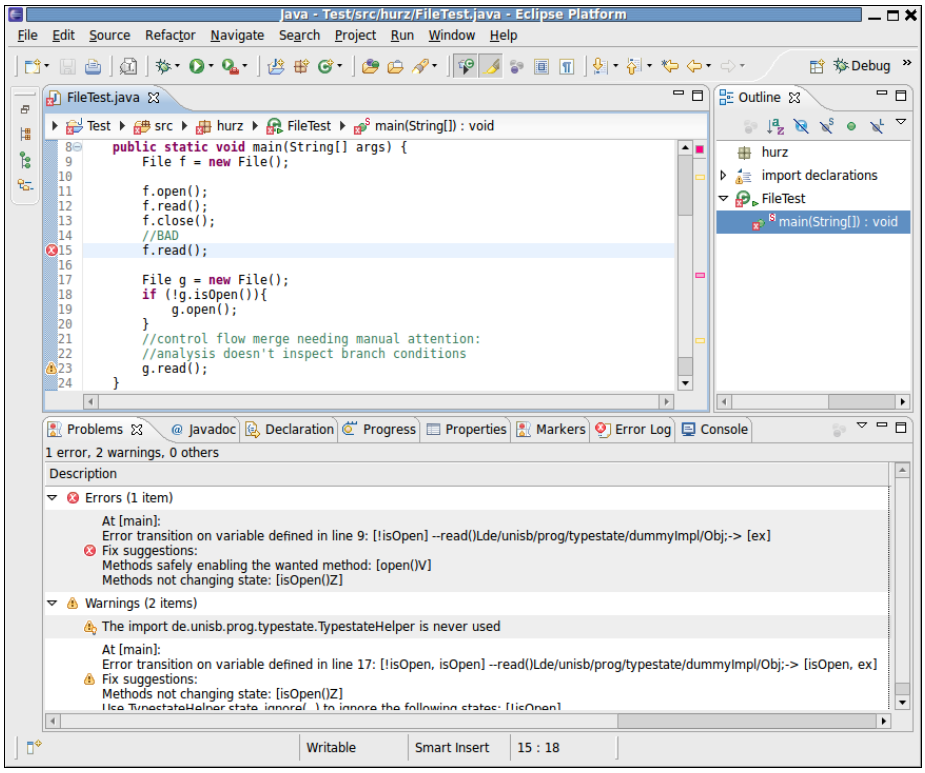


Figure A.1: A screenshot of an early version of the Eclipse plug-in for jFTA. So far the usage of the Marker interface [mrk] is very basic, as it can only annotate the line where the exception occurred. Fixing suggestions do not make use of QuickFix yet. The images shows the appearance of an example must- and may-violation.

B The `IMAPProtocol` specification automaton

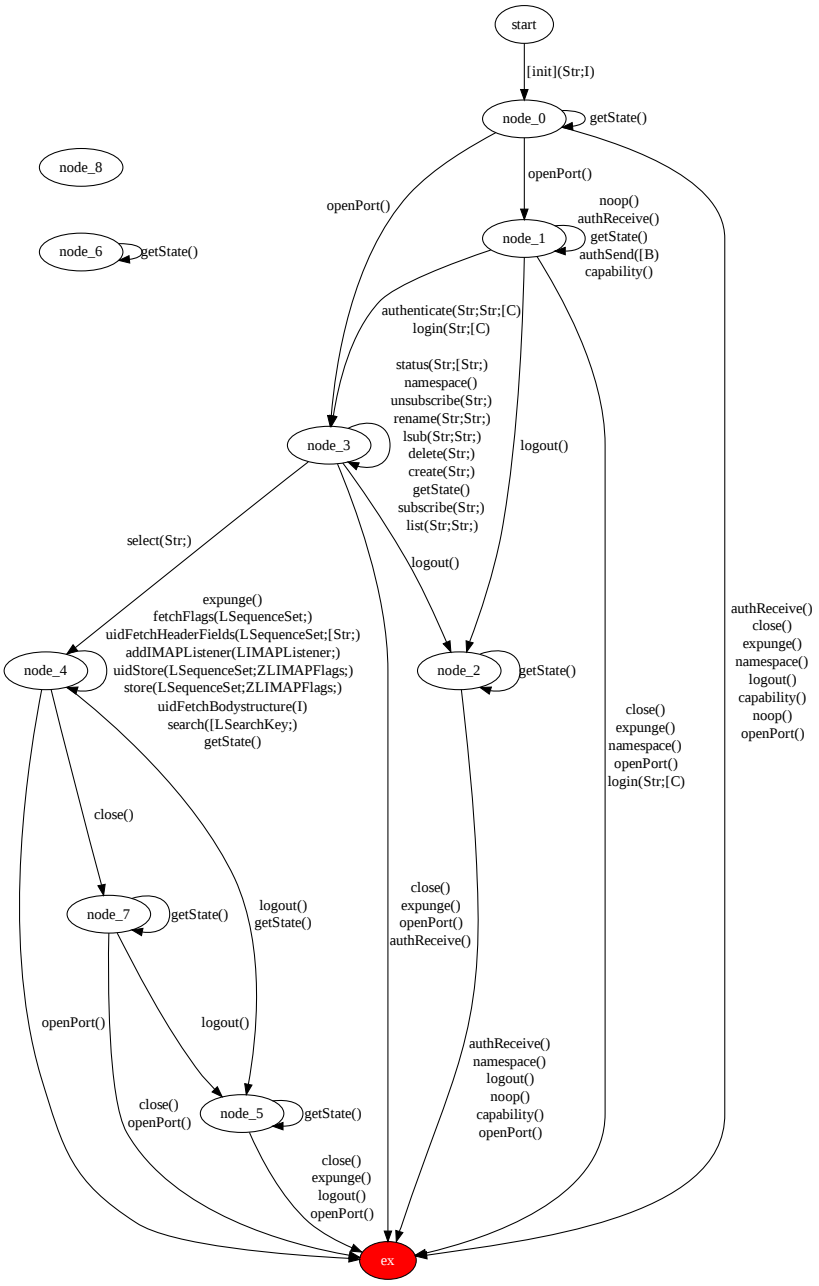


Figure B.1: The specification used for validation of the `org.columba.ristretto.imap.IMAPProtocol` type. This graph was automatically acquired with ADABU and visualised using DOT [dot].