Commutativity Analysis in ABS

Sondre Skaflem Lunde

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture

60 credits

Department of Informatics

Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Commutativity Analysis in ABS

Sondre Skaflem Lunde

Commutativity Analysis in ABS

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

When analysing programs statically, one quickly runs into issues of too many states and too many transitions, which combine to make the state-space very large. Commutativity analysis is one strategy for reducing the state-space, by identifying pairs of transitions that result in the same state regardless of execution order. This thesis presents an investigation into the suitability of SMT solvers for determining the commutativity of methods in an active object language.

ABS, as an active object language, has cooperative scheduling and strong encapsulation, which makes it possible to work with methods as atomic units. Analysis of these methods should be able to identify commuting methods and reduce the state space.

Satisfiability Modulo Theories (SMT) solvers allow programmers to declare problems rather than constructing solutions themselves. Modern solvers are highly optimized for speed, and able to solve very complex problems. This makes their use very compelling as part of larger analyses.

A new addition to the ABS compiler has been written to output a type checked syntax tree as JSON. A program has been developed in Haskell to take JSON from the ABS compiler and generate code for an SMT solver, and these all combine to determine whether methods within a class commute.

We demonstrate how to leverage an SMT solver to find commuting methods in ABS, and conclude it to be a promising tool for this kind of analysis.

Acknowledgements

I would like to thank Lars Tveito for supervising my work on this thesis. His genuine interest in the project, attention to detail, understanding of the field, and encouragement to follow fun detours wherever they may lead has had a huge impact on the direction and final result of this thesis.

I am grateful to Einar Broch Johnsen for the feedback on my thesis in the final weeks.

Finally, I am always grateful for my family. The unconditional encouragement and support of my parents, who are always only a phone call away, are a constant source of happiness and joy.

Sondre Skaflem Lunde
Oslo, May 2021

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis presents a tool for determining whether two methods commute in ABS, an active object language. Two methods commute if they result in the same state, independent of the order in which they are executed. The key features of ABS that make the analysis possible are a distributed, actor based semantics, and cooperative scheduling. Investigating whether two methods commute is interesting for its potential as part of larger analyses, as well as being a sufficiently interesting problem for examining the appropriateness of SMT solvers for analysing active object languages.

The results from this analysis is a set of pairs of methods within a class that commutes, that can be used by a larger analysis which focuses on the possible execution paths of a program. The shear number of such paths make exhaustive analyses unfeasible for larger programs, and any reduction in the number of paths requiring analysis can drastically lower the time required to test a given program. With access to commuting methods, it is possible to reduce multiple paths down to a single path if the only difference between the paths are the ordering of two such commuting methods.

To determine if two methods commute, we must be able to determine if the contents of those methods break commutativity. In imperative programs, those methods contain *statements*, some of which are independent of state and some of which are not. It is not sufficient to look at individual statements, or even individual methods, as one method might commute with one method and not commute with another. To determine commutativity it is necessary to look at the cumulative effect of two methods on the state of the class, and consider how they affect each other as they are executed.

In an actor-based system, every method can be executed at many different points in the overall run of the program, and can in the worst case increase the number of paths from $n!$ to $(n+1)!$. Any reduction in this case will

have enormous implications for the number of paths that must be tested.

In this analysis we will utilize an SMT solver, which is a highly researched and optimized class of programs that takes as input a series of constraints and resolves whether it is possible to satisfy those constraints. SMT solvers are an extension of more traditional SAT solvers, with a much more expressive input language that allows the user to formalize constraints in what is essentially the same language we are used to from arithmetic, logic, and programming. These solvers allow their users to leverage known axioms in common theories such as arithmetic, logic, and more.

As programmers write a program, it is first constructed in their head before being typed into the computer. As part of this process, knowledge about various aspects of the program is held in the mind of the programmer, and used to leverage the tools in the language. Perhaps an integer is always within a certain bound, or the ordering of elements is irrelevant when combining them from a collection into a single element. Often, these elements remain implicit as the programmer uses them to solve the problem, or they might be made explicit through comments, variable naming, or types.

Many of the advancements in modern compilers are examples of such implicit knowledge made explicit *to the compiler*, not just to other programmers. By making them explicit to the compiler, they can be used to optimize the code in various useful ways. Type systems force the possible inputs and outputs of procedures to be enumerated, for error handling and effects to be reasoned about, and aids in the creation of correct code.

This analysis is interested in a particular aspect of the code that is not common for programmers to consider when writing normal functions, namely commutativity. For programmers, commutativity is most commonly used when dealing with arithmetic and boolean operations. However, this analysis is not so much about making programmer assumptions explicit, but instead tries to formalize (possibly) accidental commutativity for a compiler or other static analysis tool to use as a part of its own analysis.

In addition to looking at the suitability of SMT solvers for this particular problem, a more naive approach, read-write sets, is tested to provide a point of comparison. This approach looks only at operations directly relating to the *state* of the program, and any reading of or writing to the state is considered. This analysis is naive because it does not take into account *the kind* of manipulation that is done when reading or writing, or the order in which they occur. The benefit of this naive approach is guaranteed soundness, and how much easier it is to implement.

1.1 Problem Statement

The main objective of this thesis is the following.

> *Investigate the suitability of SMT solving to determine the commutativity of methods in an active object language, such as ABS.*

Furthermore, such a tool must:

1. *Maintain soundness.* Only methods that are definitely commuting can be determined to be so. Since this tool can be used in a tool chain for static analysis, any unsoundness would spread into other analyses. To maintain soundness, completeness will most likely suffer.
2. *Be comparable to a more naive analysis.* A read-write analysis will provide a baseline to compare the results of the SMT analysis against. While the main target is testing the suitability of SMT solving in general, its success will depend on how *good* it is. The suitability of the SMT based analysis will depend on how easy implementation is, how reusable it is, or how portable it is to other similar problems one might want to use that kind of analysis for.

Modern SMT solvers provide a way to utilize highly researched programs to solve complex problems, as long as they can be encoded into the solver. With modern libraries for different languages, this is easier than ever before, and the potential of SMT solvers is one worthy of serious study.

1.2 Motivation

We want to test the suitability of SMT solving because we are interested in its potential, working on an active object language, to examine if interference between methods is sufficient to determine commutativity.

Partial-Order Reduction

In a program graph, each state can be reached in many different ways because of various possible interleavings. It is the case that certain sequences of states can exist in which the order of individual states are different, but the final state is the same, i.e. two different executions of the program that cause the same result. Since they cause the same result in the end, it should suffice to only consider one of the executions. The purpose of partial-order reduction is thus to identify the paths that are equivalent for the purposes of our analysis, and only analyze one path as a representative for all those paths.

One key insight is that even though there are many different interleavings, many different paths end up in *exactly* the same place [12]. There are some important caveats for two paths to be considered identical, as there are many ways in which one method can influence the execution of another method. If one method running before another causes the second do something different that if the first method never executed, there is *interference*. One possible way to find such cases is to look at which variables are written to in the first method, and whether those variables are read in the second. However, if swapping the two methods cause the method now running second to change its behaviour in exactly the same way, it would leave the state following the execution of the two methods the same regardless of the order of execution. If it can be shown that the state following two methods is the same regardless of the ordering of the methods, then it is sufficient to consider only a single run in any analysis interested in the possible states reached at the end of a program.

