

Jerify

-

An Intra-procedural Type-state Verifier for Java

Daniel Wand - typestate@ewand.de

Abstract

In this paper we will describe a ready-to-use type-state verifier for Java. It is easy to use and fast. Type state extends the standard type system with context sensitive parts that are statically verified in order to avoid the misuse of objects. The type-state specification can be specified with standard Java annotations. It also integrates very well into the sun's javac or any other Java compiler that implements the annotation processor API.

Contents

1	Introduction	1
1.1	Contributions	3
2	Terminology and Essential Concepts	4
2.1	Type-state model	4
2.2	Aliasing	4
2.3	Objects	6
2.4	Control-Flow Analysis	6
2.5	Data-Flow Analysis	6
2.6	Context and Flow Sensitivity	6
2.7	Abstract Syntax Tree	7
2.8	Java Annotations	7
3	Related Work	9
4	Our Approach	11
4.1	Type-state Contracts	11
4.1.1	Type-state Specification	11
4.1.2	Data-Flow Type-State relations	12
4.1.3	Annotations	12
4.1.4	Limitations	12
4.2	Integration into the compile process	13
4.3	Efficiency	13
4.4	Analysis	13
4.4.1	Type-state Analysis	14
4.4.2	Alias Analysis	15
4.4.3	Data-Flow Type-State Relation Analysis	15
4.4.4	General Remark	16
5	Implementation	17
5.1	Overview	17
5.1.1	Initialization	17
5.1.2	Preparing the Analysis	17
5.1.3	Type-state Analysis	17
5.1.4	Tests	18
5.2	Specifying the Type-state Model	18
5.2.1	Annotations	18
5.2.2	Available Annotations	18
5.3	Constructing the extended Control Flow Graph	20
5.3.1	Init	20
5.3.2	Methods	20
5.3.3	Exceptions	21
5.3.4	Conditionals	21
5.3.5	Loops	22
5.3.6	Break, Continue and Return	23
5.3.7	Aliasing Specific Node	24
5.3.8	Method Invocation	25
5.3.9	For Each Loop	25

5.3.10	Data-flow Nodes	25
5.3.11	Expressions	25
5.4	Analyzing the extended Control Flow Graph	26
5.4.1	Control Flow handling	26
5.4.2	Alias handling	27
5.4.3	Type-state handling	27
5.4.4	Data-flow handling	27
6	Evaluation	28
6.1	General Evaluation	28
6.2	Quality Assurance	28
6.2.1	Small Test Programs	28
6.2.2	Verifying itself	28
6.3	Performance	29
7	Outlook	30
7.1	Future Work	30
7.1.1	Type-state Contracts	30
7.1.2	Data-flow Analysis	30
7.1.3	Cluster of Objects	30
7.1.4	Integration in Development Tools	30
7.1.5	Collections	31
7.2	Conclusion	31
	References	32

List of Figures

1	A simple File Type-state example	1
2	<i>@ChangeState</i> Example	4
3	Accurate Alias Example	5
4	Inaccurate Alias Example	5
5	Example of an annotation declaration	7
6	Example of an annotation use	8
7	javac option to use	13
8	Analysis Overview	14
9	<i>@TypestateChecked</i> and <i>@ScopeEndStates</i> Example	18
10	<i>@ChangeState</i> Example	19
11	<i>@RequireState</i> Example	19
12	<i>@SetState</i> Example	19
13	<i>@TS</i> wrapper Example	19
14	<i>@ReturnValue</i> Example	20
15	Method Control Flow	21
16	Exception Handling Control Flow	21
17	Conditional Control Flow	22
18	Switch Control Flow	22
19	Do-While Loop Control Flow	22
20	While Loop Control Flow	23
21	For Loop Control Flow	23
22	Break Control Flow	23
23	Continue Control Flow	23
24	Return Control Flow	24
25	Var Declaration (Alias Handling)	24
26	Assignment (Alias Handling)	24
27	Typestate Handling	25
28	For Each Loop Control Flow	25
29	Runtime	29

1 Introduction

The maintenance cost of software is one of the prevalent parts of software development budgets. Some researchers even refer to the rise of maintenance cost in software development as a crisis [16]. This leads to the question “What makes maintaining software so incredible expensive?”. One factor is that in software projects there are usually many developers each working on different parts of the software. Thus each developer only has a general idea about the whole software program. As the developers who work on a certain piece of code change over the years, information about its subtle properties might slowly decline. This is especially the case with features that are not enforced by the development tools or not even documented. One particular example which modern programming languages, such as Java, lack, is the ability to specify and enforce object usage protocols specifying how an object can be used. Leaving them implicit increases the effort that developers will have when they change the code. With every change there will also be a greater risk of introducing bugs. Preventing the misuse of objects by specifying their behavior and enforcing the correct usage will therefore decrease the effort and the expenses in maintaining a piece of software. In this thesis we present an extension to the Java programming language that can statically detect and prevent this class of bugs.

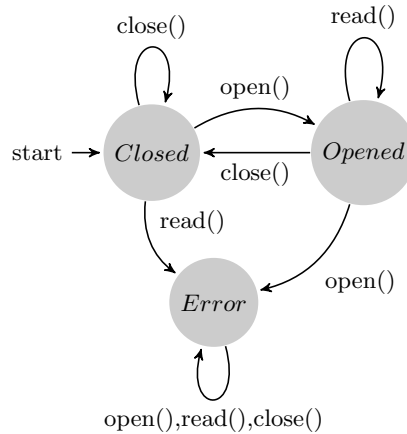


Figure 1: A simple File Type-state example

An example of such a usage protocol would be a File which initially is closed and has to be opened in order to read from it. It has to be opened because reading from a File before it is opened may lead to undefined behavior or an exception being thrown. The usage protocol of such a simple File interface is illustrated in Figure 1. Many programs — respectively objects — have to be used according to such usage protocols. Leaving them implicit causes a greater risk of bugs, if the code has to be extended. Clearly it is desirable to have an extension of the compiler that also verifies the correct usage of those objects.

Extending the normal Java types with context-sensitive states leads to a so called type-state systems which were introduced by Strom and Yemini [17] and can encode such usage protocols. Whereas Java types only prevent the developer from invoking methods that are not implemented in the object, type state also prevents him from invoking methods in an order that is not meaningful, respectively legal for that object. Using this type-state systems to encode these usage protocols, we can enforce them and have the additional benefit of an explicit, always up-to-date

documentation of these protocols.

We feel that it is most intuitive if a type-state change (for example the transition from the Closed to the Open state via the method “open”) corresponds to a method invocation. The type state of an object possibly changes with every change to the object, either through method invocations or direct field accesses. Our analysis does not provide any type-state guarantees for direct field accesses. The analysis then would have to have much more detailed knowledge about the objects, which would complicate the analysis. Furthermore, it also complicates the specification of the type-state protocol itself which we want to keep as simple as possible. Instead, the analysis warns the developer if any type-state checked object’s field is directly assigned and those field accesses should be converted to getters and setters; at least if they change the type state.

One of the main challenges for a reliable type-state verification analysis for Java is the problem of pointer aliasing. Since objects are referenced through variables, several variables can point to the same object. In order to perform the type-state analysis we not only have to know which variables (references) are manipulated but exactly which objects are manipulated and which are not.

There are fast implementations for may-alias analysis, but unfortunately a may analysis is too inaccurate. As its name already implies, a may analysis result does not provide enough information to know if the object is actually referenced. Thus we only know that every method invocation on that variable may change the type state of the objects. Therefore, the type state after every method invocation is the combination of both the old type state (the object might not be referenced) and the new type state (the object might be referenced). Fairly soon this leads to a situation in which almost no operation is legal, only because the analysis does not know in which type state the object actually is in. Therefore an alias analysis that can guarantee that a variable references an object is clearly preferable for a type-state analysis. For a more detailed explanation with an example of this problem see Section 2.2.

Additionally in most related work the context in which a method is invoked does influence the method’s analysis. This is contrary to our goals. We want to provide a type-state verification algorithm which can be used while developing a program. For this purpose, the analysis has to be fast. Furthermore, one has to be able to run it even if parts of the program are absent. Thus we choose to use a modular approach. This means that each method has its own type-state specification and its body is verified separately from the rest of the program. The verification then uses only the specifications of the other methods which are used within the method’s body. Hence we are not using a context-sensitive analysis, because we only use the specification and not the actual analysis result.

