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

Master thesis

Typestate Inference with Preconditions

submitted by

Klaas Boesche

on February 01, 2012

Supervisor

Prof. Dr. Sebastian Hack

Reviewers

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

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.

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, February 01, 2012

Abstract

Specification inference aims to alleviate the problems in writing specifications and missing specifications for legacy code. A common approach is to infer finite typestate automata that model legal call sequences. This yields intuitive specifications.

I present a new approach that enhances the expressiveness of typestate automata by adding conditions on the parameters of each call. In addition each state is identified by a condition on the fields of the modelled class. The approach makes use of an over-approximate weakest precondition analysis. It explores the state space of the class from the error state with a new algorithm and repeatedly adds states and transitions by analysing preconditions.

With an implementation of the fully automatic inference algorithm I show that the algorithm is practical for real world examples taken from the Java library without modification. Where previous approaches are restricted to a small set of assertions as the specification of the error state, my approach handles all assertions and exceptions of a class in a single automaton. The evaluation of a prototype verifier shows that the generated specifications can be used to find defects in code using the target class.

Contents

1 Introduction

The field of automated program verification has come a long way since the original introduction of the concepts by Floyd, Hoare, Dijkstra and others. Many approaches such as model checking, theorem proving and abstract interpretation have spawned tools for formal verification of software properties which are increasingly employed in industry.

However all of these techniques require a specification of the properties which should be checked for. Missing specifications are a major reason why software verification and advanced testing methods are not used more often [DKM+10, WML02].

Writing a specification which can be used for verification is a labour intensive process. The required effort should be reduced for verification to gain more widespread acceptance.

As modern software development also extensively builds upon previously written components for which no specification may exist, the difficulty of creating and maintaining formal specification increases. The effort is thus often avoided in new projects or is abandoned during the process in favour of more traditional testing techniques.

To support legacy components in verification and ease specification writing a number of *Specification inference* or *mining* algorithms [BHS07, HJM05, NGC05, WML02, ACMN05, RGJ07, SYFP07] were developed in recent years.

These compute a part of the specification that is often not documented: In which order can one call methods of the component without violating some implicit constraint? A Stream object, for example, must be connected to a Sink by a call to `connect(Sink)` before the method `write(byte[])` may be called on it.

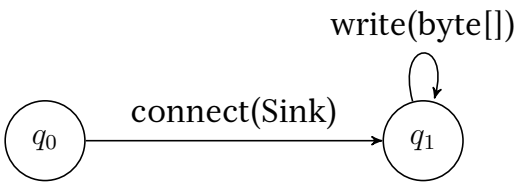

Figure 1.1: Typestate automaton for a Stream object

The specification is modelled as a finite state automaton or regular language that represents call sequences which do not lead to an error state. In Figure 1.1 the automaton for a Stream object is shown. Each transition not in the automaton is assumed to be erroneous. This example is a reduced version of automata for `java.io.PipedOutputStream` from previous work [BHS07, ACMN05] in which I omitted some methods of the class for clarity.

1.1 Contributions

In this master thesis I make the following contributions towards verification of modular software components and specification inference in Java.

I implemented and designed a static *precondition analysis* that computes an approximation of the weakest precondition of a method call with regard to a postcondition. The precondition analysis operates on the implementation of a logic modelling all features of the Java language. For satisfiability and validity checks on formulas of the logic, I added a translation to the CVC3[BT07] theorem prover.

I present a new *automaton inference algorithm* which automatically extracts a finite state specification of a class. This algorithm takes the assertions as specification of an error state. From this state it repeatedly uses the precondition analysis on the methods of the class to find additional relevant states and illegal call sequences.

The main advantage of the new approach is that each transition in the automaton includes a condition on the parameters that guards the transition. This allows the automaton to model all details of call sequences leading to any assertion violation in a single automaton and in a readable manner. In addition each state represents a set of runtime program states with a formula over the fields of the class. Unlike previous static approaches, the algorithm also handles the special properties of constructor methods to provide an initial state and model conditions on constructor parameters.

The automaton for the Stream class is shown in Figure 1.2. Each state is labelled with a predicate on the state of the object and each transition includes a condition on the parameters or none if the condition is `true`. The error state is made explicit and labelled with the `ExceptionThrown` predicate.

In the evaluation I show that the implementation of the precondition analysis and the automaton inference can handle classes from the Java standard library without modification and works completely automatic.

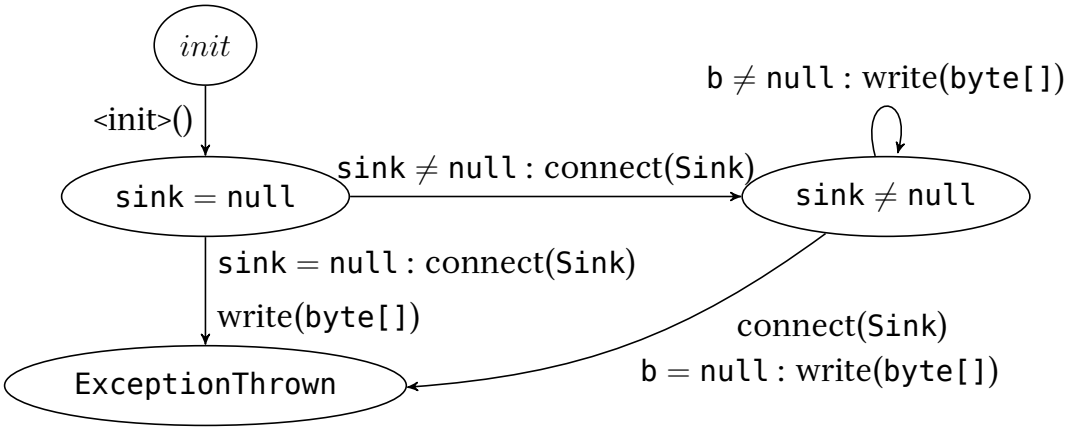

Figure 1.2: Stream automaton

To show the applicability of these automata in verification I implemented a prototype verifier that checks code using the library class against the automata. It finds defects in the code by tracking field values and automata states.

2 Background and Previous Work

In this chapter I describe the concepts behind precondition analysis and specification inference. I discuss previous work in these areas and provide background on the intermediate representation the implementation is built upon.

2.1 Preconditions

This thesis builds on the weakest precondition calculus introduced by Dijkstra [Dij68] and extended by Gries [Gri87] and many more.

A *precondition* $pre(S, \phi)$ with regard to a program statement S and postcondition ϕ satisfies the following: If the state before the execution of S satisfies $pre(S, \phi)$, then the execution of S terminates in a state which satisfies the postcondition ϕ.

The definition of state in this context depends on the execution environment. In Java the state refers to the values of local variables and the values of objects and their fields in the heap.

The *weakest precondition* $wp(S, \phi)$ is the precondition implied by all other preconditions. A precondition a is thus *weaker* than another precondition b if $b \Rightarrow a$. This relation is a partial order on the preconditions.

Consider the statement `x = x + 1;` and the postcondition $x > 5$. Then some preconditions are $\{x = c, x > c \mid c > 4\}$. The weakest precondition is $x > 4$.

The weakest precondition calculus is a set of *predicate transformers* that compute the weakest precondition for a statement and a given postcondition. The weakest precondition of an assignment statement $wp(\texttt{x = e}, \phi)$ for example, is defined as $\phi[\texttt{e}/\texttt{x}]$ where `e` is substituted for `x` in ϕ.

In the case of loops the weakest precondition requires a loop invariant which is maintained by the statements within the loop in each iteration and a loop variant which is decreased in every iteration to ensure that the loop terminates. For arbitrary programs loop invariants and variants cannot be computed, but some approximation of it may be derived.

2.1.1 Precondition analysis

The precondition analysis builds on the theory of abstract interpretation [CC77], a theory of approximation of the concrete semantics of a program. An abstract interpretation makes use of a mapping of the program states to an *abstract domain* such that a set of concrete program states is represented by an element in the abstract domain. This also defines the approximation, as there is no one-to-one mapping from the abstract domain back to a program state. The abstract interpretation then defines the semantics of program statements within the abstract domain. The semantic rule associated with a particular statement is also called a *abstract transformer* as it transforms an element in the domain into another. As the limits of loops and recursion may not be decidable the semantics require the computation of a greatest fixed point over the transformers.

An abstract interpretation with the goal of inferring preconditions of Java method calls can be defined using a form of the weakest precondition calculus as the transformers. The values of the abstract domain are formulae in a logic that can encode the features of the Java program.

In [CCM10] Cousot et al. give background on the use of first-order logical formulae as domains in abstract interpretation. Abstract domains consisting of first-order formulae are usually not finite. While $true$ is the largest element in a domain of formulae ordered by $\Rightarrow$, there are chains of successively larger formulae such as $x > 10 \Rightarrow x > 9 \Rightarrow ....$

To ensure the termination of the analysis, an operator called a *widening* is required. When during the iterative computation of the fixed point a widening, is applied it maps the current and future values of the abstract domain to one larger than both with the intent of reaching a fixed point. A trivial example in the case of first-order formulae is the widening which always returns $true$.