Using an SMT solver

SMT solver are very powerful logic engines, that have been developed for years by major companies in industry. By formalizing your problem as logical formula, you can utilize the power of these solvers to perform difficult computations. They allow you, in some sense, to outsource the tougher aspects of your problem to a tool that has years of research put into it to make it as effective as possible.

As a user of these engines, their declarative nature provide a way to formalize problems without having to specify a solution. Instead, you specify the inputs and the restrictions on those inputs, telling the solver what *must* be true and what it depends on, and the solver will try to find a satisfying model. It is harder to write a correct solution than it is declaring the problem. The high-level language used to input problems into the solvers make declaring problems even easier, and when combined with libraries for mainstream languages it is feasible to generate these input programs. It is not trivial by any measure, but provides an alternative to traditional methods that make it well-worth exploring.

SMT-Lib [7] is a programming language designed to act as a common language for SMT solvers.

The ABS concurrency model

ABS is an executable modeling language with certain novel features that provide a foundation upon which to do interesting static analysis.

Firstly, its concurrent object model provides strong encapsulation, in the sense that an object cannot access or modify the state of another object. Only methods defined on each object can directly manipulate the fields of that object, and other objects are restricted to calling these methods if they want to interact. In a sense, all class fields are private, and must be accessed by methods in the object. Secondly, calling methods on objects does not synchronously execute these methods, but places them in a queue, and the internal scheduler of the object decide which waiting call will be executed next. Finally, each methods decides when to yield control. This is called cooperative scheduling.

The consequences of these points for static analysis is that each object can be analysed independently of other objects, vastly reducing the number of analyses needed for the particular investigation this thesis will present. Rather than analyzing every pair of methods in the *module*, we can analyse every pair of methods in every class, for every class in the module.

Commuting Methods

Because of cooperative scheduling and the active object model, it is possible to determine statically if a method in ABS will run to the end or suspend execution at some point inside the method body. As a result, we can analyse ABS at the level of methods, to determine if they commute, i.e. if the state of a class is the same regardless of the ordering of two methods, after the two methods have been executed.

1.3 Contributions

The main contribution is a prototype of a commutativity analysis for ABS, using an SMT solver to search for commuting methods. The development of and results from this tool is used to examine the suitability of using an SMT solver for static analyses of properties of programming languages.

As part of the process there was created:

- Support for outputting the syntax tree of an ABS program as JSON was implemented on the ABS compiler itself. This is a large compiler with multiple backends, and with over two hundred thousand lines of code in the project as a whole.
- An analysis tool written in Haskell takes ABS programs, serialised into JSON, as input. It produces SMT-Lib code that is passed to an SMT solver for evaluation. This analysis will for each pair of

methods determine if they commute or not. Additionally, a read-write analysis is done as part of this execution to provide a point of comparison.

These tools let us contribute to further analyses that want to exclude redundant paths of execution from more general static analyses.

1.4 Chapter Overview

Chapter 2 is an overview of the background theories and tools used to create this analysis.

Chapter 3 is an introduction to Abstract Behavioral Specification (ABS), the language on which this analysis is done.

Chapter 4 covers the translation from ABS to JSON to Haskell data types, which includes the tool serializing ABS into JSON as part of the ABS compiler.

Chapter 5 is a theoretical discussion on what commutativity *is*, in general and in the context of methods in a programming language. This chapter includes a discussion on how different programming statements commute, which is then used when implementing constraints in chapter 6.

Chapter 6 presents the implementation of the tool itself, generating code for the SMT solver from ABS programs. This includes how the state is handled, and how specific language constructs are translated into constraints.

Chapter 7 is an evaluation of the results, a comparison of using the solver and using a read-write analysis, and some concluding thoughts.

1.5 Source code

The source code is available on Github.[1]

[1] www.github.com/sondresl/master

Chapter 2

Background

This chapter will cover the concepts that one should be familiar with in order to fully understand the rest of this thesis. As such, there will be an overview of the relevant static analysis and symbolic execution, before discussing implementation-specific topics like Haskell and SMT solving.

2.1 Soundness and Completeness

The terms soundness and completeness originate from logic. Informally, a system is *sound* if everything it can prove is in fact true. Conversely, a system is *complete* if everything that is true in the system, can be derived by the system.

In the case of the analysis presented in this thesis, it is sound if everything determined to commute actually commutes, and it is complete if everything that commutes is determined to commute. At times when constraints on the implementation forces a choice, we will always favour soundness over completeness.

2.2 Static Analysis

Static analysis has came about as a complement to dynamic analysis of programs, as a way to improve software quality [19]. Dynamic analysis involves running the program and comparing the actual outputs with expected outputs. Static analysis attempts to gain insights about a program from its source code or some representation of the source code.

The most common form of dynamic analysis it testing. By testing the program it is possible to see how the program responds to a wide variety

of expected inputs, and thereby verify that the program works as intended in those specific circumstances. The downside to testing is that the amount of possible inputs to a program is so large that verifying every combination is most often infeasible or impossible. Instead, tools and metrics exist that attempt to ensure a sufficiently wide *variety* of inputs to have every branch of execution tested at least once. One way to judge the quality of dynamic analysis is measuring the number of branches of execution that are actually executed during the run of at least one test. This is known as *branch coverage*.

Early ideas around static analysis emerged as a response to the increased complexity of programs once concurrency was introduced [4], and a wish to *prove* the correctness of such programs [16]. Static analysis instead aims to confirm some aspect of the program that must be true regardless of input. Canonical examples of such analysis are static type checking, verifying that all variables are instantiated before their first use, or ensuring there is no dead code in the program. Since the purpose of the analysis is to prove some property of the program for all states the program can reach, it is focused on staying sound and often suffers in completeness.

A common use of static analysis is inside compilers. Every time a program is compiled, static analysis is performed to ensure the program adheres to certain specifications, and to make optimizations that make the generated code faster. Compiling the source code of a programming language means transforming it between different representations, often multiple times and sometimes with very minor changes. The compiler provides several different representations of the source code throughout this process, as the different versions have different attributes, and as the process progresses elements are either discarded or created depending on the needs of the process at that particular time. Any one of these representations can be used for static analysis, and picking one depends on the characteristics one wishes to analyse.

The most common representation of a program is the Abstract Syntax Tree (AST). As a tree, the program has been transformed by the compiler from pure text into a representation of *multiple dimensions*. This is to say that various aspects of the program have been grouped together in a way that might not be clear from the textual representation of the program, and which now can be treated as separate entities. Rather than thinking about the program in terms of lines and columns, it is now functions, classes, declarations, or any other construct that exists within the programming language in question. This makes it possible to talk about scopes and states, to ask high level questions about lifetimes and availability of names in the program, and find properties that generalize to all programs which share the same basic structural elements.

Certain forms of static analysis are so ubiquitous that they are hardly considered static analysis, like type checking in statically typed languages. Indeed, every instance of analyzing the representation inside the compiler is static analysis of some sort. Type checking is a success story of static analysis, and most interesting problems in the field today would take as a starting point a correctly typed program. Modern static analysis is focused on proving more advanced properties of programs, or finding errors.