One the one hand this enables a fast analysis which can analyze incomplete (sub-)programs as long as the type-state behavior of all methods used in that (sub-)program is specified. On the other hand this approach has some drawbacks. As the analysis is performed on each method separately, the analysis can not track behavior that involves several methods. The main limitation of this is, that the alias analysis has no possibility of knowing if a reference, that is returned from a method invocation, is aliased to any reference, which is already present in the method. During this thesis it is assumed that the references will not be aliased. This is unsound (unless dynamical checks are inserted) but seems to be reasonable as this case is very common.

In contrast to all other related work we know of, our analysis allows the developer to query the type state of an object, within a condition of an if statement (or conditional expression). This enables our analysis to correctly verify the type state in scenarios where an object (e.g. a file) is only changed (opened) under certain conditions (it was closed).

1.1 Contributions

The main contributions of this thesis are:

- A context-insensitive, flow-sensitive type-state verifier that uses a combined type state, alias and data-flow domain and integrates smoothly with the standard Java development tools.
- An easy way of integrating the type-state protocols in the source code with the help of Java annotations.
- An approach that tries to make intra-procedural analysis sufficiently effective for type-state verification.

Our implementation can handle almost every feature of the Java language, except if the type-state behavior is influenced by either concurrency or reflection our analysis is inapplicable.

The rest of the thesis is organized as follows:

Section 2 introduces all used terminology and concepts used in the later sections. Section 3 gives an overview over related work and Section 4 describes in general the approach we took for solving intra-procedural type-state verification. Section 5 then gives an in detail description of our implementation of the analysis algorithm presented. In Section 6 the results obtained will be presented and Section 7 will give an outlook on which improvements can be made.

2 Terminology and Essential Concepts

In this section the terminology and concepts used throughout the thesis will be explained. The following two sections then provide an outline of our algorithm and present related work.

2.1 Type-state model

In Java, a type defines a fixed set of methods which are legal to invoke on any object of that particular type. Type state extends this static type with respect to the order in which those methods can be invoked. Using type state, the legal methods now dependent on the context the object is in, whereas before all implemented methods were legal. For type-state verification, a type-state model is needed. This model has to describe which sequence of method invocations are legal. Additionally the model has to provide an efficient way to check whether a particular method invocation sequence is legal or not. Finite-state automata are sufficiently expressive to describe the most common type-state scenarios. They consist of finitely many states, an input alphabet (which in our case are the methods names) and a state transition function which computes the new state from the current state and the method's name. All method invocation sequences that are recognized by the finite-state automaton are legal, all the others do not fulfill the type-state property and are therefore rejected as illegal. The finite-state automaton can be implemented very efficiently, because the number of states is usually small (small enough so that humans are able to understand the specified type state) and the states do not change during the analysis. It can also be easily annotated to the source code. As is shown in Figure 2 it is only necessary to annotate the from-states and the to-states to each of the method's header.

```
@ChangeState( from="Closed", to="Opened" )
void open();
```

Figure 2: Example of an type-state annotation, specifying the type-state implications of the method *open* of Figure 1

By default, all states of the finite-state automaton are accepting states, which means that it does not matter in which type state an object is at the end of its lifetime. Naturally the type state in which the object has to be at the end of its lifetime can be restricted, such that an object has to be in certain type states before it is garbage-collected.

2.2 Aliasing

A major problem which type-state verification for Java has to solve is that all objects are allocated in the Heap. Because of this, several variables may point to the same object. A change of the type state by a method invocation on the object which a variable points to, will implicitly effect all other variables that point to the same object. Thus the analysis has to keep track of all these changes in order to compute a sound result. Analyzing to which objects a variable may point to is possible with efficient algorithms (for an example see [10]).

Unfortunately a may points-to analysis is not enough. Assume the scenario of Figure 3 in which there are two variables (of a type that obeys the type-state requirements shown in Figure 1) and two objects (objects *o1* and *o2* of a that type). Both objects are initially in the “Closed” state and variables *v1* and *v2* both point to object *o1*. Then the method “open” is invoked on variable *v1*, changing the type state of object *o1* to “Opened”. Now the method “read” is invoked on variable *v2*.

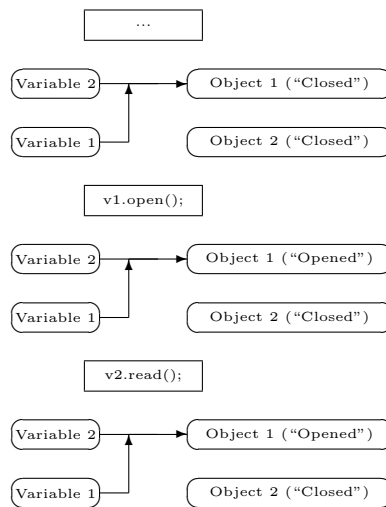


Figure 3: Example of a simple type-state analysis with accurate aliasing information. The Boxes represent the code steps, the type-state analysis is shown in the brackets behind Object 1/2 and the arrows show to which object a variable points

Now assume the scenario of Figure 4 in which the alias analysis erroneously reports that variable `v1` may also point to object `o2`. After invoking the “open” method, object `o1` and `o2` are either in the state “Opened” (exclusive-) or in the state “Closed”. If variable `v1` points to object `o1`, then object `o1` is opened and object `o2` is closed. Otherwise variable `v1` points to object `o2`, then object `o1` is closed and object `o2` is opened. In the next step the analysis rejects the invocation of “read” as possibly leading to an error only because the aliasing information at the beginning was inaccurate.

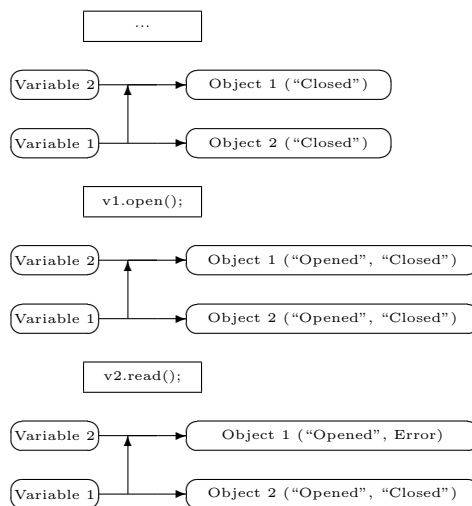


Figure 4: Same example type-state analysis as in Figure 3 but with inaccurate aliasing information, leading to an inaccurate type-state analysis and thus a wrong error report

This example illustrates that reliable alias information is very important for a reliable type-

state analysis. Because we showed that inaccurate alias analysis information quickly leads to an explosion of possible type states. The main cause for this is that the analysis has to assume that the type state of the objects might have changed and that it might not have changed. Leading to an analysis result which is the superset of both possibilities for all following analysis steps.

2.3 Objects

The term “Object” normally refers to instances of classes that are created at runtime. But as statical program verification is not carried out at program runtime, it can not consider the runtime objects. It has to group objects, which would be created at runtime, into so called abstract objects (without actually analyzing any program execution). Therefore, the alias analysis has to analyze the program and statically verify where objects are created and abstractly compute a superset of all possible runtime constellations of objects.

2.4 Control-Flow Analysis

In order to describe the verification algorithm the notion of control flow is needed. Control flow is the order in which elementary blocks (statements) can be executed. Although in general there are elementary blocks for all statements, we only consider those elementary blocks that will have effect on the type-state and / or aliasing behavior of the program that is to be verified. A control-flow analysis [13] algorithm will construct a Control Flow Graph for methods, which describes the entire control flow for one particular method. In this graph the nodes are the relevant elementary blocks and the edges describe how the control might pass from one elementary block to another. Our Control Flow Graphs, will always have one root (the method entry point) and two sinks. One of the sinks corresponds to an exception being thrown which is not caught inside the method and therefore leaves the control flow of the method. The other sink corresponds to all other means of ending the methods execution. An execution path is a list of nodes that are connected by edges. We call an execution path “terminating” if it reaches the end of the graph or subgraph. Normal termination will corresponds to reaching the successor node or the method endpoint, whereas not normal termination will be by an exception being thrown and not caught within the graph or subgraph.

2.5 Data-Flow Analysis

Once the control-flow graph has been constructed it is possible to use it for the type-state verification. In order to perform the type-state verification we treat the type state of an object as a hidden field for which we statically verify that it never reaches an error state. This can be done with an adapted data flow analysis [14] which essentially uses the control-flow graph to compute a fix point analysis of those hidden fields. The exact details are described in the implementation section (starting at page 17) of this thesis.