Previous work by Lev-Ami et al. [LASRG07], makes use of a special logical domain of predicates that implement heap structures to compute quantified preconditions, but requires the definition of which heap structures to analyse.

Another approach by Cousot et al. [CCL11] also uses specialized domains to infer preconditions from assertions in collections to generate contracts for runtime checking.

The work closest to this analysis in terms of the abstract domain used and the practical application of the analysis is the work by Chandra et al. on their tool Snugglebug [CFS09]. While the logic is inspired by their work and theirs is also an interprocedural backwards analysis, their goal is not to derive a precondition guaranteed to be weaker or equal to the weakest precondition,

but a stronger or equal precondition. In particular, they employ a variety of search heuristics that try to find a stronger precondition such that the analysis may quickly terminate.

2.2 Specification Inference

Approaches to specification inference can be broadly classified into dynamic or static and client- or component-side analyses.

Static approaches [ACMN05, NGC05, BHS07, HJM05, WML02, SYFP07, RGJ07] generate specification models from symbolic analysis of code at compile time. They do not necessarily require executable code.

Dynamic approaches [WML02, DKM+10] observe program executions and record executed call sequences. This results in a high precision of the generated models in that only actual usage is presented. However, the models are likely to be incomplete and especially may not include usage that leads to errors [DKM+10]. As a component usually cannot be executed on its own these approaches are also client-side approaches.

Client-side approaches [SYFP07, RGJ07, DKM+10] make use of existing client code or test cases or generate additional tests[DKM+10] and extract possible call sequences. These approaches may also be called *mining* algorithms in the literature.

Static client-side approaches [RGJ07, SYFP07] make the assumption that the analysed client code models valid usage of the component. As such they represent a static view on the actual usage.

Component-side approaches [ACMN05, NGC05, BHS07, HJM05] only analyse the code of the component and can thus employ static analyses only as a software component is usually not executable on its own. As such they make worst case assumptions about the client's behaviour. Thus there may be call sequences marked as erroneous which could still be valid for some client. Checking any client against a soundly computed specification would find all errors however and can prove safety of the client's library usage. Client-side approaches cannot deliver this guarantee.

The dynamic approach by Dallmeier et al. [DKM+10] mines an initial model from an executable client or test suite. Their approach then generates additional client code by mutating the original client. From executions of the mutated versions they add missing states and transitions to the initial model. They show that this technique increases the completeness of the models and especially adds transitions to the error state. These were often missing from the initial models as the test suites rarely tested for exceptions.

Whaley and Lam [WML02] were the first to compute finite state machine representations of call sequence constraints for classes. In their representation each method corresponds only to a single state. This yields an imprecise model and thus they split the model based on the class hierarchy and field accesses. Additionally their approach combines a static, component-side analysis with a dynamic mining algorithm.

Alur et al. [ACMN05] use a learning algorithm for regular languages. The algorithm repeatedly queries a model checker whether the currently found automaton is safe with respect to the component and whether the model is approximately complete. The first query provides the algorithm with a counter-example call sequence to remove from the automaton, the second allows it to add call sequences. Their models additionally distinguish transitions based on the return value of the method call. The user of their approach is required to enter a set of predicates with which a finite abstraction of the component is built and a single exception predicate which signifies the error state. This requires multiple models for a complete specification of all erroneous call sequences.

Nanda et al. [NGC05] simulate the client behaviour with an increasing number of objects from different but interacting classes, such as a set and its iterator. This results in complex models describing effects of calls on one object on the state of other objects. Their approach also abstracts the input classes with respect to the value of predicates but depends on an initial phase in which predicates are computed from the classes.

Henzinger et al. [HJM05] propose an approach where abstract versions of the component are iteratively refined using counter-examples. These are generated from checking whether a candidate automaton is safe with respect to the abstraction and whether it permits all legal call sequences. The candidate automaton is recomputed and checked under the refined abstractions.

Beyer, Henzinger and Singh [BHS07] compare the previous approaches from [ACMN05, HJM05] and another algorithm in a unified formal setting and on a small number of Java classes transformed into this setting manually. They show that with regard to theoretical as well as practical time complexity each algorithm can outperform the others.

2.3 jFirm Intermediate Representation

jFirm is an intermediate program representation (IR) for bytecode. It is descended from the libFirm IR[BBZ11] but reimplements the concepts with operations more specific to bytecode. Where libFirm is written as a C library

and generic IR for compilers, jFirm is written in Java as a tool for static analyses on bytecode.

Both are based on the concept of Static Single Assignment (SSA) form, in which local variables only have a single definition in the representation. This also requires the addition of ϕ functions which select variable definitions from predecessor blocks based on the actual control flow at runtime.

jFirm is graph based in that each operation is a node that directly depends on its operands via edges. All dependencies between nodes have been made explicit via edges. In particular memory dependencies are made explicit via memory values.

An operation modifying the heap, such as a store to a field of an object, takes a memory operand and produces a new memory value on which other operands may depend. This allows the intermediate representation to directly encode that some memory-dependent operation may not modify the result of another if an analysis can prove this.

Nodes do not generally have a fixed order in which they must be executed, except for the requirement that operands must be computed first and in order. However, the nodes are embedded in a control dependency graph of basic blocks where a block is control-dependent on nodes that have execution type such as unconditional jumps. In particular jFirm also makes exceptional control flow explicit.

Figure 2.1 shows a simple getter method in the jFirm representation. The dashed edges represent control dependencies and the solid edges data correspond to data dependencies. The jFirm graph for each method has a dedicated starting block containing the parameter (`PARAM`) nodes and an `END` node which acts as a sink for the memory values which may be alive after the execution of the method.

In jFirm each node produces exactly one value, which may be a tuple. This is the case with nodes 14 and 21 which have directly dependant nodes that project parts of the tuple. In the case of node 14, the projection of `DATA` returns the value of the field `algorithm` of the object at operand 1. The projection of `EXC` produces a jump to block 5 in case the operation throws a NullPointerException. This block contains the exceptional return where the exception value is re-thrown to the caller. In this case an analysis could prove that node 14 may never throw an exception, as **`this`** can never be **`null`** and thus the control flow from node 18 to block 5 could be removed as well as the ϕ node 7. The `CONT` projection of node 14 jumps to block 20 in case no exception was thrown.

In Java even a **`return`** may throw an exception if some some constraints on the state of monitors are violated. In this thesis I assume that such exceptions never occur, as the approach does not take multi-threading into

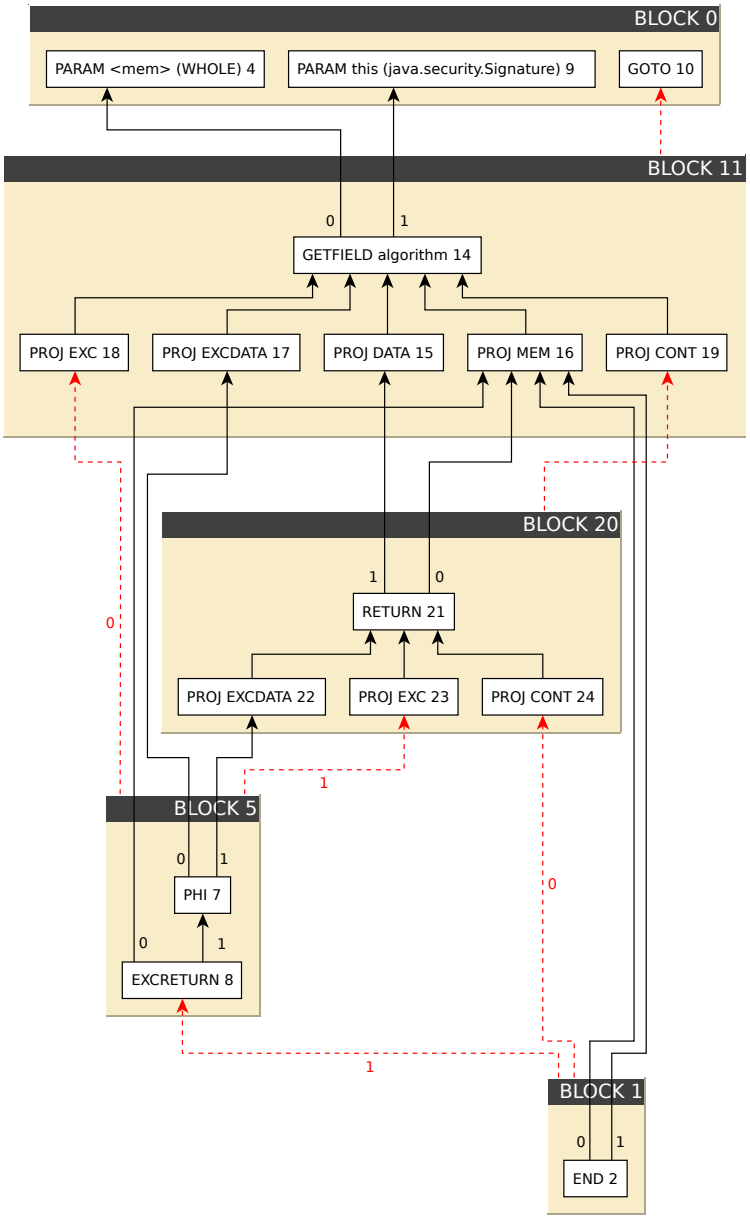

Figure 2.1: The jFirm graph of java.security.Signature.getAlgorithm()

account.