Static analysis seeks to prove such properties of the program through mathematical principles, which take the form of logic. The most common issues of dynamic testing are the impossibility of testing programs for every possible combination of inputs. It would simply take too much time to do so. The most common classes of properties are liveness properties and safety properties [2, 16]. A liveness property states that *something good should eventually occur*, such as exiting a loop or calling a specific function. A safety property states that *nothing bad ever happens*, with examples being never returning a null pointer, or a certain invariant never being broken.

If the most pressing problem for dynamic analysis is the enormous amount of possible values to begin a program with, the problem in static analysis is the vast amount of states that a program can reach. This problem is known as the *state-space explosion*, and occurs when the number of available operations in a given state is high, and the ordering of these operations are not fixed and can change between runs of the program. If the program is represented by a control-flow graph, in which every state the program is a *node* and every possible state transition is an edge, the interleaving of states means that the number of edges becomes very high. As the number of nodes and edges increases, the number of possible paths through the graph explodes until it becomes unfeasible to consider them all [12].

To make static analysis possible, the number of states necessary to check must be reduced, or analyses themselves must be confined to sufficiently small subsets of the state space to not run into the problem in the first place.

2.3 Symbolic Execution

One reason to avoid dynamic testing of programs is the huge number of runs that would have to be tested, since each run only provides information about *that particular run*. Symbolic execution provide a way to explore multiple executions *simultaneously*, by avoiding concrete input and instead representing variables as symbols and having a solver determine after the fact various properties of the runs [5].

In symbolic execution, if a variable is ever assigned as a constant, it will be treated as a constant. However, if is assigned some dynamic value, such as an input from outside the program, it is represented completely free. In a statically typed language, the type will be known at compile time, and so no variable will be *completely* free. If there is an operation that either reassigns this variable, or uses it to create a new one, then this operation will use the current constraints on the variable when generating the result of applying the operation. At the end of the symbolic execution, each variable will be constrained in various ways, or possibly not at all, or be a concrete value. Then, with all the variables in with their respective constraints, it is possible to *ask* the solver if the constraints can be satisfied. If the constraints have been constructed correctly, a satisfying model should mean that the property holds.

2.4 Boolean Satisfiability (SAT)

Boolean satisfiability (SAT) is an old problem in computer science, and the first to be proved NP-complete in 1971 [10]. The problem is to prove the satisfiability of a propositional logic formula, which is a logic formula consisting of boolean variables and logical connectives, e.g. conjunction, disjunction, and negation.

A common approach is to write the formulas on *Conjunctive Normal Form (CNF)* which means that it consists of *clauses*, every one of which must be true for the entire formula to be satisfied. The following is an example of a formula in CNF:

$$(a \vee b \vee c) \wedge (\neg a \vee c) \wedge b$$

As an NP-complete problem there is no known algorithm to determine the satisfiability of a boolean formula in polynomial time.

SAT has become the foundation for *SAT solvers*, which use heuristics and techniques for reducing the complexity of SAT formulas in such an effective way that modern solvers can deal with thousands of clauses and millions of variables [9].

2.5 SMT

Satisfiability Modulo Theories (SMT) generalizes boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. [17]

Extending the input language from propositional logic to first-order logic allows for more expressiveness, and the ability to formalize equations in a language very close (if not identical) to the original formulation of those equations [6]. However, given that SAT is NP-complete and that SMT is built on top of SAT, the formulas constructed using an SMT-solver are also inherently difficult to solve.

The advantages of SMT over normal SAT are most visible in the ways it can better formalize advanced queries, and how the built-in theories have encoded the axioms in such a way that they are exploited by the solver to improve performance, and thus provide answers for more complex formulas.

The addition of *types* is a huge ergonomic improvement. Now, instead of laboriously specifying the properties of numbers, it is possible to state that a certain variable is an integer of a certain size and the solver will know its possible values, and its axioms. In logic, types are known as *sorts*, and the support for multiple sorts is known as many-sorted logic.

To formalize a problem as a propositional logical formula is tedious, difficult and error-prone, and places much weight on the translator. The conversion from first-order logic to propositional logic can lead to an explosion in terms of size, and is a natural job for a computer. Translating the entire equation set into conjunctive normal form, and passing that formula to the SAT-solver is called the *eager approach*. This can be useful for smaller formulas, and makes use of highly optimized SAT solver that form the backbone of SMT solvers.

The alternative is the *lazy approach*, and involves solving as few equations as possible at one time, leveraging solvers with knowledge of specific theories. These additional solvers are what separates SMT from normal SAT solving. A specific theory solver functions by fixing the expected interpretation of certain symbols, and the two theories that provide the most obvious benefit over simple propositional logic are the theories of arithmetic and uninterpreted functions.

Another important benefit of the lazy approach is that some of the constraints might be determined either completely, or restricted sufficiently, to the point where a fewer possibilities have to be explored. This is done by heuristics for finding shortcuts during execution being programmed into the solver.

2.5.1 SMT-Lib

The default input format is `SMT-Lib`, a language developed specifically to be a common input language for SMT solvers [7]. The input language

consists of S-expressions, with parentheses and prefix notation.

Listing 2.1: SMT Example Program 1

```
(declare-const b0 Bool)

(define-fun b1 () Bool (or b0 (not b0)))

(assert b1)
(check-sat)
(get-model)
```

Listing 2.1 shows a very simple program sent to the solver.

1. A constant is defined, with the sort `Bool`. A constant is syntactic sugar for a function with no arguments and a fixed value at creation. It is constant once a value has been assigned as part of the solver execution, but can be initialized with different values in different executions.
2. A new variable is created, with the correct sort, and its value depends on the value of the previously created constant.
3. The value of this last variable is asserted (i.e. it must be true for the model to be satisfiable).
4. The satisfiability of the model is checked. In this case there is a tautology, and a model is always found regardless of the value given to the constant.
5. If there is such a model, that model will be printed to the terminal by the final instruction, `get_model`.

This is the general form of most `SMT-Lib`-programs, and further definitions will look similar to `b1`, but with restrictions not only on initial constants. Since any new declaration can build on all the previous declarations, it is possible to pass very complex questions to the solver.

Z3 supports not only `Bool`, but several other useful sorts that map directly to familiar types, including unrestricted integers, restricted integers (modeled as bitvectors), and strings.

2.5.2 Theories

The following is an explanation of a few theories present in most modern SMT solvers.

Equality Logic and Uninterpreted Functions

A fundamental theory in SMT solving, it allows for equations stating that two values are equal [15]. This is very useful, and can be leveraged by the solver to simplify certain problems quickly.

The core of this theory is that without knowing anything about a function except that it must produce the same result if called with the same value:

$$a = b \implies f(a) = f(b)$$

This holds true for all pure functions, which include mathematical functions like the ones we are used to from arithmetic. Many expressions can be reduced to this theory by syntactic substitution, which means the solver can determine satisfiability without providing a specific interpretation of the function symbol.

Any set of equations that need $a = b$ and $f(a) \neq f(b)$ cannot have a satisfying model, and an SMT solver can determine this quickly without having to invoke the SAT solver, or any other solver for that matter, regardless of the sorts involved or the rest of the equations.