2.6 Context and Flow Sensitivity

A control-flow analysis is not needed in all cases. An analysis which does not require the information in which order the elementary blocks are executed is called “flow insensitive”. An analysis that depends on that information is called “flow sensitive”. The flow-sensitive case can be further divided in an “inter-procedural” analysis, which analyzes the flow information of the whole program and in an “intra-procedural” analysis which analyzes the flow information on a per-method basis. Because it takes significant effort to compute the flow information, flow

insensitive information is computationally cheaper than the flow sensitive analysis. The intra-procedural analysis on the other hand is significantly cheaper than the inter-procedural analysis. Similarly, an analysis is context sensitive if the analysis of a method depends on the analysis of the part of the program (the context) where the method is invoked from. Thus an analysis is context insensitive if the analysis of a method does not depend on anything else but the method itself. As methods can be called from many different contexts it is obvious that the computation of a context sensitive analysis is significantly more expensive. But the context sensitive analysis might be able to compute more reliable and precise results.

Our analysis is an intra-procedural, context-insensitive but flow-sensitive analysis.

2.7 Abstract Syntax Tree

Although the source code is a simple text document, its interpretation it is not a simple linear stream of tokens but a tree structure. Therefore, the source code can be converted into an equivalent concrete syntax tree, which preserves all details of the code. But this tree has several redundancies, for example the parentheses are already implicitly given by the tree. Aside from this language constructs such as the if-then-else construct can be described by a single node with two (condition and then) or three (condition, then and else) branches, omitting all the keywords and parentheses. By leaving out all redundant and therefore irrelevant information one gets a tree structure that represents all language constructs by a node, and all complex sub constructs (like the if branches) as subtrees. This tree is called the abstract syntax tree. Fortunately, we do not have to construct the abstract syntax tree as this is done by the Java compiler, which generates the abstract syntax tree, then checks the source code for validity and then passes the abstract syntax tree on to our analysis.

2.8 Java Annotations

For type-state verification we need a type-state model that can be checked. One could save this additional information in an external file, but it would be much more convenient to integrate this directly into the Java code of the class. For the purpose of providing a meta-data facility for Java, annotations were introduced in 2004 [2] and made available with the fifth release of Java 2. Annotations do not directly influence the program's semantic but can be used by tools

```
public @interface ExampleAnnotation {  
    String[] someStringArray();  
}
```

Figure 5: Example of an annotation declaration

such as our type-state verification algorithm. They are declared similar to interfaces by using the special *@interface* keyword. Additionally method declarations must not have parameters nor throw clauses; the return types are restricted to primitives, String, Class, enums, annotations and arrays thereof. With the seventh release of Java 2, annotations will be permitted to be written on nearly any use of a type. This is due to including JSR 308 [8] which addresses shortcomings in the previous annotation specifications. Java also provides a standardized meaning of processing annotations, the Annotation Processor API [4]. We use the Checker Framework [15], which is build to enable easy use of the features of annotations and annotation processing capabilities of the seventh release of Java 2, to implement our verification algorithm. Unfortunately we had to

```
@ExampleAnnotation(  
someStringArray = { "String1", "String2", ... } )  
SomeTyp variableDeclaration;
```

Figure 6: Example of an annotation use

implement some parts, e.g. the flow analysis again, because our requirements differed from those of the Checker Framework.

3 Related Work

There are many projects producing tools (and frameworks) that try to find bugs in programs. Many of those expect a specification of what the program should do and try to find parts of the program that violate those specifications. Here we will focus on related projects that use type-state analysis or similar approaches.

Such projects are, in general, distinct by the approach they take for implementation and design of their algorithm. The general properties according to which these projects can be categorized are described in the following:

As we have discussed in Section 2.2 all type-state analysis algorithms also need to have access to reliable aliasing information. This can either be done by first computing the alias information and then running the type-state analysis algorithm or by using a combined approach which generates the aliasing information simultaneously to checking type-state correctness. An example of a tool that uses a phased approach is ESP [5].

In the combined approach the type-state analysis has (in some cases) access to more precise alias information than in the two-phase approach. This is because the aliasing information may change with each analysis iteration. Assume that a variable inside a loop is assigned with a different abstract object in each analysis iteration. In the two-phased approach the aliasing analysis would “inform” the type-state analysis that each of those abstract objects may be referenced from that loop variable. Whereas in the combined approach, the type-state analysis would know (for each iteration) the exact abstract object that is referenced from that loop variable. Clearly, the two-phased approach can lead to more erroneous error reports, because the alias information is less precise as the result of the combination of all iterations.

This is why our analysis and the other tools like the type-state algorithm developed by the IBM T.J. Watson Research Center and the Haifa Research Lab [9] use a combined domain. In contrast to our analysis, their analysis is context-sensitive. Context-sensitive analyzes require only the specification of the type-state automata. Our analysis requires a specification which defines the type-state effect of every method and the type-state side-effects it has on its parameters and its return value. For context-sensitive analyzes the specification of the parameter is not necessary as those are specified by the context the method is called in. The specification of the return value is not necessary as for each context the type state of the return value is computed.

Obviously, as long as there is no type-state misuse, the context analysis of the context is at least as precise as the specification has to be in the context-insensitive case. In most cases the context is by far more precise, thus enabling a far more precise analysis. Although the context-sensitive analysis is more precise, we maintain that it is unsatisfactory for type-state analysis. Our main reason is that the core goal of type-state abstraction is to provide a way to specify and encapsulate the behavior of each object. Thus, each object (and each of its methods) should have a specification of what behavior is legal and which is not. We discuss this in more detail in Section 4.1.

Additionally, context-sensitive analysis requires the complete program to be available. This makes it impossible to analyze partial programs, for which only the specifications for parts of the program is available. The context-sensitive analysis also requires more computational resources and it requires a complete rerun on the whole program — even after small changes. Thus, it is significantly slower.

Another important distinction is whether the tool uses a verification algorithm or is a defect detector. A defect detector (for an example look at a defect detector from Microsoft [18]) only tries to find errors respectively bugs. Verifiers on the other hand prove the absence of bugs (at least of those the verifier promises to find). Our algorithm can be categorized in both categories, for local type-state guarantees the algorithm proves the absence of type-state protocol breaches,

thus it is a verifier. But as we use a modular approach for type-state analysis, some type-state properties that are either context sensitive or involve aliasing behavior in different objects are not handled by our analysis. In fact the fundamental problem is that they can not be expressed in our type-state model at all. See Section 4.1.4 for more details.

Type-state algorithms can further be divided by the kind of type-state model they use and the way those are specified. For example IBM's type-state algorithm (presented above) does not integrate the type-state specification into the code of the program that is to be checked. This has the advantage that the type-state specification is possible even for code which normally can not be changed (for example the standard library). Microsoft Research's Fugue [6] and our analysis use annotations to specify the type-state behavior. On the one hand this has the disadvantage that if the code can not be annotated (or changed at all) it can not be checked. On the other hand the type-state specification is always present when the developer reads the code. This helps keeping the type-state specification up-to-date and is an additional documentation for any developer who is trying to understand what the code does.

Fugue provides a modular static type-state verifier for all languages that compile to the Common Language Runtime (all .NET languages). Although not explicitly stated we assume that their analysis is, like ours, context insensitive and intra procedural. In contrast to our tool theirs does not integrate into the build process but rather works on the compiled byte code. Furthermore, they do not use a finite-state automaton to model the type state, but use plug-ins to compute if a state is valid and which the next state is. This makes their approach more flexible, as plug-ins which are much more expressive than finite-state automaton are possible. By assuming that finite-state automatons are sufficient to express most type-state contracts, we can specialize our analysis and can therefore provide a presumably more efficient analysis. We can also provide a simpler way to specify the type state.

In some cases the developer wants to query an object to find out if it is in a particular type state. Although this is useful in many circumstances (e.g. to open a file if and only if it is closed), none of the tools discussed so far provides this functionality. Our analysis supports this feature which is discussed in Section 4.1.2.

4 Our Approach

4.1 Type-state Contracts

Object oriented programming in modern programming languages such as Java enforce an encapsulation of implementation details within one part of the program, usually within one class. Those classes, more specifically their types, define which operations they can perform, respectively which methods they implement. As described, type-state verification further restricts the order of method invocations. Many of those type-state properties can be described by finite-state automata. Since method invocations change the type state of an object the transitions of the automata are labeled with the methods which cause those type-state changes. It is possible that several transitions from one state are labeled with the same method, but in general this is not desirable as it makes the analysis process harder, because the object may be in one of several states. Whereas the transitions correspond to method invocations, the states of the automata describe a subset of methods whose invocation is legal if the object is in that particular state. If a method is invoked but is not in the subset of methods for the current type state, the automata are assumed to have an implicit transition to an implicit error state.