The main advantages of using jFirm for static analysis are that all effects of operations are made explicit by dependencies in a graph. Additionally, the large amount of bytecode operations that differ only in argument types such as array loads for different primitive arrays and calls for different types of methods are handled by a much smaller set of operations.

3 Approach

The first part of the approach is a precondition analysis similar to Snugglebug by Chandra et al.[CFS09]. However, instead of finding preconditions that may be stronger than the weakest precondition we aim to find at least the weakest precondition. This ensures that the precondition represents all possible program states leading to an error.

In the context of the set of program states encoded by a certain precondition, we call a weaker precondition an over-approximation of the states that may reach a postcondition. A precondition of a method call with regard to the error postcondition must thus represent at least the set of all states that ensure that the error state is reached to prove that a certain program state is not contained.

The second part of my approach will extract a model of erroneous call sequences by exploring the state space of a single object of a target class. Beginning with the error state it finds predecessor states and incoming transitions by examining preconditions of method calls. From these states further predecessor states are found via preconditions of method calls. Finally the algorithm examines which states are reachable by computing and analysing preconditions of constructor calls.

3.1 Precondition Analysis

The precondition analysis is a backward abstract interpretation of blocks in the control dependence graph. In the following I define the logical domain and the abstract semantics of jFirm operations as transformers on the logical domain.

3.1.1 Logical Domain

The domain consists of formulae in a first-order logic without quantifiers. A limitation of the domain is that it does not correctly model overflow and floating point types and as such the analysis may be unsound and/or imprecise. This does not limit the approach in general[BKW09] and is .

The logic is inspired by that of the Snugglebug [CFS09] tool. The main difference is with regard to the intermediate representation, as there are no local variables in the jFirm IR and no fixed execution order of the operations within a block. Each data dependence edge can be seen as an assignment to a variable x_n representing the value of the dependant node n. The precondition at the start of the method will never contain such variables, except for those representing parameters.

The analysis handles Java operations that are not implemented in the logic using `unknown` terms that may have any possible value of a certain type. The only operators missing in the logic are `%`, `&`, `|`, `^`, `>>`, `>>>` and `<<`. Terms with `unknown` value are also used for the return values of calls that are not analysed, such as calls to native methods. New `unknown` values are created and thus indexed by a new $id \in \mathbb{N}$ to guarantee that these values may be equal to other values of the particular type but cannot be proven to be equal to other `unknown` values of that type.

In the logic the memory is modelled using the theory of arrays [SDBL01] an instance of the theory of uninterpreted functions. In Java the memory can be separated into disjoint parts by fields, as the field written to is always statically determined and two different fields may never refer to the same memory location. This introduces a primitive form of alias analysis by types into the logic and thus reduces the complexity of terms built during analysis. Each field f defined in any class C is thus represented as an array, indexed by references to objects that include the field in their Java runtime state. In the following I will call such arrays, implementing disjoint parts of the Java heap space, *memory arrays*.

The same can be done for primitive array types, but not for array types with reference element type. As an example consider that an array of type `Integer` might alias with an array of element type `Object`, but never with an array of **`int`** elements.

Array accesses are two-dimensional array accesses in the theory because they are translated to memory accesses. The memory array is indexed by the reference to the array object and the element index of the original array.

Table 3.1 gives an overview of the terms of the logic. Terms named o refer to terms of reference type, i to terms of integer type and v to terms of any Java type. Terms named f stand for terms that evaluate to a memory array in the logic. This is either a memory array constant F_g which implements the memory of a field g or a write `write(`f_1`,`o`,`v`)` where the same constraints hold for the term f_1. The `write` term models a functional update of the memory array f_1, by remapping the index o to the value v. In the imperative setting of Java this corresponds to a field write `o.f = v`.

Terms named a in the table similarly refer to either A_r, a particular A_t or

x_n	Variables for each node n in the graph
`unknown`$_{id,t}$	Variables for over-approximated and thus unknown values of a certain type t
`null`	Constant for the **`null`** object reference
F_f	Memory of field f as an array indexed by an object reference
A_t	Memory of primitive element type t arrays
A_r	Memory of reference element type arrays
T_t	Constant for each Java reference type t
M_m	Constant for each Java method m
`Ret`	Special variable referring to the return value in postconditions
`Exc`	Special variable referring to an uncaught exception
`read(`f`,`o`)`	Field load function where f is an array of **`int`** or reference type indexed by an object reference o
`read(`f`)`	Field load function for static fields
`write(`f`,`o`,`v`)`	Field write function, which returns a functional update of f where o maps to v, that is $f[o \mapsto v]$
`write(`f`,`v`)`	Field write function for static fields
`readA(`a`,`o`,`i`)`	Array read function where a is an array indexed by the array object reference o and the accessed index i
`writeA(`a`,`o`,`i`,`v`)`	Array write function
`typeOf(`o`)`	Dynamic type of the object referenced by term o
`dispatch(`t`,`m`)`	Function to resolve a call to m on receiver type t
t_1 `{+, -, *, /}` t_2	Arithmetic operations
`length(`o`)`	Length of an array referenced by o
`fresh(`id`,`T_t`)`	Uninterpreted function for creating new object constants of reference type t
`freshA(`id`,`T_t`,`t_0`,…,`t_n`)`	Uninterpreted functions for creating new array object constants with array type t and given dimension size terms t_i

Table 3.1: An overview of the terms in the logic.

a `writeA`(a_1,o,i,v). This allows the logic to model sequential updates to memory locations as functional updates of an array, as the logic may not include side-effects.

New object references are created in the logic using the uninterpreted functions `fresh`(id,T_t) and `freshA`(id,T_t,t_0,…,t_n) with fresh $id \in \mathbb{N}$. The id ensures that such object references are never equal.

The `typeOf`(o) function allows the logic to reason about dynamic Java types and assert the type of the objects referenced by the `fresh` functions.

The `dispatch`(t,m) function models the dynamic dispatch semantics of Java calls. As the analysis assumes that only the classes known to it may be accessed, it can assert the value of this function for a given type constant T_t and method constant M_m.

In table 3.2 I give an overview of the literals used in the logic. These are the atomic elements of the first-order logic formulae in the abstract domain. Other than the usual boolean constants, equality predicates and the negation operator, the `subType` predicate takes two terms evaluating to types and implements the Java subtyping relation.

`false`	
`true`	
`subType`(t_1,t_2)	Whether type t_1 is a subtype of type t_2 in Java
$t_1\{\neq,=\}t_2$	(In)Equality of reference and primitive type terms
$t_1\{\leq,<,\ldots\}t_2$	Inequalities of primitive type terms
$\neg l$	Negation of another literal l

Table 3.2: An overview of the literals (or atoms) in the logic.

The axioms of the logic are summarized in table 3.3. Axioms F1 to F3 relate field loads and describe how these terms can be reduced under certain circumstances. Axiom F1 asserts that the fields of newly created objects are initialised with the default values of the field's type while axioms F2 and F3 are the read-over-write axioms from the theory of arrays.

Similarly, axioms A1 through A3 give the axioms of a two-dimensional theory of arrays with default values. Properties of the length of arrays and how it can be extracted from a fresh array reference are handled by axioms A4 and A5. The array axioms A1, A4 and A5 do not handle the case of arrays with multiple dimensions. The basic scheme can be extended by wrapping the `freshA` function in a finite number of array reads ensuring that the indices are larger than zero and smaller than the corresponding lengths given by the parameters of `freshA`.

Axiom S asserts the typing rules of the Java Language Specification[GJSB05]

for the type constants in the logic. Axioms T1 and T2 specify how `typeOf` operates on new (array) object references by extracting the type constant parameter. The Java dynamic dispatch rules are asserted for classes known to the analysis in axiom D.