Theory of Arithmetic

A clear advantage of these theories is providing standard interpretations of common function symbols. Propositional logic have operators on booleans, like $\neg$ and $\vee$. Operations like addition and multiplication get the same treatment in the theory of arithmetic, and can be used directly, without having to tell the solver how the functions work by specifying the axioms of arithmetic.

Not only are the common arithmetic functions a part of the theory, but the numbers themselves form part of it, and thus all the known relations between them. The solver knows that integers commute and associate over addition, and will use this identity to its advantage if it can.

The program in Listing 2.2 shows usage of the arithmetic theory to find an instance of the Pythagorean theorem.[1] The theorem states that in a right triangle, the sum of the squares of the two shorter side equal the square of the longer side. This program finds one valid instance of the lengths of such a triangle. Three constants are declared, and then asserted to all be greater than 0. Then, another assertion is created, asking the solver to find values for the constants such that the value of c^2 is equal to the value of $a^2 + b^2$.

[1] https://en.wikipedia.org/wiki/Pythagorean_theorem

Listing 2.2: Arithmetic Theory in Action by finding an instance of the Pythagorean theorem.

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)

(assert (< 0 a b c))

;; Asserting that (a^2 + b^2) must equal c^2.
(assert (= (+ (* a a)
              (* b b))
           (* c c)))

(check-sat)
(get-model)
;;     Output:
;; sat
;; (model
;;   (define-fun c () Int
;;     5)
;;   (define-fun b () Int
;;     4)
;;   (define-fun a () Int
;;     3)
;; )
```

Also shown is the output from calling `get_model`. The model can be easily verified, as $5^2 = 25 = 16 + 9 = 4^2 + 3^2$.

2.5.3 Z3

Z3 is an SMT solver developed by Microsoft Research for program analysis, testing, and verification. [17] It is one of the fastest implemented SMT solvers, has been used in industry and academia to aid in verification, and multiple programming languages have libraries that serve as an interface to call the solver.

2.5.4 SBV

SBV [11] is a Haskell library that lets users *express properties about Haskell programs and automatically prove them using SMT (Satisfiability Modulo Theories) solvers.*[2] Providing bindings to all the features of SMT-Lib, it is

[2]http://hackage.haskell.org/package/sbv

possible to write generic constraints that can be sent to multiple different SMT solvers, including Z3.

Listing 2.3: SBV Example Program

```
f :: Symbolic ()
f = do
  b0 <- sBool "b0"
  constrain (b0 .|| sNot b0)
```

The example program in Listing 2.1 on page 12 has been codified using SBV in Listing 2.3.

Several unique aspects of Haskell will be immediately visible, such as the type signatures (may be cryptic to the inexperienced), but which is not necessary to understand for this use case.

The function `f` has two effects: It creates a free variable, restricted to a `boolean` and with the internal name `b0`, and it then creates a constraint that is the result of a disjunction (`.||`) between the existing variable by itself and negated.

SBV does some useful things while generating SMT-Lib code:

1. **Static Single Assignment (SSA):** Each new variable or constraint gets its own name. In Listing 2.3 this means that variables are created for `b0`, the negation of `b0`, and the disjunction. Since the `constrain`-call asserts that whatever is being constrained must be *true* for the model to be satisfied, it is this final variable that *must* be true in the final model. By having more constraints, rather than fewer and more compact constraints, there are more constraints for the solver to work with for finding counter-examples and for reusing existing values.

2. Additional analysis is done by SBV itself, looking for possible shortcuts when generating code. The clearest example occurs if the final assert is that a variable is equal to itself, in which case SBV will add `(assert false)` as the final clause rather than `(assert (not (= a a)))`. This is most often the case when testing whether a method commutes with itself, as the constraints created are identical.

   However, this is not the case for the program in Listing 2.1, and the generated output from Listing 2.3 can seen next to the arguably more natural way of writing the constraints for a human, in Listing 2.5.

Listing 2.4: The Hand-Coded example

```
(declare-const b0 Bool)

(define-fun b1 () Bool (or b0
    (not b0)))

(assert b1)
(check-sat)
```

Listing 2.5: SBV Generated Code

```
; tracks user variable "b0"
(declare-fun s0 () Bool)

(define-fun s1 () Bool
        (not s0))

(define-fun s2 () Bool
        (or s0 s1))

(assert s2)
(check-sat)
```

In addition to calling the solver by itself, there are multiple optional arguments that can be passed to the solver through the SBV-interface, leaving the user free to configure the solver to their liking.

Some of them are:

1. `verbose` will show the complete output as passed to the solver, as well as the response from the solver back to SBV.
2. `timeout` lets you set a maximum time the solver is allowed to operate before it terminates without a result.
3. `timing` will show how much time the solver spent on the different phases.

Perhaps the most interesting configuration is that SBV allows the user to choose between multiple solvers, or even running the same program through more than one solver at a time. For a complete list of solvers, see the documentation of SBV.[3].

2.6 Haskell as an implementation language

Some of the key reasons for choosing Haskell as an implementation language:

- It is considered a strong language for working with programming languages,[4] and often used for compilers or interpreters. There are a lot of similarities between that and what this analysis is doing. Haskell has algebraic data types make it easy to construct the syntax tree needed to represent the ABS program.

[3]https://hackage.haskell.org/package/sbv-8.9/docs/Data-SBV.html
[4]https://github.com/Gabriel439/post-rfc/blob/master/sotu.md#compilers

- There exists several very well developed libraries that create bindings into SMT solvers. `SBV` was chosen. It is a Domain Specific Language for creating SMT-Lib constraints from Haskell. `SBV` also creates good SMT-lib code.
- It is a mature language with robust libraries for performing high-level operations, including JSON parsing,[5] on the other side of the spectrum from a language like C in terms of the memory model and levels of abstraction.

The reader should note that familiarity with Haskell is not necessary for the theoretical aspects of this thesis, and only necessary if one is interested in the source code of the implementation. The results should be reproducible in other languages with bindings into SMT-Lib.

2.6.1 Algebraic Data Types

One key aspect of Haskell is the support for algebraic data types, or ADTs. ADTs are the way to define data structures in the language, and is used extensively to model the abstract syntax tree that is eventually used to create constraints.

ADTs can create composite types, which can be nested further to create arbitrarily complex data. This means that the programmer is free to model whatever they want, and is an excellent way to model ASTs. Here is the way `Bool` is modeled in the language.