In order to preserve the encapsulation of implementation detail which the Java type system attempts, it would be harmful to make the type-state requirements of the methods depend on the context the method is invoked in. Instead we choose to annotate each method with a type-state contract. The contract expresses which requirements the method has on type-state of the receiver object and the parameters passed and also which effect it will possibly have on them. The type-state requirement for the receiver should naturally also express the requirements and effects the method has on the field of the class (not only the type state but all other requirements, e.g. initialization). The field requirements possibly make the type-state model for the object more complex than handling the fields implicitly by the analysis algorithm. But this would contradict the usefulness of the type-state models contract descriptions, because it would not express all necessary preconditions and make the type-state correctness dependent on more complex analysis than checking the fulfillment of the type-state contract. An additional benefit is that developers using the object now have an explicit (and enforced) description of how the object is allowed to be used.

Preserving the encapsulation and requiring an explicit specification for each object makes our analysis modular. This even enables our analysis to verify partial programs.

4.1.1 Type-state Specification

To be able to type-state check an object, one first has to specify the desired object usage protocols. Our analysis uses a finite-state automaton as type-state model. To specify this automaton there are several annotations with which the corresponding methods, respectively classes are annotated. The main advantage of using annotations is their tight integration into the code of the program.

4.1.2 Data-Flow Type-State relations

In a program there are frequently cases in which the required behavior is dependent on the type state the object is in. In these cases the developer needs a way to query the object for its type state. Assume a file is passed to a method. The method then should open the file only if it is not opened yet. After that it should write some output to the file.

```
if (!file.isOpen()) {
    file.open(defaultOutputFile);
}
file.write(SomeOutput);
```

Naturally, more complex scenarios, for example the combination of several type-state requirements, are possible.

```
if (file.isOpen() && socket.isConnected()) {
    file.read();
    socket.send();
}
```

Obviously it is desirable for a type-state analysis to be able to handle these scenarios. In order to support this in an analysis there are two requirements. First, the developer has to be able to specify the relation between the return value of a method and the type state of the receiver. Second, the analysis has to be able to track the data flow of the return values and to evaluate the conditions of an if statement and conditional expressions. The current implementation of our analysis only provides this feature for methods that return a boolean value direct inside a condition. In principle this is also possible for non-boolean types and to method invocations outside of conditions (although the relation will only be evaluated in a condition). As most of the required functionality is already in the analysis implementation, this extension is planned to be integrated in the future.

4.1.3 Annotations

The type-state automaton is directly annotated to the methods that change the type state. There are several kinds of type-state requirements. First there is a need for an annotation that simply enforces that the method is only called in certain type states. Second sometimes it is necessary to express that after this method is invoked the type state is always exactly specified. The majority of the type-state specifications will be type-state changes, requiring to be in one state and afterwards being in an other state. The annotations available in our implementation are explained in more detail in Section 5.2.

4.1.4 Limitations

Unfortunately, the use of type-state contracts on a per method basis causes some limitations. We restrict our type-state verification to type-state properties that are contained within one class. Nevertheless there are cases in which objects are referenced from several classes, in those cases the type state is inherently non local. In some cases this might be overcome by a clever design of the type-state automata for those classes, but in general this will not be possible. A solution to this is discussed in Section 7.1.3. The problem is solved by extending the type state to clusters of objects. The cluster then shares a common type-state instead of each single object having its own. Additionally, the restriction to intra-procedural analysis also causes our alias analysis to become inaccurate, as the references that are returned from method invocations come from

“outside” the analyzed scope. Thus we have no information if they are the same to any other reference already inside the method. For the scope of this thesis we assume that the references brought into the method are fresh. This is obviously unsound, but we expect this to be true in the majority of the cases. To circumvent this problem one could introduce dynamical checks wherever potential aliasing would have effect on the type-state correctness.

4.2 Integration into the compile process

A program verifier for Java can work at different stages. For example, it can run on byte-code after the Java compiler created it. Here some of the Java language constructs are broken down to more basic constructs. If it has its own Java parser it can directly run on the source code independent of the compiler. We choose to integrate our type-state verification algorithm as a Java Annotation Processor [4]. This way each time the compiler is run (while our analysis is enabled) not only the standard Java semantics and syntax are checked but also the type state our processor enforces. All type state related error messages are presented to the developers via the same standard error and warning reporting mechanism that the compiler uses, but our analysis will only be run if the code compiles without any Java compiler error.

```
javac
-processor de.unisaarland.cs.st.jerify.verifier.Checker
normal parameters
```

Figure 7: To use the type-state verifier simple add the `-processor` switch to your Java compiler options

Currently we require a Java compiler which supports JSR 308. One compiler is available from the JSR 308 homepage [7] and JSR 308 will be integrated into the next release of Java.

4.3 Efficiency

As our analysis is integrated into the compiler, it is intended to be run with each compilation. In the future the analysis maybe even be run while the developer is programming. Because of this, performance is very important. In Section 2.2 we explained that we have to use a more expensive aliasing analysis contradicting our desire for highly efficient algorithms. Fortunately, the restrictions we imposed by using type-state contracts enable us to analyze each method separately from the rest of the program.

In general method size seems to be limited and it is even suggested that it will not be longer than certain thresholds. Especially if there are complex type-state requirements methods that are longer than those thresholds will certainly be very hard to understand for humans. Due to the rather small size which methods tend to have and the context-insensitive approach of our algorithm, we can employ techniques which in the context-sensitive or inter-procedural case would be infeasible.

4.4 Analysis

The analysis process is split in two phases (see Figure 8 for an illustration). Before the analysis is run, the Java compiler parses and checks the program, if any error is found our analysis is not run at all. After that in the first phase the program is analyzed and all parts that could possibly effect the type state, alias or control flow analysis are integrated into an graph structure. This

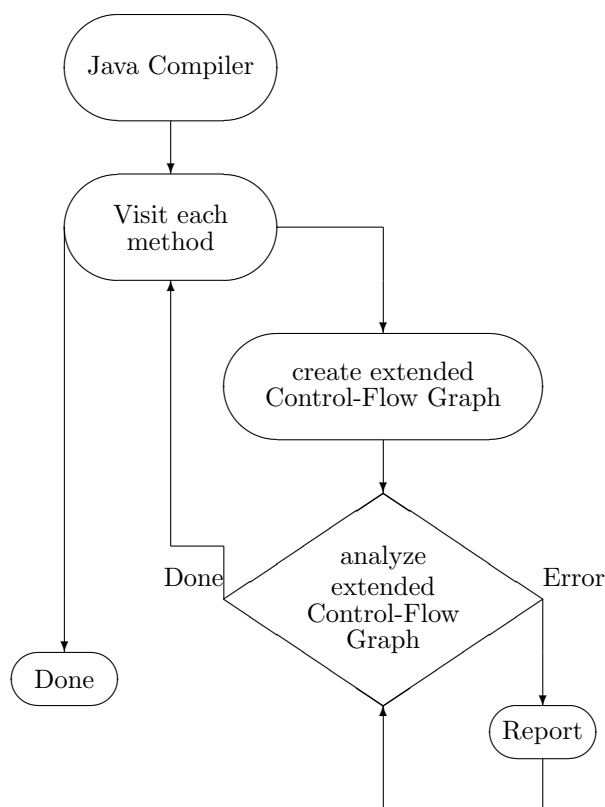


Figure 8: Overview over the type-state analysis algorithm’s stages

essentially represents a control flow graph extended with additional nodes that correspond to type-state and aliasing changes.

The second phase then uses this graph to verify all uses of objects that are type-state checked. The type-state and alias information is simultaneously pushed through the graph. By combination the alias and the type-state domain a more precise result (as explained in [3] and discussed in Section 3) can be achieved and thus false errors are reduced. The main reason for this is that the type-state analysis in the combined domain has access to the alias results for each analysis iteration, which is more precise than the superset of that alias analysis results that is used in a two-phased approach.

4.4.1 Type-state Analysis

The type-state part of the analysis is relatively simple. The type-state analysis only keeps a set of states (of the type-state automaton) for each abstract object. A set of states is required as during the analysis an abstract object might be in one out of several states. As long as all method invocations are legal in all states the abstract object is in, this causes no problem. If one invocation is not legal in one of the states, the analysis has found a type-state misuse.

When running the first part of the analysis, the control flow graph is created. For the type-state analysis, nodes are inserted whenever a method invocation that changes a type state is found. This node contains the information which method is invoked on which variable. The actual type-state verification happens during the second phase of the analysis. Because the assignment

of abstract objects to variables might change during the analysis, the type-state analysis queries the alias analysis each time an type-state node is reached in the extended control flow graph. The alias analysis then provides the abstract objects that are referenced by the variable. Then the type-state of these abstract objects is updated. The type-state analysis now invokes the transfer function of their type-state automaton with all possible previous type states (which it has saved) and the method's name (which is saved in the node). The new type states are then saved for future use. Afterwards the analysis continues with the nodes that are next in the graph.