F1	$\forall t \forall f :$ `read(`F_f`,fresh(`id`,`T_t`))`$= d_f$
F2	$\forall f \forall o_1, o_2 \forall v : o_1 = o_2 \Rightarrow$ `read(write(`f`,`o_1`,`$v`),`$o_2$`)` $= v$
F3	$\forall f \forall o_1, o_2 \forall v : o_1 \neq o_2 \Rightarrow$ `read(write(`f`,`o_1`,`$v`),`$o_2$`)`$=$ `read(`f`,`o_2`)`
A1	$\forall i, l : i < l \wedge i \geq 0 \Rightarrow$ `readA(`$A_{r/t}$`,fresh(`id`,`T_t`,`$l`),`$i$`)`$= d_{r/t}$
A2	$\forall a \forall i_1, i_2 \forall o_1, o_2 \forall v : o_1 = o_2 \wedge i_1 = i_2 \Rightarrow$ `readA(writeA(`a`,`o_1`,`i_1`,`$v`),`$o_2$`,`$i_2$`)`$= v$
A3	$\forall a \forall i_1, i_2 \forall o_1, o_2 \forall v : o_1 \neq o_2 \wedge i_1 \neq i_2 \Rightarrow$ `readA(writeA(`a`,`o_1`,`i_1`,`$v`),`$o_2$`,`$i_2$`)`$=$ `readA(`a`,`o_2`,`i_2`)`
A4	$\forall o :$ `length(`o`)` ≥ 0
A5	$\forall l :$ `length(fresh(`id`,`T_t`,`$l`))` $= l$
S	$\forall t_1, t_2 :$ `subType(`T_s`,`T_t`)` $\Leftrightarrow$ s is a subtype of t
T1	$\forall t :$ `typeOf(fresh(`id`,`T_t`))`$= T_t$
T2	$\forall t \forall l :$ `typeOf(freshA(`id`,`T_t`,`$l`))`$= T_t$
D	`dispatch(`T_t`,`M_m`)`$= M_{m'} \Leftrightarrow$ a call to m on dynamic type t resolves to method m'

Table 3.3: An overview of the logic's axioms. The default value d_f of a field f is $d_r :=$ **`null`** in case of a reference type or $d_t := 0$ in case of any primitive type.

All formulae in the domain F are in disjunctive normal form which has advantages for the analysis as shown by Chandra et al.[CFS09] and is necessary for the automaton inference algorithm in Section 3.2.

The domain is ordered by implication, thus the smallest element is `false` as it implies every other element in the domain and the largest element is `true` since it is implied by every other element. The least upper bound of two elements is their disjunction.

3.1.2 Symbolic Transformers

Before the actual execution, the precondition analysis computes symbolic transformers for each edge (A, i, B, n) in the control dependency graph (CDG). A is the block that is control-dependant on node n in block B via its ith control-dependency. This transformer captures the effect of the edge and block B on the precondition at A and is defined as follows.

Definition 1 $TF(A, i, B, n)(\phi) := (\phi \wedge condition(n))[phis(A, i)][mem(B)]$

In this definition $condition(n)$ is the formula that enables the jump n. The substitution $[phis(A,i)]$ exchanges each PHI node in block A with its ith dependency in parallel which captures the semantics of the SSA ϕ nodes. The effects of memory-dependent nodes in block B are summarized by the substitution $[mem(B)]$ and is defined in Definition 4.

The precondition ϕ_B at a block B is defined as the least upper bound $\bigvee\{TF(A,i,B,n)(\phi_A) \mid (A,i,B,n) \in CDG\}$ over the predecessors in the CDG.

For the following definitions a function $term(n)$ is needed which computes the term in the logic corresponding to the value computed by the node n. This function is only defined for side-effect free computations such as arithmetic operations. A PROJ DATA node n of a node with side-effect is represented in the term as the variable x_n. These variables are then replaced with terms by the substitution $[mem(B)]$.

Definition 2 $[phis(A,i)] := [term(v)/x_n \mid n = \texttt{PHI} \in A \wedge v = in(n,i)]$

Similarly, PHI nodes are represented using such variables, as their value depends on the particular control-flow predecessor of the containing block. Definition 2 shows how the substitution of the PHI nodes of a block by their ith input $in(n,i)$ is built.

In jFirm there are seven different control-flow operations as summarized in the first part of the table in Definition 3 which shows how the $condition$ that enables its execution is computed. All of these, except GOTO, depend on its *producer* node, their single input dependency. The two branches of an IF depend on the conditional expression c that is the input to the IF and can be either INSTANCEOF or a comparison CMP. For a potentially exception-throwing node p the $condition$ that leads to the exception being thrown is shown in the lower part of the table. An arbitrary number of CASE nodes and a single DEFAULT node may be dependent on a SWITCH node. The input v to the SWITCH determines whether a certain CASE is executed while the DEFAULT case must exclude all other cases.

The last part necessary for the transformer is the Definition 4 of the substitution $[mem(B)]$. The substitution is built by composing substitutions along the chain of memory dependencies which are always $in(m,0)$ of a node m. This process terminates when either the beginning of the block or method is reached. As jFirm currently does not support the representation of different exceptions being thrown, Exc is replaced by a generic instance of type java.lang.RuntimeException. This is another difference to the precondition analysis of Chandra et al.[CFS09].

The $substitute(m)$ method in particular implements the effects of nodes that read from or write to memory. The operation PUTFIELD f for example

Definition 3 *$condition(n)$ represents the formula that enables execution of a node n. It depends on the operation of n and its input dependencies as defined in the following table:*

Node n	Inputs	$condition(n)$
`GOTO`		`true`
`PROJ CONT`	p	$\neg condition(p)$
`PROJ EXC`	p	$condition(p)$
`PROJ TRUE`	`IF`$\rightarrow c$	$condition(c)$
`PROJ FALSE`	`IF`$\rightarrow c$	$\neg condition(c)$
`CASE` c	`SWITCH`$\rightarrow v$	$term(v) = term(c)$
`DEFAULT`	`SWITCH`$\rightarrow v$	$\bigwedge\{term(v) \neq term(c) \mid \texttt{CASE}\ c \rightarrow \texttt{SWITCH} \in E\}$
Node c	Inputs	$condition(c)$ the literal implementing the branch condition c
`INSTANCEOF` t	v	`subType(typeOf(`$term(v)$`),`T_t`)`
`CMP` $relation$	l, r	$term(l)\ relation\ term(r)$
Node p	Inputs	$condition(p)$ for an exception to be thrown
`GETFIELD` f	m, o	$term(o) =$ `null`
`PUTFIELD` f	m, o, v	$term(o) =$ `null`
`ARRAYLENGTH`	m, o	$term(o) =$ `null`
`ARRAYLOAD`	m, o, i	$term(o) =$ `null` $\vee\ i < 0 \vee i \geq$ `length(`$term(o)$`)`
`ARRAYSTORE`	m, o, i, v	$term(o) =$ `null` $\vee\ i < 0 \vee i \geq$ `length(`$term(o)$`)`
`CAST` t	m, o	$term(o) \neq$ `null` $\wedge\ \neg$`subType(typeOf(`$term(v)$`),`T_t`)`
`NEWARRAY` t	m, d	$term(d) < 0$
`DIV`	m, l, r	$term(r) = 0$

modifies the memory array of the field f by substituting previous occurrences with a functional update.

Different to reads or writes from memory the `CAST` and integer `DIV` operations are only memory-dependent because they may throw exceptions.

The precondition of a method m with regard to a postcondition ϕ is computed by a fixed-point iteration which applies the transformers until the precondition ϕ_B stabilizes for every block B. The analysis initializes every block precondition with `false`, except for the `END` block to which the postcondition is assigned.

The transformers are monotonous in that any ϕ_B can only increase during the computation of the fixed-point. However, as the domain is not finite and contains infinite increasing chains of formulae we must apply a widening after a finite number of iterations. This widening assigns `true` to a precondition which is always a fixed-point since it is the largest element in the

Definition 4 *Let (A, i, B, n) be the edge the transformer is built for. Then $[mem(B)] := sub(m)$ where m is the last memory node in block B and $sub(m)$ is defined as follows.*

$$sub(m) := \begin{cases} [\,] & \text{if } m = \texttt{PARAM} \\ & \vee\ m = \texttt{PHI} \\ & \vee\ block(m) \neq B \\ sub(in(m,0)) \circ [\texttt{fresh(}T_{exc}\texttt{)/Exc}] & \text{if } n = \texttt{PROJ EXC} \wedge n \to m \in E \\ sub(in(m,0)) & \text{if } m = \texttt{PROJ MEM} \\ sub(in(m,0)) \circ substitute(m) & \text{else} \end{cases}$$

The function $substitute(m)$ is defined via the following table where x_d is the variable for a node $d =$ `PROJ DATA` with m as its input node. id is an integer not used before to identify an object reference.