```
data Bool = True | False
```

The fundamental nature of ADTs in the Haskell ecosystem makes it very pleasant to clearly define the domain a as data types, and then create functions that use them. Combined with the strong type inference, the compiler guides you towards code what works, and works correctly.

[5] https://hackage.haskell.org/package/aeson

Chapter 3

Abstract Behavioral Specification

This chapter will cover the key features of Abstract Behavioral Specification (ABS) which make it both interesting and possible to investigate using static analysis. Certain aspects of this analysis relies heavily on the features in the ABS languages. The syntax of ABS is similar to Java and C, and will be familiar to most programmers.

ABS has been used to analyse, e.g. complex low-level multi-core systems [8], cloud-deployed software [1], and railway networks [14].

3.1 Key Features

ABS[1] [13] is an *active object* language, in addition to being both object-oriented and concurrent, with support for first-class futures. An active object language has some specific features, and these features make it possible to make assumptions about our analysis that we could not make for mainstream imperative languages like Java or C#.

The most important language features for this analysis are:

1. **Distributed, actor-based semantics:** Each object runs on its own thread, and can only manipulate fields inside other objects indirectly, calling a method on that object asynchronously, which then has to wait its turn to run on the thread of that object. This means that only the methods on a class can run on the thread of that object, and that thread is the only way to alter any state inside of the object.
2. **Cooperative scheduling:** Each object will only schedule a new process from the set of waiting processes once *the running process itself* yields control. Since each methods must yield control itself,

[1] https://abs-models.org/

it is clear from the code whether a method will run to completion or yield control at some point during its execution. In the case when a method yields control behind a conditional, the method is considered non-atomic. If there are no *scheduling points* during the body of the method, we know it will run all the way to the end in one execution.

A scheduling point, or *release point*, is any point in the program that causes the scheduler to activate.[2]

The result of these two features is a language well-suited to certain forms of static analysis. The most useful consequence of these features for our analysis is that much longer blocks of code become atomic. Without each object on its own thread, the internal state of the object could be mutated at the same time by two different threads. If there was no cooperative scheduling, there would be no way to determine how much of each method would be executed at one time. Most mainstream languages feature *preemptive scheduling* instead of cooperative scheduling. This means that the scheduler can interrupt a process at any point and schedule a different process.

Instead, we can determine statically how much code will be executed while also knowing that the current method is the only one altering the state while it is running. That is, we can identify atomicity at the level of methods. In languages with preemptive scheduling atomicity can only be assumed at the level of statements, and even then certain statements are complex to the point where one statement in the high-level language is translated into multiple atomic statements in the underlying machine/byte-code. Any attempt at finding commuting statements would be futile without restricting the programs to be written in very specific ways to simulate an active object language.

In ABS, all objects reside on a COG,[3] a Concurrent Object Group, and each COG runs a scheduler in charge of the processes on that cog. However, we assume for the entirety of this analysis that every object resides on its own COG, and can work as if each object has its own scheduler.

3.2 Example ABS program

The ABS program in Listing 3.1 on the facing page will be used for later examples of code generation and solving in this chapter and the next, and has been written with several key points in mind:

[2] https://abs-models.org/manual/#sec:side-effect-expressions
[3] https://abs-models.org/manual/#sec:concurrency-model

Listing 3.1: Example ABS program

```
module Example;

interface Count {
    Unit increment();
    Unit setBool();
}

class CountImpl implements Count {

    Int a = 0;
    Bool b = False;

    Unit increment() {
        if (b) {
            a = a + 1;
        }
    }

    Unit setBool() {
        b = !b;
    }
}
```

In terms of syntax, ABS is close to Java and most modern imperative languages. For a program, a module is specified first. In the case of Listing 3.1, there is an interface declaration, followed by a class that implements the interface. The two fields inside the class are local to that class, and only the methods on this class can read and write them.

1. Both a boolean and an integer are being read to and written to, and both fields are used in at least one method.
2. Both methods commute with themselves, but the methods do not commute with each other.
   (a) Calling increment twice `increment` will either increment `a` twice or not at all, depending on the prior state of `b`.
   (b) Calling `setBool` twice will leave `b` in the same state as before.
   (c) The final state if both `setBool` and `increment` is called once depends on the state of `b`. `b` will end up negated, but whether or not `a` is incremented will vary.

Chapter 4

From ABS to Haskell

We now consider the toolchain which will transform the abstract syntax tree (AST) of an ABS program into data types in Haskell. This transformation consists of two steps; the AST is first turned into JSON, which is then parsed and interpreted as Haskell algebraic data types.

The analysis tool is in many respects a compiler, exhibiting traditional compiler features such as transforming between different representations and producing output runnable by a different piece of software. This chapter and the next will cover these transformations.

4.1 The Code Pipeline

Figure 4.1: Data Transformations

Analyzing a single program involves several steps from the original source code in ABS to the final output as pairs of commuting methods.

1. The source code is transformed by the ABS compiler into an internal representation. This internal representation is an abstract syntax tree (AST), modeled as classes in Java. As part of this process the program is also type checked, and if the transformation succeeds the AST corresponds to an executable program. The code is added to the ABS compiler using the `jadd`-extension[1] to gradle, which lets the programmer write all the code for each class in a single file. The tool will then add the methods for each class as part of the compilation process.

[1] https://github.com/d10xa/jadd

This addition to the compiler becomes part of ABS Tools, and will allow others to output a generic version of the syntax tree for their own use.

2. This internal representation is translated to JSON by invoking the modified ABS compiler with the `--jsonTree`-flag. Each JSON object contains a name referring to the type of object it is in the AST, as well as any fields required by the object, e.g. an assignment will have a left side and a right side, and an `if`-test will have fields for the condition, the conditional branch, and the alternative branch.
3. The resulting JSON is passed to the analysis tool written in Haskell, producing a representation of the AST in algebraic data types. The Haskell data types cover a non-trivial subset of ABS.
4. With the AST as a Haskell data type, aspects of the program are transformed into `SMT-Lib`-code that can be sent to an SMT-solver. Each pair of methods in the original program is sent to the solver as distinct queries to determine if they commute.
5. The final output of the analysis is a list of pairs, each pair being names of commuting methods.

4.2 Translating ABS to JSON

To export the AST from the ABS compiler, it is printed as JSON to the standard output. This is done by walking the AST top-down and printing a representation of the current object into JSON, recursively. To generate JSON from an ABS source code file, pass the flag `--jsonTree`.

The compiler will fail to compile if there is a syntactic error in the source code, or if it fails to type check the program. The rest of this analysis can therefore work with the assumption that the AST represents an executable program.

JSON is a general-purpose text syntax for facilitating data interchange between computers,[2] and is used due to its ubiquity which nearly guarantees support for parsing it regardless of language.

Each language construct of the internal AST representation has its own Java class. To make the compiler able to output JSON involves writing a method for each relevant class specifying how that particular class should be serialized. For different classes, there are different fields of interest that must be included.

[2]https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf

Listing 4.1: Example ABS program as JSON

```
{ "ModuleDecl":
 { "ModuleName": "Example"
 , "Declarations":
  [ { "NotSupported": "InterfaceDecl" }
  , { "ClassName": "CountImpl"
   , "Parameters": [ ]
   , "Fields":
   [ { "Type": "Int"
     , "FieldName": "a"
     , "InitExp": { "IntLiteral": 0 }
   }
   , { "Type": "Bool"
     , "FieldName": "b"
     , "InitExp": { "DataConstructor": "False" }
   }
   ]
    , "Methods":
   [ { "MethodName": "increment"
     , "ReturnType": "Unit"
     , "Params": [ ]
     , "Body":
       [ { "IfStmt":
           { "Condition": { "FieldName": "b" }
           , "Then":
             [ { "AssignStatement":
              { "Variable": { "FieldName": "a" }
              , "Value":
               { "Operator": "AddAddExp"
               , "Left":  { "FieldName": "a" }
               , "Right": { "IntLiteral": 1 }
               }}}]}}]
     }
   , { "MethodName": "setBool"
     , "ReturnType": "Unit"
     , "Params":
      [ ]
     , "Body":
     [ { "AssignStatement":
      { "Variable": { "FieldName": "b" }
      , "Value":
       { "Operator": "NegExp"
       , "Value": { "FieldName": "b" }
       }
      }}]}]}}
```

The JSON in Listing 4.1 shows the serialization of the program in Listing 3.1 on page 21. Since JSON is not white space sensitive, the visual output is maintained mainly for debugging and human understanding of the serialization. The easiest way to find any bugs is to open the JSON in a code editor and have it warn you of any errors. There are currently no known errors being produced.

At the top level there is the module declaration from ABS as a JSON object, with the module name as one field and a list of declarations as a second field. These various supporting structures of ABS are necessary to include in the JSON, in order to make the serialization a useful tool in general. Ideally, it is good enough so that others can use it to gain access to the ABS syntax tree. For the analysis in this thesis, only the methods inside a class are needed. The interface declaration is superfluous for our needs, as the methods themselves contain all the necessary information, and it can be seen from the JSON that the serialization can produce output despite not supporting the serialization of specific constructs. The only other part needed are the field declarations of the class.

Listing 4.2: Serializing `if`-stmt into JSON