4.4.2 Alias Analysis

The alias analysis part works similar. It saves which variable references which set of abstract objects. There are potentially several abstract objects for each variable, as for example the variables might be assigned to a different abstract object during each loop iteration. Whenever the analysis does not know how often a loop iterates (and it almost never does), it has to assume that the loop (potentially) terminates after each iteration. Thus after the loop each abstract object that is referenced by the variable in the loop, is potentially referenced by it after the loop. In addition to that the analysis also counts how often an abstract object is referenced, if this counter becomes zero the object is going to be garbage collected. Because the type-state analysis has to check that the abstract objects are in the correct type state before being garbage collected, it is informed about all those abstract objects by the alias analysis.

During the first phase of the analysis nodes are inserted into the extended control flow graph, just like in the type-state part. However, this time the nodes do not save the method name but the variable which is assigned and the sources from which the abstract objects might be assigned. These sources are either other variables, new statements or the return values of method invocations. New statements and return values are always assumed to be unaliased with any other abstract object. Clearly, in the case of return values this is unsound but very common and easily fixable by inserting runtime checks (which are currently not inserted by our implementation). Additionally, at the end of each scope a node is inserted that contains all the variables which scope ends.

Again, the actual analysis happens in the second phase. Whenever a alias node is reached, the abstract object sources saved in the node are evaluated. If they “produce” fresh abstract objects, those are introduced to the alias information. The counter of all abstract objects that are now additionally referenced by the variable is increased by one respectively set to one if they are fresh. Then the counter of the abstract objects that were previously referenced by the variable are decreased by one. If any counter reaches zero the type-state analysis is informed, so that it can check that the type state of the abstract object is in a state that allows garbage collection. As a last step the alias information which reference the variable has is then updated. The analysis then continues.

Whenever the analysis reaches an “end of scope” node in the graph, all the counters of the abstract objects referenced by the variables are decreased by one and the type-state analysis is informed (if any counter reaches zero).

4.4.3 Data-Flow Type-State Relation Analysis

The current implementation limits the relation between data flow and type state to the return values of boolean methods inside of conditions. The decision if such a relation exists can already be made during the first phase of the analysis. If any method whose return value offers insight on the receiver's type state contributes to a condition, two nodes are inserted into the extend control flow graph. The first is inserted as the first node in the true branch and the second as the first node in the false branch. The method's specification provides the relation which the

return values has. The specification lists all the possible type states for the return values true and false. The variable (receiver of the method invocation that provides the information) that is affected and the possible type states are saved in the node.

Once these nodes are reached during the second phase of the analysis, the alias analysis is queried to find out which abstract objects are referenced. The type-state analysis information of those abstract objects is then filtered against the possible type states (which are saved in the node). This means that no state is added but that those type states that are not mentioned as possible are removed from the type-state analysis.

4.4.4 General Remark

The analysis terminates once a fix-point over the complete method is reached.

Separately, the type-state analysis is useless, as it only knows in which type state the objects are in and which variables (objects that are not assigned to any variable are internally handled with unnamed but distinct variables) are updated. To find out which abstract objects are referenced by the variables, it needs to query the alias analysis. The data-flow type-state relation analysis is a refinement of the type-state analysis and depends on the alias analysis. So alone it is also useless, but it could be used to predict the return value of method invocations and therefore reduce the number of possible execution paths.

The alias analysis on the other hand could be used by any other analysis that requires aliasing information. A full data flow analysis, for example.

5 Implementation

5.1 Overview

The type-state checking algorithm is implemented as a Java Annotation Processor (a standard Java API [4]) using the Checker Framework [15] (and JSR 308 [8] [7])

The annotation processor is run as a plug-in to the compiler. First the compiler runs, and aborts if it finds any errors. If there are none, it runs the annotation processor, which gets the abstract syntax tree of the program. Because we decided to type-state check per object/class, the algorithm splits the abstract syntax tree for the whole program into abstract syntax trees for each class. Each class is then parsed and a control flow graph describing the alias and type-state changes is constructed.

5.1.1 Initialization

The code of the analysis algorithm is executed as part of the Java compiler. As explain in Section 2.8 we use the Checker Framework to do all the necessary initializations. After the framework has been initialized the control is passed to out analysis, which then parses the abstract syntax trees, extracts each class and verifies the type-state correctness of each method.

5.1.2 Preparing the Analysis

In order to verify the correctness of a program, the abstract syntax tree is converted into a extended control flow graph for each method of a class. The graph corresponds to possible execution paths and the nodes represent any the splitting and merging of the control flow, special nodes also represent program parts that are relevant for either the alias analysis or the type-state analysis.

In order to convert the abstract syntax tree to the control flow graph all loops, switches, continue, break, return, try-catch blocks, throw, if-then-else expressions and conditional expressions into graph edges, assignments and object creations into alias graph nodes and method invocations into type-state graph nodes. To allow the algorithm to use more efficient iteration schemes and easier labeled statement handling, additional nodes are introduced which handle control flow splits and merges but do not semantically change the graph. For more details see below.

5.1.3 Type-state Analysis

The constructed control flow graph is always (by construction) in a specified shape. It has one entry point, which type state is defined by the receiver and the parameter type-state requirements of each method. Furthermore the control flow graph is limited to up to two exit points, one for normal method termination via a return statement or the end of the method and possibly one for a method termination with an exception thrown. Only explicitly thrown exceptions are considered. The type state which is known for the method entry is now pushed towards the exit points until no further changes are observed. If any object misuse is witnessed it is reported via the compiler's error report mechanism. Here, two cases are distinguished. First the algorithm can prove that the misuse will lead to a protocol violation. Second that it can not prove that the object's protocol is followed. In the first case the algorithm will report an error whereas in the second it can either report an error or it can report a warning. This is especially helpful if the developer wants to focus on those bugs for which the analysis can prove that they exist, before spending time on those error reports for which the analysis is uncertain if the usage is correct.

5.1.4 Tests

For most parts of the algorithm there are JUnit tests which test certain language constructs and type-state concepts. Most bugs found during the development of the verifier were also converted into tests. Currently more than 90 tests exist which have a code coverage above 70% even though debug and report code is not tested.

5.2 Specifying the Type-state Model

5.2.1 Annotations

To specify the type-state model, the program that is to be verified has to be annotated. The annotations are provided by the type-state analysis algorithm's implementation. Therefore at least the classfiles of the annotations have to be in the classpath of the program that is compiled and verified. Our algorithm provides a variety of annotations from which the developer can choose. Each of those can be used to specify a certain kind of expected type-state behavior.

The annotations must provide a way to specify which states of the type state are meant. As Java restricts the data types possible passed to annotations, the states can be either represented as Strings or as Enums. We choose to represent them as Strings. Enums have the disadvantage that, because they are used inside the program that is being checked and inside the analysis, they have to be known (at compile time) to both. This makes creating a new type-state automaton more laborious than with the use of Strings. The disadvantage of using Strings is that there is the possibility of misspelling the states, but in most cases this will quickly lead to type-state errors which can be easily recognized as being caused by misspelling some type-state name. The only requirement on those Strings is that they do not begin with "TS ", as those are considered to be reserved states that only the algorithm itself may use. Using such Strings as a name for a state may lead to undefined behavior.

5.2.2 Available Annotations

In the following we present all available annotations and give examples of their use. The examples together (Figure 9 to 14) will present a slightly extended specification of the previously shown File interface example of Figure 1.

@TypestateChecked: This annotation tells the type-state verifying algorithm that all methods of this class or interface have to be annotated. If this is not specified methods without annotation are assumed to have no effect on the type state.

@NoStateChange: The Checker can be put in a mode that enforces annotations. In this mode every method has to be annotated. Methods that do not change the type state will have to be annotated with this annotation.