Node m	Inputs	$substitute(m)$
`GETFIELD` f	m, o	$[\texttt{read(}f\texttt{,}term(o)\texttt{)}/x_d]$
`PUTFIELD` f	m, o, v	$[\texttt{write(}f, term(o), term(v)\texttt{)}/f]$
`ARRAYLENGTH`	m, o	$[\texttt{length(}term(o)\texttt{)}/x_d]$
`ARRAYLOAD`	m, o, i	$[\texttt{readA(}A_{r/t}, term(o), term(i)\texttt{)}/x_d]$
`ARRAYSTORE`	m, o, i, v	$[\texttt{writeA(}A_{r/t}, term(o), term(i), term(v)\texttt{)}/A_{r/t}]$
`CAST` t	m, o	$[term(o)/x_d]$
`NEWARRAY` t	m, d	$[\texttt{freshA(}id, T_t, term(d)\texttt{)}/x_d]$
`NEWOBJECT` t	m	$[\texttt{fresh(}id, T_t\texttt{)}/x_d]$
`DIV`	m, l, r	$[(term(l)/term(r))/x_d]$
`RETURN`	m, v	$[\texttt{null/Exc}, term(v)/\texttt{Ret}]$
`EXCRETURN`	m, v	$[term(v)/\texttt{Exc}]$

domain.

The precondition of the whole analysed method with respect to the postcondition, is the precondition ϕ_S where the block S contains the `PARAM` nodes of the method.

3.1.3 Interprocedural analysis

For interprocedural analysis we must also handle any `CALL` node n in the graph. This deviates from the above scheme, since we must compute the precondition of a possible callee method m' with regard to the postcondition.

Definition 5 *Let x_d and x_e be the variables for the* `PROJ DATA` *and* `PROJ EXCDATA` *nodes of the call node c respectively. In addition let node o be the receiver object of the call and a_i the parameters to c, in particular $a_0 = o$. The transformer for a control dependency edge (A, i, B, n) with a block B ending with a call c is defined as follows.*

$post(\phi) := \phi[phis(A,i)][\texttt{Ret}/x_d, \texttt{Exc}/x_e]$

$disp := \texttt{dispatch(typeOf(}term(o)\texttt{),}m\texttt{)}$

$[params(m')] := [term(a_i)/v_{p_i} \mid p_i = \texttt{PARAM} \in N_{m'}]$

$pres(\phi) := \bigvee\{pre(m', post(\phi))[params(m')] \wedge m' = disp \mid \text{possible callee } m'\}$

$$TF_{call}(A,i,B,n)(\phi) := \begin{cases} (pres(\phi) \wedge term(o) \neq \texttt{null})[mem(B)] & \text{if } n = \texttt{PROJ CONT} \\ (pres(\phi) \vee term(o) = \texttt{null})[mem(B)] & \text{if } n = \texttt{PROJ EXC} \end{cases}$$

In this definition $post$ translates the postcondition into the space of any called method by substituting the formal special variables `Ret` and `Exc` for their projections in the caller method. The term $disp$ evaluates to the method that the call dispatches to, based on the type of the receiver object o. pre computes the precondition of a method call using the interprocedural analysis while $pres$ captures all possible callees and replaces the formal parameters of the callee by the actual terms in the caller.

3.2 Automaton Inference

Given the definitions to derive preconditions of method calls, we are now able to use these to define automaton inference algorithm that analyses the state space of an abstract object O of a class C. Beginning with the error state labelled with the predicate `ExceptionThrown` $:=$ `Exc` $\neq$ `null` the algorithm repeatedly computes preconditions of method calls and states as postconditions and translates these preconditions into additional states and edges of the automata.

The algorithm requires a set $methods(C)$ which defines the interface of the class that should be considered when constructing the automaton. In most cases this set is defined by the visibility of the methods and includes only those that are visible to any class and thus **`public`**.

Definition 6 *A typestate automata for a class C is a tuple $(S, E, init, E_{init})$. $S \subset F$ is a finite subset of all formulae in the logical domain F and each $s \in S$ is called a state. Each edge $(s, a, t) \in E$ is composed of a source state $s \in S$, target state $t \in S$ and an action $a \in \{p : m \mid p \in F \wedge m \in methods(C)\} \cup \{\varepsilon\}$. $init$ is the initial state of the automaton signifying the state of the object before a call to the constructor. Edges $(init, p : m, t) \in E_{init}$ are transitions from the initial state with a condition $p \in F$, $m \in constructors(C)$ and a target state $t \in S$.*

```
AUTOMATONINFERENCE(C)
    E, E_init = ∅
    S = {ExceptionThrown}
    O = new variable
    AXIOMS(O ≠ null, typeOf(O) = T_C)
 1  workList = [ ]
 2  PREDECESSORS(ExceptionThrown, ExceptionThrown, workList)
 3  while workList ≠ ∅
 4      s = POLL(workList)
 5      PREDECESSORS(s, s ∧ ¬ExceptionThrown, workList)
 6  for m ∈ constructors(C)
 7      for s ∈ S
 8          if s = ExceptionThrown
 9              post = s
10          else post = s ∧ ¬ExceptionThrown
11          precondition = pre(m, post)[O/this]
12          for (φ_s, φ_p) ∈ SPLIT(precondition)
13              if VALID(INITIAL(C, O) ⇒ φ_s) ∧ SATISFIABLE(φ_p)
14                  E = E ∪ (init, φ_p : m, s)
15  EPSILONS(S, E)
16  REMOVEUNREACHABLE(S, E, init, E_init)
17  return (S, E, init, E_init)

PREDECESSORS(target, post, workList)
 1  for m ∈ methods(C)
 2      precondition = pre(m, post)[O/this]
 3      for (φ_s, φ_p) ∈ SPLIT(precondition)
 4          if SATISFIABLE(φ_s) ∧ SATISFIABLE(φ_p)
 5              if φ_s ∉ S
 6                  ADD(workList, φ_s)
 7              S = S ∪ φ_s
 8              E = E ∪ (φ_s, φ_p : m, target)
```

Figure 3.1: Automaton inference algorithm

Figure 3.1 shows the main procedures of the algorithm. First, we initialise the set of states with the error state and create a fresh variable in the logic to represent the abstract object O. Contrary to the function `fresh` of the logic,

the axiom for default field values does not hold for this variable. This is because we use the variable to substitute the **`this`** parameter in each computed precondition and we want to keep the symbolical values of the fields in the precondition to identify the state.

The algorithm begins by computing the predecessors of the error state and adding these to a work list. The PREDECESSORS procedure iterates over all possible calls that could have been made on the object to reach the target state. To identify predecessor states the algorithm makes use of the disjunctive normal form of the precondition and splits it into pairs of formulae (ϕ_s, ϕ_p). ϕ_s is a conjunction which may never refer to x_p variables of the logic where $p =$ `PARAM` is some parameter, while in ϕ_p each conjunct includes a parameter variable. Figure 3.2 illustrates this process. For each such pair, the algorithm adds a state ϕ_s with a transition to the target state, guarded by the parameter condition ϕ_p.

The loop over the work list terminates when no more new predecessors can be found. Termination is only guaranteed for classes where finitely many steps can lead to an exception. The full specification of any other class however, cannot be encoded in a finite state automaton. In particular it would require very coarse over-approximation to encode such a class.

An example of such a case would be a bounded stack that can grow with PUSH operations and shrink with POP operations and would throw an exception when it reaches a certain maximum size or POP is called on an empty stack. A precise automaton specifying these properties would need at least as many states as the maximum size, that size might be unknown however. The operations might thus throw exceptions from any state, as it cannot be determined statically whether the size has been reached.

After finding a fixed point on the set of states in the automaton, the algorithm proceeds to check for each state *s* whether it may be reached by a call to the constructor. This is the case for non-error states when the constructor does not throw an exception and the state related part of the precondition is implied by a formula INITIAL(C, O) describing the initial values of the object's fields.

After adding the constructor transitions, the algorithm adds ε-edges from any state a implying another state b. In particular whenever a method m call may throw an exception independent of the state of the object, for example a `NullPointerException` when a parameter is equal to **`null`**, the automaton includes an edge $(\texttt{true}, \phi_p : m, \texttt{ExceptionThrown})$ and a $true$ state is created. This state is implied by any other state and thus all states have a transition to $true$.

Since there may be states which are unreachable from the $init$ state the algorithm removes these from the automaton in a last step.

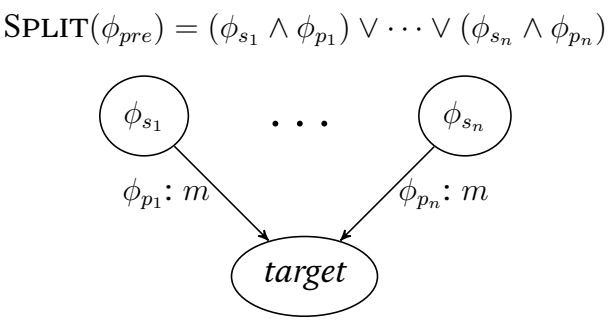

Figure 3.2: Adding predecessor states of a target state to the automaton. The s_i are conjuncts of predicates containing no references to parameters and p_i are conjuncts of predicates each including some reference to a parameter.