```
public void IfStmt.jsonTree(CodeStream stream) {
    stream.println("{ \"IfStmt\":");
    stream.incIndent();
    stream.println("{ \"Condition\": ");
    stream.incIndent();
    getCondition().jsonTree(stream);
    stream.decIndent();
    stream.println();
    stream.println(", \"Then\": ");
    stream.incIndent();
    getThen().jsonTree(stream);
    stream.decIndent();
    stream.println();

    if (hasElse()) {
        stream.println(", \"Else\": ");
        stream.incIndent();
        getElse().jsonTree(stream);
        stream.decIndent();
    }

    stream.println("}");
    stream.decIndent();
    stream.println(" }");
}
```

The serialization uses the `jsonTree`-method to recursively serialize nodes in the syntax ree. The example in Listing 4.2 shows the serialization for `if`-statements.

4.2.1 Implementation Details

- The method `jsonTree` is created as a method in the class that should be serializable. In the case of Listing 4.2, the class is `IfStmt`. This makes the method available to the class at runtime, and the

method has access to the internal variables and methods of the class, such as `getCondition`.

- The method itself takes as an argument a `CodeStream`, which is a class in the compiler backend written to provide an interface for printing from the AST to the standard output. It has methods for printing, indenting, and dedenting, which makes it suitable for our needs.
- To create valid JSON, curly brackets must be printed, as well as quotes surrounding key words. Some whitespace is added, but this is purely aesthetic to make it easier on the human eye.
- The methods `incIndent` and `decIndent`, a part of the `CodeStream` interface, are used liberally to make the output easier to read.
- In the case of the `if`-statement, the alternative branch is optional. As a result, is it behind a guard and will only be included in the JSON output when it exists in the AST. This kind of optionality is something Haskell is built to handle, and will be reflected in the corresponding data type declarations.
- `jsonTree` is called recursively on nodes further down in the AST. These nodes are returned by methods like `getCondition` and `getThen`.

4.3 Algebraic Data Types in Haskell

To represent the AST in Haskell we make use of algebraic data types to model all the necessary constructs.

In Listing 4.3 on the following page some of the algebraic data type declarations can been seen, including all the definitions required to construct a data type that includes a `Module` on the top-level, all the way down the syntax tree to the statements that make up method bodies.

At the top level there is a `Module`, with a name and a list of declarations. The declarations themselves are in the `Decl`-datatype, and here only the `ClassDecl` has been fully fleshed out. The other declarations are part of ABS, but are not relevant to this particular analysis. As part of the class declaration is a list of methods that belong to that class, which is created using the `Method` data type, and each of these methods have a list of statements as their body.

The `Module` and `Method` types are examples of product types, mentioned in Section 2.6.1 on page 17. They require all their fields to be present in

Listing 4.3: Some examples of the Algebraic Data Types

```
data Module = Module
  { moduleName :: String,
    moduleDecls :: [Decl]
  } deriving (Show, Data, Generic)

data Decl =
  ClassDecl
    { className :: String,
      classParams :: [Param],
      interfaces :: [String],
      classFields :: [FieldDecl],
      classMethods :: [Method]
    }
  | InterfaceDecl
  | TraitDecl
  | FunctionDecl
  | PartialFunctionDecl
  | TypeSynDecl
  | DataTypeDecl
  | ExceptionDecl
  deriving (Show, Data, Generic)

data Method = Method
  { methodName :: String,
    returnType :: Type,
    params :: [Param],
    statements :: [Statement]
  } deriving (Show, Data, Generic)

data Statement
  = SkipStmt
  | VarDeclStmt String Type (Maybe Exp)
  | AssignStmt Exp Exp
  | ExprStmt Exp
  | AssertStmt Exp
  | ReturnStmt Exp
  | Block [Statement]
  | IfStmt PureExp [Statement] (Maybe [Statement])
  | WhileStmt PureExp [Statement]
  deriving (Show, Data, Generic)
```

order to be created. The `Decl`- and `Statement`-types are *sum types*, and require that only one of their possible variants be created at one time.

In the statement data type, the `SkipStmt` has no fields, and can be constructed by itself, while an `AssignStmt` requires two expressions, corresponding to the left and right side of the assignment operator.

This example showcases the terseness of Haskell, codifying `if`-statements in a single line as a product type taking a pure expression, one mandatory list of statements, and an optional list of statements. Since `if`-statements in ABS are not required to have an alternative branch, this optionality is formalized by using the `Maybe`-type. This particular type, `Maybe [Statement]`, may contain a list of statements or be a `Nothing`, and the type system will enforce checks at compile time that both of these cases are covered.

4.3.1 The AST as Algebraic Data Types in Haskell

Listing 4.4 on the following page shows the syntax tree as an instantiated Haskell data type, from the JSON in Listing 4.2 on page 26. The correspondence between the JSON and Haskell data types is close, and most of the difference comes down to the Haskell types having names rather than being an object with a `name`-field. Rather than types as strings in JSON, `Bool` and `Int` now refer to actual Haskell types. In the field declarations, the initial expressions are optional. In this case they are present in both examples, as required by the implementation language for simple types like `Bool` and `Int`. The other example is in the `if`-statement.

Looking at the `if`-test in the first method is a good example of how the data types work. The test is a `PureExp`, in this case a `FieldUse`. The first branch is a list of one assignment statement, which requires two expressions, the first of which must be a local or global field in the program. In the second expression the natural nesting is visible, as the `Operator`-expression contains a binary expression, which is a larger object with an operation, a left side, and a right side. In this case, the operation is addition, with a field on the left side and a constant literal on the right side, matching what is seen in the JSON.

One very useful feature of the ABS syntax tree can be seen in the test of the `if`-statement, as the AST refers to a `FieldUse`. Rather than just specifying a variable name `b`, there is specified a `FieldUse` and a variable, informing us that this is a class field rather than a local variable in the method. This makes the analysis much easier, as it does not have to keep track of scoping.

Finally, the alternative branch in the `if`-test is `Nothing`, the value of the

Listing 4.4: Instantiated AST in Haskell