@ScopeEndStates: If a usage protocol requires the object to be in one or several specified states at the moment at which no reference is left which points to it, use this annotation.

```
@TypestateChecked
@ScopeEndStates( are="Closed" )
interface File {
```

Figure 9: Example of a *@TypestateChecked* and a *@ScopeEndStates* annotations, specifying the File interface shown in Figure 1

@ChangeState: Changes the type state from one specified state to a list of states. If this method is invoked the analysis verifies that the object is in the from state and then sets the new type state.

```
@ChangeState( from="Closed", to="Opened" )
void open();
```

Figure 10: Example of a *@ChangeState* annotation, specifying the type state for the method *open* of Figure 1

@RequireState: Require one specific or one out of several specified states. This annotation is equivalent to a *@ChangeState* annotation form and to the same state.

```
@RequireState( isIn="Opened" )
void read();
```

Figure 11: Example of a *@RequireState* annotation, specifying the type state for the method *read* of Figure 1

@SetState: Set the type state to a specified state no matter which state the object is in (overwrites even the hidden error state)

@TS: This is a wrapper for several *@ChangeState* annotations, because Java does not allow

```
@SetState( to="Closed" )
void reset();
```

Figure 12: Example of a *@SetState* annotation, specifying the type state for a method *reset* that resets the type state to “Closed” even if it was in an error state before

several annotation of the same annotation type. Naturally only one of the *@ChangeState*’s “from” states has to be fulfilled by the receiver. It is also possible to have the same “from” state several times if needed.

```
@TS({
  @ChangeState( from="Opened" to="Closed"),
  @ChangeState( from="Closed", to="Closed")
})
void close();
```

Figure 13: Example for several *@ChangeState* annotations using the *@TS* wrapper; specifying the type state for the method *close* of Figure 1

@ReturnValue: Verify that the return value of the method that is always in one of the states that are specified.

The *@RequireState*, *@SetState* and the *@ChangeState* are not only used to annotate the type-state automaton but are also used to annotate method parameters. The algorithm then also enforces

```
@ReturnValue( isIn = { "Opened", "Closed" })
File getSomeFile();
```

Figure 14: Example of a `@ReturnValue` annotation, specifying that the `File` returned by `getSomeFile` is either “Opened” or “Closed”

that all arguments passed as method parameters that are annotated with the `@RequireState` and `@ChangeState` passed to the method are in the required type state and in the case of the `@ChangeState` and `@SetState` annotation updates the type state of the passed argument. Furthermore parameters with the `@SetState` annotation are initialized to an undefined type state, meaning the first method that changes the type state for those parameters has to be a method that is setting the type state and not changing it.

5.3 Constructing the extended Control Flow Graph

5.3.1 Init

Because our type-state verification algorithm is implemented as an Annotation Processor, we do not have to write a parser, but can instead use all compilers that support the Annotation Processing API. After the compiler verified that the program is valid Java code, the Annotation Processor is invoked. The program that is to be checked is passed to our analysis by the compiler as an abstract syntax tree. The abstract syntax tree is represented by a set of (root) nodes each of those represents a compilation unit (for example a package). The nodes contain additional information about their content (such as Name) and the subtrees (such as all classes implemented in a package) if appropriate. The Analysis then splits the abstract syntax tree into the class subtrees and for each of the class subtree the methods subtrees are visited and the an extended control flow graph for each method is created, which captures all relevant information about all the possible flows of execution within that method and their effect on the type state and aliasing and is later used to verify the type-state correctness of the program. We now describe the construction of the extended control flow graph.

5.3.2 Methods

Each AST node that corresponds to a method is converted into a graph node that “begins” the graph, a node for normal termination and a node for termination by exception are created. The begin (or entry) and the two end nodes are connected by a subgraph that is created for the method body. Figure 15 shows how the method graph looks like. The grayed rectangles stand for a subgraph whereas the white stand for normal graph nodes. The Exception node is grayed out as there may be no exception thrown. In this case the exception node will not be connected and will be disregarded during the analysis of the program.

Some parts of the method body must be analyzed and converted into the method body subgraph of the Control Flow Graph in order to be able to perform the type-state analysis later. Some parts of the method body can be removed without affecting the verification process. The following sections describe all language constructs that need to be preserved. The “Pre” Box is always the last node of the previously analyzed language construct. For the first statement of a method body this will always be the method entry node constructed for the method graph. The “Succ” Box is always the first node of the language construct that is analyzed next, e.g. the next node that is going to be created. If the node is the last node on the execution path it will be

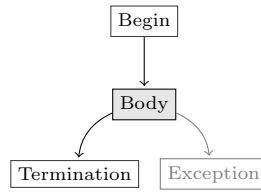


Figure 15: The Control Flow Nodes and Transitions for a Method

connected to the appropriate end node. During normal execution this will always be the method end node, exception if the execution path terminates the method with a throw expression.

5.3.3 Exceptions

In order to handle the control flow of thrown exceptions in the control flow graph construction, the analysis keeps a set of the exception that are thrown in any particular scope. Every time the conversion algorithm converts a try-catch-finally construct, it saves the exception set and creates a new empty set. It then converts the body of the try block into the corresponding subgraph. The analysis then continues within the try block and thus fills the set with all uncaught exceptions which occur in the try block and with their type-state and aliasing information. All uncaught exceptions are then checked against the catch statements. If a match is found the subgraph for the catch is created and then graph node leading into the exception is connected to the catch. The remaining exceptions are merged with the saved exception set. The merged set is then used as the working set. Figure 16 illustrates the created graph.

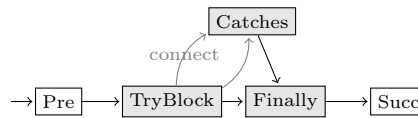


Figure 16: The Control Flow Nodes and Transitions for Exception Handling

After the try block is analyzed and there is an execution path exiting it normally it is connected to the finally block. The finally block is only included in the graph if either a catch or the try block is connected to it. Then the process continues with the next statement. Figure 16 illustrates the created graph.

In the next figures the exceptions handling will be only sketched the possible exception graph edges will be gray lines to a small gray box.

5.3.4 Conditionals

Exceptions are not the only way for the control flow to change. There are also conditional expressions, that depending on the value of the condition take different paths. The simplest are the conditional expression (“? :”) and the if-then-else construct. Depending on a boolean condition one of the two possible execution path is chosen. Both the conditional expression and the if-then-else statement lead to the same graph, which is shown in Figure 17. Where the if has only two distinct branches. The switch statement has several cases. Each case can either fall through (aka continue) with the next state or jump to the end of the switch. An example

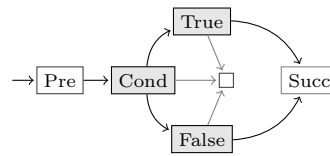


Figure 17: Control Flow Nodes and Transition for the Conditional Expression ? and the If-Then-Else Construct

graph for a switch with three cases, in which the second falls through to the third is shown in Figure 18.

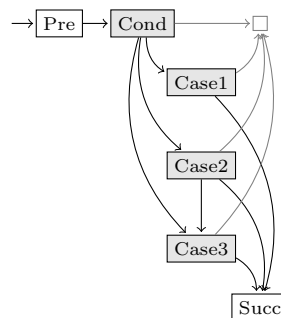


Figure 18: The Control Flow Nodes and Transitions for the switch statement

5.3.5 Loops

Java provides four different kinds of loops, all of which have a different control flow. The do-while loop has the simplest control flow. The loop body is executed then the loop condition is evaluated or in case of a break the control flow directly continues with the subsequent language construct. After the condition the control flow splits in two, one edge goes back to the beginning of the loop body the other edge goes to the node of the next language construct. The corresponding graph is illustrated by Figure 19. The while loop has a similar but reversed control flow order.

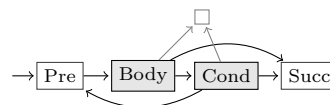


Figure 19: The Control Flow Nodes and Transitions for the do-while loop

Just like the do-while loop the body of the while loop can have several edges pointing to the condition. The first option is the normal termination of the loop body the second option is via a continue statement. This is illustrated by the introduction of the End node in Figure 20. In the actual implementation the End node is introduced, so that the break statement can be connected to it. The figure for the for loop (Figure 21) show this aspect.

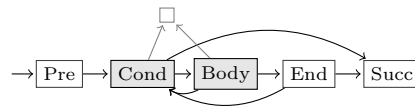


Figure 20: The Control Flow Nodes and Transitions for the while loop

The for loop is also different from the while loop, it has an initialization expression which is only evaluated once. And it has an additional subgraph for the increment part of the loop.

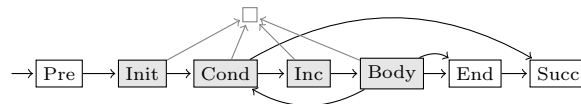


Figure 21: The Control Flow Nodes and Transition for the for loop

The last loop, the for each loop, is a little more complex than the previously considered loops. It will be discussed later, as it needs some feature that are not explained yet.

5.3.6 Break, Continue and Return

Inside the loop body the break and continue statements can not only terminate or start the next iteration of the enclosing loop, but also any enclosing loop that has been labeled. In the case of the break statement we jump to the end of the body of the labeled loop,

whereas the continue statement jumps to the beginning of the loop, to start the next iteration.