3.2.1 Example

Listing 3.1 shows a simplified version of `java.io.PipedOutputStream`. In the documentation for this class many implicit constraints such as "A PipedInputStream must be connected before writing." or "len may not exceed b.length." are left unsaid and require deduction or experimentation by the programmer.

In this example we do not take into account the calls to `sink` for sake of simplicity.

Figure 3.3 shows the computed specification using a shorter Java notation for the field `reads` like `sink` and `snk.connected`. We can see that transitions from the `true` state model method parameter preconditions which, if violated, lead from any state to the error state. Sound approaches which do not encode parameter preconditions lead to very imprecise automata in which any call to *connect* or *write* would lead to an error.

```
1  public class PipedOutputStream {
2
3      private PipedInputStream sink;
4
5      public PipedOutputStream() {}
6
7      public void connect(PipedInputStream snk) {
8          if (snk == null) {
9              throw new NullPointerException();
10         } else if(sink != null || snk.connected) {
11             throw new IOException("Already connected");
12         }
13         sink = snk;
14         snk.connected = true;
15     }
16
17     public void write(byte b[]) {
18         if (sink == null) {
19             throw new IOException("No pipe connected");
20         } else if (b == null) {
21             throw new NullPointerException();
22         }
23         sink.receive(b, len);
24     }
25 }
```

Listing 3.1: Simplified version of java.io.PipedOutputStream

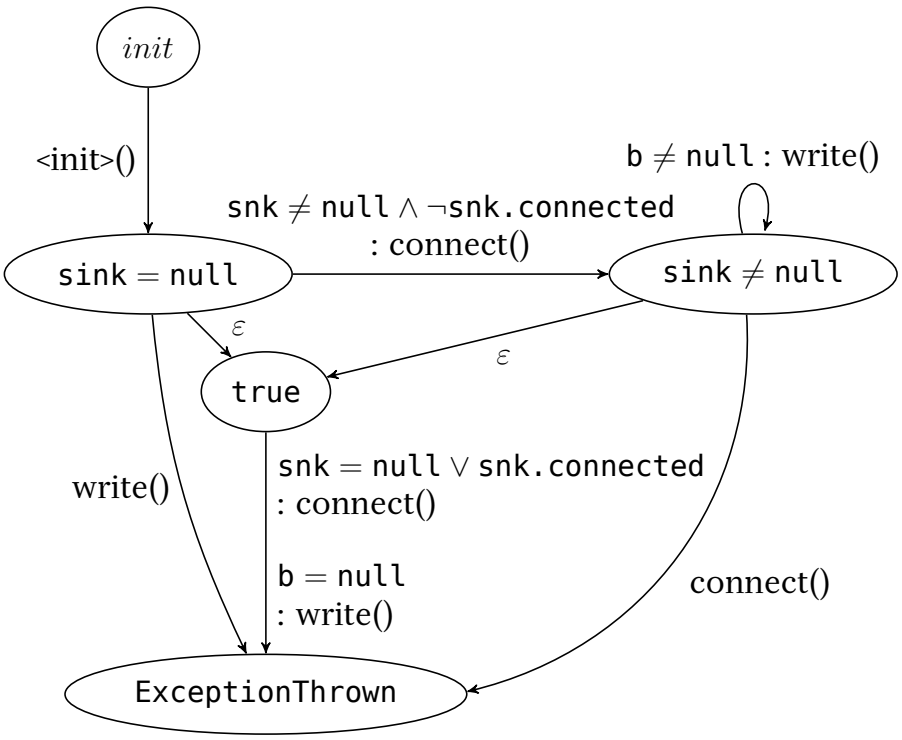

Figure 3.3: Manually generated automaton for the simplified `PipedOutputStream`

4 Evaluation

The evaluation of the algorithms developed in Chapter 3 consists of three parts. The first part is the implementation of the precondition analysis on jFirm and an implementation of the automaton inference algorithm on top of this analysis. In the second part I show that the algorithm can be used to derive meaningful models of complete Java classes from the Java Platform Standard Edition 6. In the last part I implemented a prototype verifier to show that these automata can be used to perform verification of client code.

4.1 Implementation

In the following I describe particular challenges and details of the four parts implemented during this thesis. The first is the logic underlying the static precondition analysis which is the second part. In the third part I use the analysis to implement the inference algorithm that produces the automata used in the last part, the verifier.

4.1.1 Logic

I implemented the logic of the analysis in Java using a caching scheme such that each literal and term is created at most once. While keeping the memory profile low this also allows fast lookup in hash tables and equality comparisons. The `equals` method can then act as a very fast approximation of the logical equality test and benefits from a wide range of fast simplifications done when creating new literals and terms.

In particular I pre-process the jFirm graphs using constant folding and an analysis that simplifies arithmetic terms. Additionally the axioms are implemented as simplifications as they only operate on the local structure of terms and literals or can use information from the internal class hierarchy representation of jFirm.

Much of the logic as described in Section 3.1 can be translated directly to a theorem prover for simplification and validity checking purposes. For

this implementation I chose CVC3 [BT07] which like most provers, is implemented in C/C++ but different to most, comes with a Java API wrapping the Java native interface to the library.

CVC3 does not directly support the syntax for the two-dimensional arrays necessary for the implementation of Java array reads and writes in the logic. These can be easily transformed into two nested reads from the memory array however.

The axioms for `dispatch` and `subType` are instantiated for types and methods on the fly, when they occur during the analysis.

As calls to CVC3 are very expensive I delay all translation to CVC3 expressions to the latest possible point so that most cases of satisfiability and validity checking are handled by quicker implementations of special cases in the Java implementation of the logic.

Eager creation of CVC3 expressions lead to huge memory leakage as conjunctions and disjunctions are created and destroyed often during the analysis, but did not seem to be freed by the Java native interface to CVC3 due to exceptions thrown in object finalizer methods.

4.1.2 Precondition analysis

The definitions of the procedures in Section 3.1.2 are very close to how the analysis builds the transformers, as the procedures are already defined in terms of the jFirm IR. In the case of transformers for blocks containing `CALL` nodes there are some special cases. When the target of a call is a native method, I assume that this call may return an `unknown` value of appropriate type, if any, and does not modify the postcondition. I also introduce the option to unsoundly skip calls to some methods with the same assumptions for the purpose of evaluation.

Results of the precondition analysis are cached for evaluations of $pre(m, \phi)$ and reused where possible. The implementation currently does not make use of generalization[CFS09] to increase the possibilities of reuse and reduce the size of analysed postconditions.

The interprocedural precondition analysis depends on a call graph to resolve possible targets of method calls. I compute the call graph using a class hierarchy analysis where a set of methods is given as the possible entry points to the program or library. In case of the automaton inference algorithm this is the set of interface methods that define the possible transitions in the automaton.

The iteration strategy is an implementation of the recursive algorithm from [WS10]. This algorithm recursively searches for the precondition at a block

without recomputing preconditions of blocks already visited during the descent. When a predecessor of block is updated during the descent, that block's precondition is immediately recomputed.

4.1.3 Automaton Inference

The main challenge when implementing the algorithm from Section 3.2 was scalability of the satisfiability and validity checks. This is because in general a finitely abstractable class C may have a number m of methods and a number p of predicates P relevant for the automaton. The algorithm may find any conjunction of a subset of P as a state when exploring the state space from the error state. For each of these, exponentially many, states, m preconditions and $2m$ satisfiability queries are computed.

The most expensive computations are the satisfiability queries to CVC3. In the implementation I approximate these using simplification rules for conjunctions. In particular a conjunction containing a literal and its negation can never be satisfied. I also simplify arithmetic operations when a literal equates a term to a constant. This enables fast computation of the automaton even in the case of a large number of states.

The implementation also disregards methods when computing predecessor state when the analysis deduces that any call to the method leads to error state. This is the case if it discovers an edge from `true` to the error state with parameter condition `true`.

4.1.4 Client Verification

Verifying a client against an automaton computed by this approach is very similar to typestate verification [FYD+08]. The automaton however includes additional information about method preconditions and predicates describing abstract states compared to a typestate automaton. A precise verifier must gain knowledge from the client's usage of the component to fully exploit the specification.

The verification algorithm tracks *abstract objects* in the code of the client. These abstract objects may correspond to multiple objects at runtime as a potentially infinite number might be allocated within loops or recursive methods. I use the common abstraction that tracks one abstract object per allocation site.

The verifier is an abstract interpretation [CC77] of the client code where the abstract domain maps abstract objects to sets of automaton states. At runtime an object may be in any state its corresponding abstract object is

mapped to. If these states include the error state at some program point, the verifier has found a possible violation of the inferred class specification.

Each call statement in the client's code may transform the set of states an abstract object is mapped to. For a method call on an abstract object its set of states afterwards are the ε-successors of the incoming states with respect to the called method. With my approach the verifier can disregard transitions with an unsatisfiable parameter condition.