```
ModuleDecl
  (Module
   { moduleName = "Example"
   , moduleDecls =
     [ ClassDecl
       { className = "CountImpl"
       , classParams = []
       , classFields =
         [ FieldDecl
           { fieldName = "a"
           , fieldType = Int
           , fieldVal = Just ( Literal ( IntLiteral 0 ) ) }
         , FieldDecl
           { fieldName = "b"
           , fieldType = Bool
           , fieldVal = Just ( DataConstrExp "False" ) }
         ]
       , classMethods =
         [ Method
           { methodName = "increment"
           , returnType = Unit
           , params = []
           , statements =
             [ IfStmt
               ( FieldUse "b" )
               [ AssignStmt
                 ( PureExp ( FieldUse "a" ) )
                 ( PureExp
                   ( OperatorExp
                     ( BinaryExp
                       { binaryOp = Addition
                       , left = FieldUse "a"
                       , right = Literal ( IntLiteral 1 )
                       }))))]
               Nothing
             ]
           }
         , Method
           { methodName = "setBool"
           , returnType = Unit
           , params = []
           , statements =
             [ AssignStmt
               ( PureExp ( FieldUse "b" ) )
               ( PureExp
                 ( OperatorExp
                   ( UnaryExp
                     { unaryOperator = Not
                     , unaryExp = FieldUse "b"
                     }))))]}]}]})
```

`Maybe`-type when the optional value is not present, which is what we would expect from JSON example where there was no alternative branch.

Having made several transitions from the original ABS source code into Haskell data types, the next step is to create constraints that can be passed to the solver.

Chapter 5

Identifying Commuting Methods

Before looking at the rest of the implementation, we will cover the theory of how methods might commute. This chapter will cover commutativity in general, how one might think of it in terms of SMT solving and read-write sets, before considering the various ABS constructs individually.

The purpose of the analysis is to determine if two methods commute. Before looking at the implementation, we will consider each language construct in turn to consider how they should be treated as we implement the analysis.

Because the analysis is on an active object language with cooperative scheduling, it can work with methods as atomic units.

5.1 What is Commutativity?

Deciding on and identifying commutativity is the key aspects of this analysis, so it is necessary to understand how commutativity works, in particular with regards to programming statements. While commutativity in mathematics and logic is clearly defined, it is a slightly different concept of commutativity we are working with.

In mathematics, commutativity is defined on *binary operations* where changing the order of operands does not change the result [3]. Furthermore, an *operation* is a calculation that takes two elements from a set and produces another element in the same set. One well-known commutative operation is addition on the positive integers. Because its a commutative operation, it is known that the following formula is true for all values A and B, where A and B are positive integers.

$$A + B = B + A$$

In predicate logic, the *commutative laws* state that the order of operands does not matter for conjunctions and disjunctions [3].

$$A \wedge B \Leftrightarrow B \wedge A$$
$$A \vee B \Leftrightarrow B \vee A$$

Creating a single statement from two statements is traditionally done with the semicolon in mainstream imperative languages, and is known as *sequential composition*. Ideally, we want to know if, for two statements S_1 and S_2

$$S_1; S_2 \equiv S_2; S_1$$

The semicolon takes two statements and produces a single statement. This can be done again, making this compound statement part of a larger compound statement. Compound statements can then be further composed to form even larger compound statements, growing in complexity and becoming more difficult to analyse.

When proving the commutativity of addition on the natural numbers, we can use structural induction, and because a number is either zero or some successor of zero, there are few cases to deal with. For statements in a programming language, there are many more cases to deal with, and on top of that there is *external state* to consider, in the form of existing variables that might be referenced.

Rather than proving formally anything about commuting statements, we want to use an SMT-solver to simply execute programs symbolically, and have the solver tell us about the possible values of the state.

5.2 SMT Solving for Commutativity

The general idea is that the class contains some state S at time t_0. Then time is split, and in one version method $M_{\mathcal{A}}$ is executed, leading to state $S_{\mathcal{A}}$ at time t_1, while in the other version method $M_{\mathcal{B}}$ is executed leading to state $S_{\mathcal{B}}$ at time t_1. Then, in each separate timeline the method that was not executed the first time is now executed, leading to states $S_{\mathcal{AB}}$ and $S_{\mathcal{BA}}$ at time T_2, where the ordering of the superscript refers to the order of execution. If these two states are the same for all possible initial values of S, the two methods are said to commute.

The analysis is thus concerned with a very local version of commutativity, only ever considering the states at three different times. Whatever happens to the state before we start analysing it, or after the analysis has finished is not relevant. One consequence of this is that the analysis cannot make any assumptions about the state at time t_0. Whatever initial values were assigned as part of the initialization of the class might have been changed many times over. The only information we know for certain is the type of each field, due to the static typing of the underlying language.

The idea of a completely unconstrained variable with a type is exactly what a free variable in many-sorted first-order logic is. Each field is free to hold any value within its *sort*. With each variable codified as a free variable, *simulating* each statement really means applying more and more constraints on the values of each variable according to the statements in each method. Then, once this has been done for both statements in both orderings, we must analyse all the possible states to determine if the states can be different, or if they are the same of all possible initial values of the free variables.

From a logic perspective, we know that:

$$\forall S(S_{\mathcal{AB}} = S_{\mathcal{BA}} \implies commute(S_{\mathcal{AB}}, S_{\mathcal{BA}}))$$

However, while proving something for every possible state is necessary to have our proof, that means the overall strategy is to look for an initial state in which the two final states are not the same.

$$\exists S(S_{\mathcal{AB}} \neq S_{\mathcal{BA}} \implies \neg commute(S_{\mathcal{AB}}, S_{\mathcal{BA}}))$$

The main reason for *swapping* the question around is due to the nature of the solvers. By asking if two states are the same, it will look for a single confirming initial state. However, we want to know if the final states are equal for *all* initial states. By turning the question around, the solver is now looking for a single disproving initial state. If no disproving state is found, all initial states S must case identical states $S_{\mathcal{AB}}$ and $S_{\mathcal{BA}}$.

The key condition for whether two statements commute concerns the state before and after executing each statement, and how each statement affects any state needed by the other statement. While using an SMT solver to consider this question, we will also look at a read-write analysis as a naive and easier-to-implement way to achieve some of the same goals.

5.3 Read-write sets

Before delving into constraint solving, a simpler way to determine whether two methods *might* not commute is to compare the fields that they read from or write to. Such an analysis is not very complex and will not be complete, but it should provide a sound *approximation*. A read-write analysis will, for each method, produce one set of fields written to, and one set of variables that are read. Then it is possible to compare methods by operations on these sets, and determine the commutativity of the methods themselves. The restrictions are that the methods cannot write to the same field, nor can they read a field that the other method is writing to. That is, the intersections of these three possible combinations must all be the empty set. This analysis reduces each method down to the bare minimum needed to check whether two methods interfere with each other in any way.

Table 5.1: Commuting Table for Read-Write Analysis

	Method A Read	Method A Write
Method B Read	Commute	Not Commute
Method B Write	Not Commute	Not Commute

Table 5.1 shows how stringent the requirements are for the analysis to decide whether two methods commute, and it must be considered commuting for *all* fields.

Any pair of methods in which none of the methods alter any fields, must necessarily commute. By not altering any fields, they cannot possibly change any behaviour in the other method. This would include getter-methods, and any method that only produces a return value rather than modifying the internal state of the class.

By having access to a simple yet sound analysis like a read-write analysis, it is possible to use it as a back up for any method in ABS that contain features not supported by the more advanced analysis. The more advanced analysis can be the starting point, and if it encounters an unknown construct, it stops. This way, it is not necessary to first check if the analysis supports all the constructs in the methods before trying to do the actual analysis. Furthermore, the read write-analysis can make use of the same data structure, and thus avoid having to parse the same methods multiple times.

5.4 Commutativity of ABS Constructs

The overall idea of comparing the state was presented earlier, in Section 5.2, with the state being set at some point t_0, and then branching out until we reach t_2. Then, the two possible versions of the state at t_2 are compared, and if they are the same for all possible initial value, the two ways to get there are considered commuting. Let us first consider a very simple example.

Listing 5.1: Simple method that commutes with itself