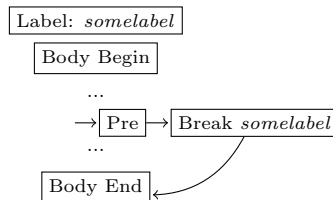


Figure 22: The Control Flow Nodes and Transitions for the break statement

The return statement terminates the method. The execution leading into the return is connected

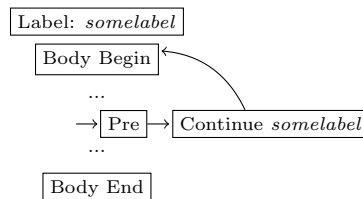


Figure 23: The Control Flow Nodes and Transitions for the continue statement

with the subgraph for the return value which in turn is connected to the Method Termination node.

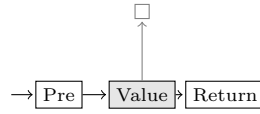


Figure 24: The Control Flow Nodes and Transitions for the Return statement

5.3.7 Aliasing Specific Node

Up to now only the process of creating nodes and edges to handle the control flow were described. But in order to use the constructed graph in the type-state analysis algorithm, we need to extend it with additional information. A prerequisite of tracking an object's type state is to know which variable points to which objects. For this purpose every variable declaration is registered. Variable declarations that are not initialized are naturally analyzed as not pointing anywhere.



Figure 25: Additional nodes and transitions for alias handling of variable declarations

If there is an initializer, its subgraph has to be added to the graph before the variable is registered. The initializer also has to be analyzed in order to find out which sources of abstract objects are possible. The exact abstract objects will only be available during the actual analysis. An assignment is handled very similar. The initializer (respectively the expression that

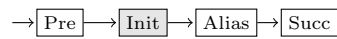


Figure 26: Additional nodes and transitions for alias handling of assignments

is assigned) is analyzed) and then the abstract object sources are “assigned” to the variable. The only difference is that (possible) existing previous assignments will be overwritten.

5.3.8 Method Invocation

After the algorithm is now able to establish to which objects the variables point, it also has to update and keep track of their type state. As our type-state analysis disallows direct field access that change the type state of the object, the only way to manipulate the type state of an object is via method invocations. All subgraphs for the parameters passed on to a method invocation

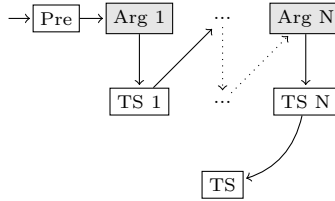


Figure 27: Additional nodes and transitions for type-state verification of parameter passed and the method invocations itself

have to be added. If the method requires certain parameters to confirm to certain type state these have to be checked, too. Then the type state of the receiver can be checked and if necessary it can be updated.

5.3.9 For Each Loop

Now we come back to the for each loop. It is different from the other loops because on each iteration it introduces a fresh object into the loop body while all other variables remain in the state of the previous iteration. On every iteration the aliasing of the loop variable has to re-established, as a new object is assigned to it. The type state also has to be re-established respectively set to the type state that is annotated to the loop variable. Its graph is shown in Figure 28.

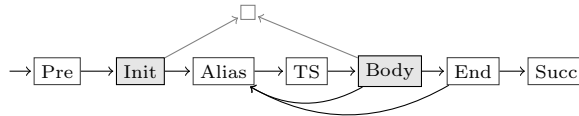


Figure 28: The Control Flow Nodes and Transition for the for each loop

5.3.10 Data-flow Nodes

In the true respectively false branches of an if or conditional expression there will be data-flow nodes that filter the type state of objects that are influenced by the evaluation of the condition. If the evaluation of the condition indicates that several type states of one (or several) object are impossible those are removed (if present) from the analysis.

5.3.11 Expressions

In general all other expressions will not be part of the constructed graph. But nonetheless each expression has to be analyzed nonetheless because some of its subexpressions may be expressions

which are listed above. Additionally all initializers for variables and loops have to be analyzed with if aliasing is produced and the possible type state of the possibly returned objects.

5.4 Analyzing the extended Control Flow Graph

Each of the nodes created in the previous sections “knows” exactly how it, respectively the code it represents, effects the analysis results. In addition each node also knows which if it has any successors. The analysis is then computed by the nodes circulating a data structure which stores the current state of the analysis. When the current analysis information is passed to a node, the node will

- compute the effect it has under the current analysis information information,
- update the current analysis information is according to the kind of node,
- (in case of a fix-point node)
check whether the current analysis information is a subset of previously encountered analysis, and if it is not pass a copy of the current analysis information to each of its successor nodes.
- (all other nodes)
pass a copy of current analysis information on to each of its successor nodes.

In order to analyze a method, a data structure that maintains the current state of the analysis is constructed (without any information in it) and then passed to the “method entry” node. The first couple nodes then initialize the analysis data, e.g. set the parameters to the correct type state. The remaining nodes then will perform the analysis and report any error they might uncover.

Each node is specialized and only updates one particular part of the analysis and they can therefore be grouped into four categories: The nodes for special control flow handling, nodes for updating the aliasing information, nodes for updating and checking the type state and nodes that update the data-flow analysis and check for the effect it has on the type-state analysis.

5.4.1 Control Flow handling

All the nodes have to “know” their successors to be able to pass the current analysis information on. Therefore, the interconnection of all nodes constitutes the control flow information.

To ease the construction of the extended Control Flow Graph it is helpful to have nodes that do nothing but pass the analysis on to its successors. These Dummy nodes are used in the case of the method entry/end nodes and with some control flow changes where it is not immediately clear where the control flow will end up (e.g. throw instructions).

It is clear, that if all nodes simply pass on the analysis and there is a loop in the graph the analysis would not terminate. Simply checking if the current analysis information is a subset of previously computed analysis information does cost resources, especially since a super-set of the previous analysis information has to be computed and kept for comparison. To minimize the overhead normal nodes do not check if the analysis actually contains some new information but always pass the updated analysis on to their successors. Therefore, the fix-point nodes are inserted wherever a loop exists. As a result the analysis terminates and unnecessary analysis iterations are avoided. The decision where fix-point nodes are really necessary can already be made while constructing the extended Control Flow Graph thus minimizing the resources needed for fix-point checks as those are only performed where they are needed.

5.4.2 Alias handling

The aliasing analysis keeps track of abstract objects. The abstract objects are represented by identification numbers, each variable is mapped (by the current analysis information) to a set (for all possible execution path that lead to the variable under the current analysis iteration) of numbers that identify the abstract objects the variable must point to.

For each variable declaration a node is created that introduces the variable to the aliasing information. When a scope ends a node is created which removes all variables whose scope end here. The nodes also keep track of how many variables reference the abstract objects so that it knows when an abstract object is not referenced anymore. Assignments increase this counter, overwriting by other assignments decreases it. Similarly, when several branches are merged, and an abstract object is assigned to the same variable in both, the counter is decreased. This is also true if a variable's scope ends.

5.4.3 Type-state handling

There exist several distinct type-state “operations” which each have their own specialized nodes. The type-state analysis information is kept as a map that maps the abstract objects ids to their current type-state analysis.

The *@SetState* type-state behavior is implemented by inserting a node that queries the current alias analysis for the abstract objects that can be the receiver and then sets their type-state analysis accordingly.

Similarly the *@ChangeState* behavior is implemented by inserting a node that also queries the current alias analysis and that checks the requirements and then updates the type-state analysis. As previously stated the *@RequireState* behavior is implemented by treating it as a *@ChangeState* behavior from and to the required type state.

Also at the end of scope of each variable, there are nodes inserted, that query the alias information if any of the abstract objects are now without a variable referencing them. Those objects lifetime has ended, and the node checks that they are in the correct type state if any restriction on their type state at the end of the lifetime has been made. Likewise the parameters are controlled to be in the required type states (if any) at the end of the method.

5.4.4 Data-flow handling

Data-flow is actually not really tracked, currently all the analysis does, is to evaluate a condition of an if or conditional expression and inserts nodes at the beginning of the true and false branch that filter the type-state information accordingly.

6 Evaluation

6.1 General Evaluation

The analysis itself is also able to find all type-state errors that are caused by local object misuse, as long as the type-state models are accurate. Through the nature of enforcing local type-state contracts, it is not possible to verify type-state correctness of objects that are spread and are aliased in several classes. To handle this case our analysis would have to be extended to be able to handle the type state of clusters of classes (see Section 7.1.3) and provide ways to specify those.

6.2 Quality Assurance

To verify correct analysis results of our analysis we ran it on several dozen small test programs and a medium sized project.