To check whether a parameter condition cannot be satisfied the verifier can employ additional static analyses such as interval analysis [CC77] or points-to analysis among others or enrich the abstract domain of the verifier such that it tracks primitive and reference values. The verifier is still sound however, if it over-approximates and declares preconditions to be satisfiable which can never be satisfied at runtime.

My implementation of a prototype verifier combines the typestate abstract domain with an interval domain for integer typed fields and a simple partially flow-sensitive and context-insensitive points-to analysis.

4.2 Results

For the following results I chose to unsoundly ignore calls to methods outside the analysed class. In many cases this is an approximation similar to previous work [BHS07, ACMN05, HJM05] where only a few predicates over the fields or local variables of the class were used to model the state of the class and exceptions within such calls were ignored. In cases where such class-local predicates may not be modified by the call, it is safely approximated by skipping it and assuming an `unknown` return value.

In general this unsound approximation disables the use of the generated automata for verifying the absence of errors with regard to the use of the class. To my knowledge no previous approach to automaton inference has achieved this. A full evaluation of the suitability of the automata for verification is beyond the scope of this thesis and the capabilities of the prototype verifier however.

For program understanding and ease of readability, it is most often not desirable to add predicates over the internal state of a field to a state of the automaton. In the case of `java.io.PipedOutputStream` analysing the calls to the field `sink` of type `java.io.PipedInputStream` for exceptions disables the representation as a finite state automaton. This is because it manages an arbitrary sized buffer array and indices to elements of the buffer which may signal exceptions.

Figure 4.1 shows the automaton computed by the inference algorithm on the original Java bytecode of `java.io.PipedOutputStream`. In essence it is the same as the simplified version from the example in Section 3.2.1, however it displays the full complexity of the parameter conditions.

In addition it shows that the approach captures the effects of the call to `connect` from within the constructor at the bottom left. It directly transitions to the state `Connected` $:= \neg$`(read(this, sink)` $=$ `null)` and models the effect of successive calls to `<init>()` and `connect()`.

However it also shows the limits of analysing only the `PipedOutputStream` class as the call to `close()` in the state `Connected` stays in that state. This calls only modifies the internal state of the `sink` field, but disallows any further calls to `sink.receive()` and thus calls to any `write()` method on the `PipedOutputStream` as they would then throw exceptions. The automaton would need at least one additional state to model this property, because the `sink` field would still be initialised and thus `connect()` would still be an erroneous call.

Alur et al. [ACMN05] also miss this fact in their automaton of `PipedOutputStream`. Additionally miss the property that additional calls to `connect()` are not possible after the first. This is due to their rewrite of the class that removes any other exception other than the one they are interested in and replaces it by adding a boolean return value to a method, signalling success or failure.

Beyer et al. [BHS07] find the property that a call to `close()` disables calls to `write()`. However, their automaton allows a reconnect after the `close()` which in fact would always throw an exception. This seems to be an artefact of their manual abstraction of the source code.

Figure 4.2 shows the automaton generated for `java.security.Signature`. This class provides a common interface to several digital signature algorithms. The automaton shows that before signing or verifying data, it must be initialized with calls to `initSign()` or `initVerify()` calls respectively. It is possible to switch between these modes at any time. Alur et al. [ACMN05] find the same automaton, except for the addition of the parameter conditions.

Both automata are generated by the analysis in a few seconds and with memory usage of a hundred megabytes on average.

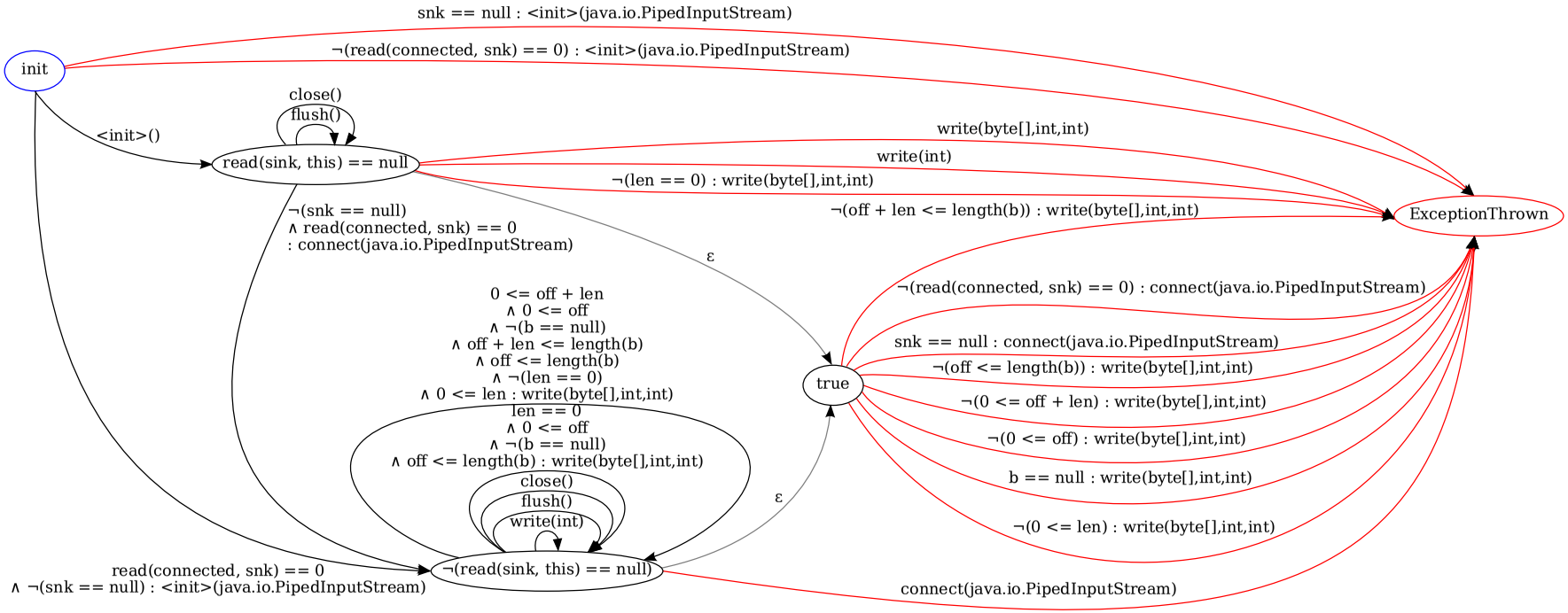

Figure 4.1: The automata inferred for `java.io.PipedOutputStream`

4.2.1 Client verification

To check whether the prototype verifier is able to find the properties encoded in the automaton I ran it on a set of small tests adapted from the OpenJDK test suite [JDK] for `java.io.PipedOutputStream`. The verifier checks these tests against the automaton from Figure 4.1 and warns upon a potential violation.

Listing 4.1 shows a test that writes to a `PipedOutputStream` which has not been connected. This property is captured by the automaton and the verifier correctly detects it and reports that a call to `write()` from the state `(read(this, sink)` $=$ `null)` will lead to the error state.

Listing 4.2 shows code that connects twice to the same `PipedInputStream` `is` from different `PipedOutputStream` objects. The verifier is able to track the field values of `is` and reports that these imply the parameter condition $\neg$`(read(connected, snk)` $= 0$`)` of the transition that leads to the error.

The test in Listing 4.3 checks the property that forbids `write()` calls after a call to `close()`. As this is property is missing from the automaton in an unsound way such that `close()` calls do not change the state, the verifier does not generate an error.

```
1  import java.io.*;
2
3  public class NotConnected {
4      public static void main( String[] argv ) throws Exception {
5          PipedInputStream i = new PipedInputStream();
6          PipedOutputStream o = new PipedOutputStream();
7
8          try {
9              o.write(10);
10             throw new Exception("Test failed");
11         } catch (IOException e) {
12         }
13     }
14 }
```