6.2.1 Small Test Programs

In general, before implementing a feature of the algorithm, we wrote several test programs which contained the features we wanted to implement. The programs then were analyzed by hand so that we knew the correct type-state analysis results. In order to be able to constantly check our analysis results against the expected results we converted all test programs to JUnit [11] tests. As our analysis runs at compile time, each JUnit tests compiles a test program with our analysis being enabled. It then checks if the analysis reports all expected type-state misuses and only those. We currently have over 90 test programs, with a code coverage of above 70%. Though most of the untested code is guarding against unexpected behavior or collects additional information about the analysis itself. Most test programs are small, nonetheless there are a few which are several hundreds of lines long. Currently three tests fail, but they test behavior which is planned to be implemented in the future.

6.2.2 Verifying itself

As our design currently requires that all type-state relevant parts of the program are completely annotated, running it on existing code faces some challenges. Possible future work might significantly ease this problem, by automatically generating the necessary type-state annotation. Therefore, we decided to run the analysis process on itself. The analysis implementation uses many different features of the Java language and includes several classes that have to be used after a certain type state.

Running the type-state algorithm on its own implementation helped to uncover two misuses of data structures. Due to the two phase nature of the type-state verifying algorithm, it has several data-structures that are initialized in the first phase, then the data is converted, respectively analyzed. In the second phase the converted data is then only used but is not allowed to change anymore (as this would require a rerun of the conversion/analysis). Annotating and running the type-state verifier on itself uncovered an execution paths in which the conversion was not done properly and another execution path in which the analysis tried to change an already converted data structure. Both bugs could have led to undefined behavior that would have been difficult to spot manually as these bugs only occurred when certain preconditions were met.

6.3 Performance

Running the complete JUnit test suite takes less than 8 seconds¹, including the JUnit initialization and the normal compilation for the test programs. Running the analysis on its own implementation takes slightly more than 2 times as long as a compilation without it. In general we noticed that running the analysis and the compilation takes from two up to four times as long as a normal compilation². Specifically measuring the time the verification of small method size takes, reveals that it takes around 10 ms for methods below 10 lines and around 20 ms for method around 20 lines. Testing methods with over 500 lines of code and nested if statements and loops, shows that it takes around 500 ms for those. Surprisingly, the dominating cost (even for large methods) is not the analysis itself, but the conversion from the abstract syntax tree to the extended control flow graph which was explained in Section 5.3.

lines of code	analysis time	thereof CFG creation
< 10	ca. 10 ms	6-9 ms
< 20	ca. 20 ms	15-17 ms
ca. 500	ca. 520 ms	ca. 400 ms

Figure 29: Runtime results in dependence on method length.

Even with very long methods (over 500 lines) this conversion still accounts for more than three fourth of the computation. This is probably due to the very deep call chains produced by overwriting the visitor which the Compiler API³ offers.

The rest confirms that it is fast enough to be run at each compilation. The analysis is slowing the compilation of even medium sized projects only by several seconds. The analysis can certainly be used in on the fly developer notifications as in this scenario only very few methods have to be analyzed at a time.

¹On a 2x800MHz Core 2 Duo with 4GB of main memory

²Compiling the type-state analysis implementation takes 2.49 seconds for normal compilation versus 6.42 seconds for a compilation with type-state analysis, on the same machine

³See the `com.sun.source.tree.TreeVisitor` interface for details

7 Outlook

7.1 Future Work

Also the analysis concept proved to be usable as well as computationally feasible there are – as always – many parts that could be extended or improved.

7.1.1 Type-state Contracts

It is clear that some type-state contracts have to be annotated manually, especially those annotation that specify how a class is to be used. But we currently require all methods that take parameters on which they have type-state prerequisites to be annotated. But, fortunately, the type-state requirement of the parameters and the type-state that the return value might be in, can – to a certain extend – be automatically inferred. It should be possible to use the same technique that is currently use to verify the type states and adapt it to generate the required annotations.

7.1.2 Data-flow Analysis

By integrating the simple data-flow analysis described in Section 4.1.2 into the analysis algorithm, it became clear that the type-state analysis can greatly benefit from exploiting the relationship between return values of method invocations and the actual type state of the receiver. As most of the required features are already present in our implementation. Expanding our algorithm's capabilities to perform a data-flow analysis should be feasible and are therefore considered to be implemented during future work. Moreover predicting the return values from the type state of an object is in might lead to more precise analyzes.

7.1.3 Cluster of Objects

One expected problem (Section 4.1.4) is that the presented analysis focuses on verifying objects. And because of the modular and thus local analysis process, it can not verify properties that are spread over several objects. To address this shortcoming one could extend the analysis to be able to group object that have interconnected type-state behavior into a cluster that shares a common type state. In this case method invocation would not alter the objects type state but a type state shared by several objects. In contrast to switching to an inter-procedural context-sensitive analysis this would preserve the modularity (at least as much as the interactions allow) and thus also preserve the lower computational effort while still addressing the problem. The handling of such clusters should be possible to integrate into the alias handling and can potentially use some of the infrastructure needed for expanding the data-flow analysis, because there also objects can have influence on different object's type state. The main obstacle we foresee is the specification of which (static/abstract) objects have which common type state.

7.1.4 Integration in Development Tools

Also the analysis is easy to integrate into existing Java compilers that implement the needed standard APIs[2][4][8] developers could certainly profit from an integration into an Integrated Development Environment. The analysis proved to be even fast enough to type-state check the code written on the fly. And could give better error reports than are possible via the normal compiler error report mechanism. For example it could highlight the offensive execution path.

7.1.5 Collections

Also the handling of collections and arrays is still rudimentary. Currently the analysis can not trace objects inside of collections, once they are inside the collection the analysis can not distinct them. A specialized analysis [12] that can trace those objects to a certain extend possible can improve the range of type-state verification problems we can solve. Already knowing that all members of a list are only once in the list can help improve the type-state verification.

7.2 Conclusion

In this thesis we have presented a context-insensitive, flow-sensitive type-state verifier that uses a combined type state, alias and data-flow domain. The verifier supports almost the complete Java language but most notably does not explicitly support concurrency. We presented an easy yet expressive way of specifying the type-state model with the help of Java annotations. The use cases, especially running the analysis on its own implementation and uncovering some bugs in it, demonstrated that intra-procedural type-state analysis can be useful. The performance is good, especially if only several methods have to be analyzed it is almost instantaneous. Additionally the analysis is easily integrable into the compile process as it is implemented as a Java Annotation Processor.

References

- [1] Manuel Dähndrich and Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast reliable message-based communication in singularity os. In *SIGOPS Proceedings of the 2006 EuroSys conference*, pages 177–190, Apr. 18-21 2006.
- [2] Joshua Bloch. Jsr 175: A metadata facility for the java programming language. <http://jcp.org/en/jsr/detail?id=175>, Sep. 30 2004.
- [3] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages*, pages 260–282, Jan. 29-31 1979.
- [4] Joe Darcy. Jsr 269: Pluggable annotation processing api. <http://jcp.org/en/jsr/detail?id=269>, Dec. 11 2006.
- [5] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *SIGPLAN Proceedings of Conference on Programming Language Design and Implementation*, pages 57–68, June 17-19 2002.
- [6] Robert DeLine and Manuel Fähndrich. The fugue protocol checker: Is your software baroque? Technical report, Microsoft Research, Microsoft Corporation, Jan. 2004.
- [7] Michael D. Ernst. Type annotations (jsr 308) and the checker framework. <http://pag.csail.mit.edu/jsr308/>, Aug. 10 2009.
- [8] Michael D. Ernst. Type annotations specification (jsr 308). <http://types.cs.washington.edu/jsr308/specification/java-annotation-design.pdf>, May 14 2009.
- [9] Stephan Fink, Eran Yahav, Nurit Dor, and G. Rammalingam. Effective typestate verification in the presence of aliasing. In *ISSTA Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, July 17-20 2006.
- [10] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, May 2001.
- [11] Junit – homepage. <http://www.junit.org>.
- [12] Mark Marron, Mario Méndez-Lojo, Manuel Hermenegildo, Darko Stefanovic, and Deepak Kapur. Sharing analysis of arrays, collections, and recursive structures. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–49, 2008.
- [13] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Control Flow Analysis*, chapter 5.1. Springer Verlag Berlin Heidelberg, 1999,2005.
- [14] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Data Flow Analysis*, chapter 2. Springer Verlag Berlin Heidelberg, 1999,2005.
- [15] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

- [16] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing legacy systems: software technologies, engineering processes and business practice*, chapter 1.3. Addison-Wesley Longman, 2003.
- [17] Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. In *IEEE Transactions on Software Engineering*, pages 157–171, Jan. 1986.
- [18] Yue Yang, Anna Gringauze, Dinghao Wu, and Henning Korsholm Rohde. Detecting data race and atomicity violation via typestate-guided static analysis. Technical report, Microsoft Corporation, Aug. 2008.