Listing 4.1: jdk/test/java/io/PipedOutputStream/NotConnected.java

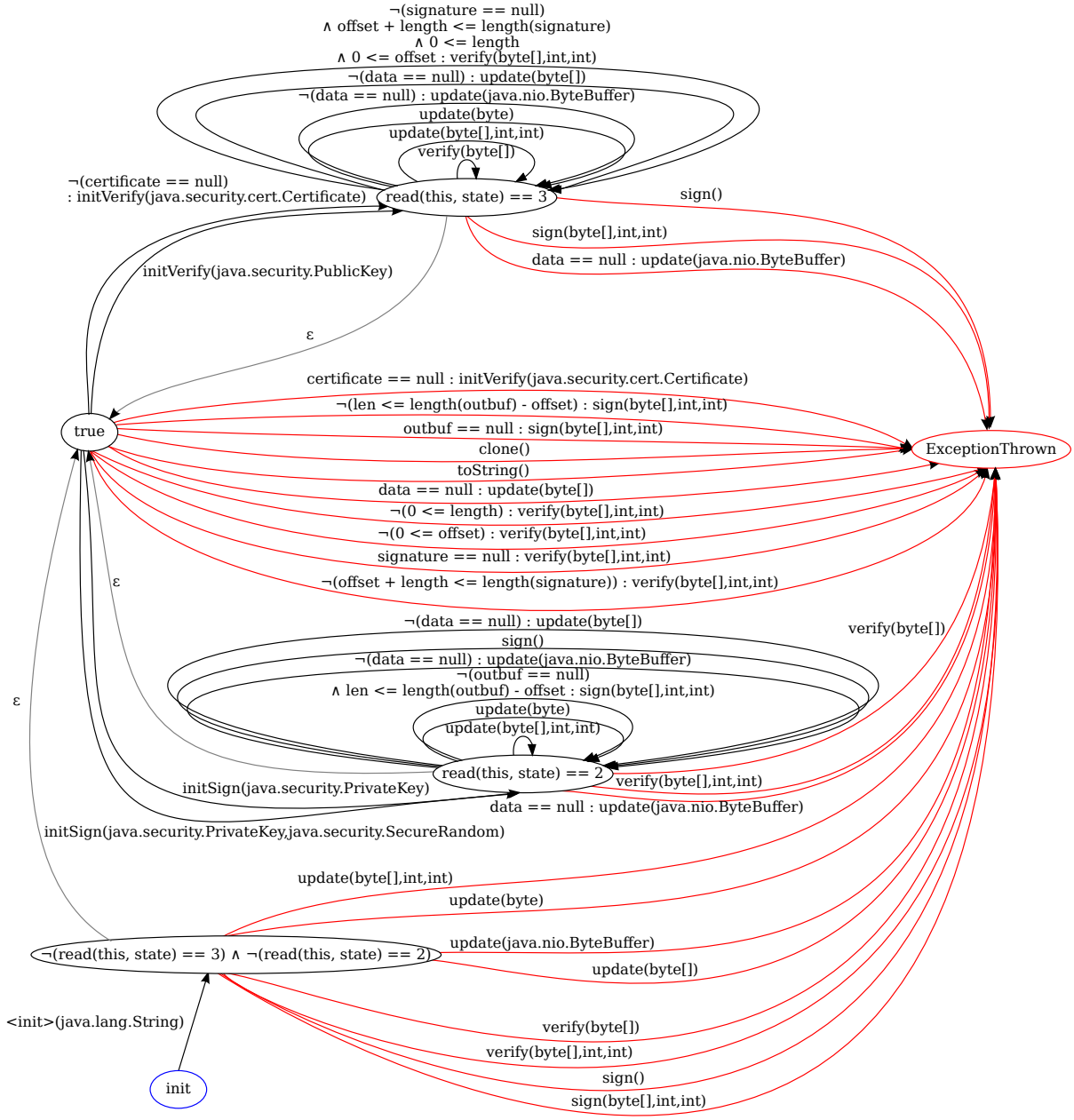

Figure 4.2: The automata inferred for `java.security.Signature`. Due to space constraints, some get*/set* methods have been removed.

```
1  import java.io.*;
2
3  public class MultipleConnect {
4
5      public static void main(String[] argv) throws Exception {
6          PipedOutputStream os = new PipedOutputStream();
7          PipedOutputStream os2 = new PipedOutputStream();
8          PipedInputStream is = new PipedInputStream();
9          os.connect(is);
10         try {
11             os2.connect(is);
12             throw new Exception("Test failed");
13         } catch(IOException e) {
14         }
15     }
16 }
```

Listing 4.2: jdk/test/java/io/PipedOutputStream/MultipleConnect.java

```
1  import java.io.*;
2
3  public class ClosedWrite {
4
5      public static void main(String[] argv) throws Exception {
6          PipedOutputStream os = new PipedOutputStream();
7          PipedInputStream is = new PipedInputStream();
8          os.connect(is);
9          os.close();
10         try {
11             os.write(10);
12             throw new
13                 RuntimeException("Test failed");
14         } catch(IOException e) {
15         }
16     }
17 }
```

Listing 4.3: jdk/test/java/io/PipedOutputStream/ClosedWrite.java

5 Conclusion

In this thesis I provide a new approach to the problem of creating modular specifications as finite state automata. These are useful for programmer understanding when learning about libraries and can enhance the documentation of software components by providing an intuitive view on erroneous usage. In addition, these automata can be used to detect defects in code using the component and are a further step towards automatic modular verification.

My approach makes use of an over-approximate precondition analysis of method calls. The analysis is implemented on top of an expressive logical theory modelling nearly all features of the Java language besides multithreading and reflections. The analysis is fully automatic and makes use of the CVC3 theorem prover to handle validity queries that it cannot solve using an array of common simplifications.

The algorithm that infers the automata assumes that the assertions in the analysed class signify the error state. From the repeated application of the precondition analysis to the discovered states it discovers transitions from additional states by splitting the preconditions of method calls into predicates on the fields of the class and conditions on the parameters to the method call. This generates sequences of calls with parameter conditions that may lead to the error state. In a final step the automaton is provided an initial state by analysing the preconditions of constructor calls.

I implemented this approach in Java on the jFirm intermediate representation and show that this approach yields automata modelling usage properties of Java classes. I show that the generated automata represent the valid properties discovered by previous approaches [BHS07, ACMN05] while increasing the expressiveness with the parameter conditions.

To demonstrate the ability of a verifier to make use of these automata, I implemented a prototype that checks test cases against a computed automaton. I showed that the verifier can detect violations of the properties modelled by the automaton.

5.1 Future work

A limitation faced by my approach and all other specification inference algorithms is unknown code. Components can include callbacks to client code or possibly operate on subclasses provided by the client. In Java especially *toString*, *compareTo* and *hashCode* are often overwritten methods accessed by library classes. In future work I could extend the approach to encode assumptions about the client code in such a way that a verifier can independently check whether they hold.

An inherent problem is the choice of finite state automata to abstract software components. These can only represent regular properties and thus cannot express more complex properties. An applicable example would be a stack, where a number of stack push operation allows an equal number of pop operations. In future work I would like to explore whether more expressive classes of automata or formal languages could be used to specify such properties and whether these can be inferred automatically.

Another challenge is the inference of loop invariants and variants using widenings. An idea to explore in future work would be to use the syntactical evolution of formulae in the precondition transformers to introduce quantifiers that can represent these properties in a finite approximation.

Bibliography

[ACMN05] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *POPL '05: Proceedings of the 32th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '05, pages 98–109, New York, NY, USA, 2005. ACM.

[BBZ11] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. Firm—a graph-based intermediate representation. In *CGO '11: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, 2011.

[BHS07] Dirk Beyer, Thomas Henzinger, and Vasu Singh. Algorithms for Interface Synthesis. In Werner Damm and Holger Hermanns, editors, *CAV'07 Tutorial: Lecture Notes in Computer Science 4950: Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 4–19. Springer Berlin / Heidelberg, 2007.

[BKW09] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *Proceedings of FM-CAD 2009*, pages 69–76. IEEE, 2009.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CCL11] P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In D. Schmidt R. Jhala, editor, *Proceedings of the Twelveth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2011)*, pages 150–168, Austin, Texas, January 23–25 2011. LNCS 6538, Springer, Heidelberg.

[CCM10] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. Logical Abstract Domains and Interpretations. In S. Nanz, editor, *The Future of Software Engineering*, pages 48–71. Springer-Verlag, Heidelberg, 2010.

[CFS09] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 363–374, New York, NY, USA, 2009. ACM.

[Dij68] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8:174–186, 1968.

[DKM+10] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM.

[FYD+08] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17:9:1–9:34, May 2008.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[Gri87] David Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

[HJM05] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *ESEC-FSE'05: Proceedings of the 10th European software engineering conference held jointly with 13th*

ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-13, pages 31–40, New York, NY, USA, 2005. ACM.

[JDK] OpenJDK 6 b24 Source Archive Nov 14, 2011. `http://download.java.net/openjdk/jdk6/`.

[LASRG07] T. Lev-Ami, M. Sagiv, T. Reps, and S. Gulwani. Backward analysis for inferring quantified preconditions. Technical report, Tel Aviv University, 2007.

[NGC05] Mangala Gowri Nanda, Christian Grothoff, and Satish Chandra. Deriving object typestates in the presence of inter-object references. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 77–96, New York, NY, USA, 2005. ACM.

[RGJ07] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 123–134, New York, NY, USA, 2007. ACM.

[SDBL01] Aaron Stump, David L. Dill, Clark W. Barrett, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the 16th IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 29–37. IEEE Computer Society, June 2001. Boston, Massachusetts.

[SYFP07] Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 174–184, New York, NY, USA, 2007. ACM.

[WML02] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 218–228, New York, NY, USA, 2002. ACM.

[WS10] Reinhard Wilhelm and Helmut Seidl. *Übersetzerbau 3: Analyse und Transformation*. Springer, 2010.