```
Int a = 0;

Unit increase() {
    a = a + 1;
}
```

Listing 5.1 shows a very simple method that increments a global field. In the case of a single method, we want to know if that method commutes *with itself*. In this case, it is straight-forward to see that `increase` commutes with itself. Regardless of the *ordering* of increments, in the end the number is incremented twice every time. Since we are dealing with unbounded integers, there cannot be overflow. However, we have to make the solver see that the two methods commute. We can consider it in terms of *steps of time*.

Time t_0

The field is initialized to zero in the example, but to the SMT solver this is a symbolic variable with no specific value. We do not know what other methods might have executed before the method we are looking at, and our analysis must consider every possible value of `a`. If there are any other fields, they must also be initialized at this time. Additionally, any parameters passed to the function should be initialized as well, again with all possible instances of their type as a value. The analysis does not know any values the methods are called with, and must account for anything here as well, just like the fields. Any locally defined variables will be added when those statements are modeled by the analysis, as they are part of the body of each method.

For `increment`, the only relevant field is `a`, and it is considered any possible integer.

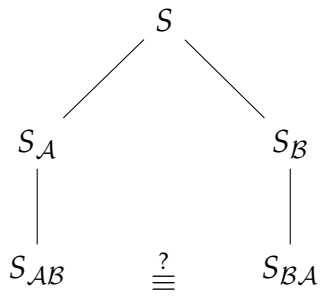

Figure 5.1: Comparing States Over Time

Time t_1

At time t_1, the states have been updated to reflect the changes made by the first executed method. The analysis must split in two, keeping two separate states in mind. In this case, the two branches do the same thing, incrementing a single variable. When incrementing, a new state is created that refers back to the old state. It is important that both branches refer to the original state, or the initial values of each state could be modeled as different, and that could make the final states different even if the methods commute.

Time t_2

At time t_2, the final state for each branch has been created, after running the second method on the state from t_1. Those states are compared, to look for fields that differ in their values. Given our example method, it will be clear that `a` has been incremented twice in both branches. Regardless of the initial value of `a`, it will be the same in both branches.

5.5 How Different Constructs Affect the State

Here we discuss how expressions, assigments, and control flow statements affect the state.

5.5.1 Expressions

Expressions are entities in a programming language that may evaluate to a value. How these values are generated will impact the analysis. In general, one may consider pure expressions and effectful expressions.

Pure Expressions

Pure expressions do by definition not cause any side effects, and are simple to consider from the perspective of the analysis. They are restricted to only returning values, and while doing so might read from the state, it cannot write to the state.

Pure expressions include:

- using fields and variables, which mean reading them from the state.
- Any literals.
- Operators, such as unary and binary operators on integers and booleans.
- Calls to pure functions.

The theory of arithmetic is one of the theories inside modern SMT solvers, and it is therefore possible to formalize expressions on integers in a straight forward manner. The same is true for operations on booleans, and the combination between them. For integers and booleans, the theories also include various axioms such as commutativity and associativity. As a result, constraints placed on these values can be optimized further by the solver as it tries to find counter-examples to them, and methods dealing only with integers or booleans will have a solver working very well.

Effectful Expressions

Among these expressions are two of the culprits that cause the analysis to fail to determine whether two methods commute. This happens when an expression requires the current method to suspend.

The expressions that cause a suspension are `await`-expressions and synchronous calls. An `await`-expression is equivalent to an asynchronous call statement, followed by an `await`-*statement*, followed by a `get`-statement, and it is the `await`-statement that causes a scheduling point.[1]

The synchronous call-expression has semantics that depend on the *COG* of the caller and the called. For the sake of this analysis we assume that each object runs on its own COG.[2] As a result, it depends only on whether the call is made on the same object that makes the call or not. If it is a synchronous call on *this*, it will cause the other method to run immediately, and thus acts as a scheduling point. If it is not on *this*, the method will

[1] https://abs-models.org/manual/#sec:side-effect-expressions
[2] https://abs-models.org/manual/#sec:concurrency-model

block, but not suspend, and is therefore within what the analysis can deal with.

The rest of the effectful expressions do not change any state in the class as they call methods on other objects. Asynchronous call expressions places a new process in the target class, and evaluates to a future in the local scope. This future might be assigned to a variable as part of an assignment.

`New`-expressions can be dealt with by considering the arguments passed to the constructor. Since objects are created on their own thread, this means that a new object cannot alter the state of the current class as it is being created. Thus, a `New`-expression is deemed commuting if the object is created with the same arguments. A `New`-expressions can cause some initial code to run in the new object, but this code cannot run any code that causes a scheduling point, and can therefore not impact the class being analysed.

5.5.2 Assignments

Assigning new values to existing variables is clearly a key part of this analysis; in fact, it is the only way to have an effect on commutativity. Without assignment, this analysis would be moot, or at least it would be interested in commutativity in a very different sense. Languages without assignment, such as certain functional languages have *referential transparency*, in which any expression can be replaced by its return value. This is not the case for ABS. Generally, when the variable in question is of a type that is supported by the solver, a new constraint is created that refers to the current constraints, and then this new constraint is set is updated as the new state of that variable. Then, whether it breaks commutativity is up to the solver to determine.

5.5.3 Branching — If Statements

Do deal with the possibility of branching during execution, there is an `if-then-else` construct in SMT solvers, which will create a way for the solver to determine which branch is realized during the symbolic execution of the program.

The solver deals with branching by generating constraints based on both possible branches and then deciding which one to keep depending on the value of the boolean expression during testing. The test expression is captured before any of the branches are executed, and can be thought of in similar terms to an `if`-statement in a familiar programming language, like Java or C.

Following the branching, one of the possible states have been chosen and will be the state that is used in the rest of that symbolic execution. An `if`-statement on its own is thus fine to deal with for the solver, and for the analysis it comes down to the contents of each branch rather than the branching itself.

In the existing literature on Static Single Assignment, this function that picks a branch from multiple possibilities is known as a ϕ-function [18].

5.5.4 Branching — While Statements

To deal with loops the analysis will *unroll* the loop into a series of `if`-tests, and at the end of each `if`-test the test will be symbolically executed to determine which branch to take the constraints from. The challenges with the actual construction of these constraints are discussed in Section 6.3.4 on page 54, but can be summarized as:

- The analysis does not know beforehand how many iterations of the loop are needed to exit the loop. Since all the possible branches have to be created before the symbolic execution of the program, a bound on the number of branches to create must be specified before the constraints are created.

- Since there is an upper bound on the number of iterations that will be simulated, it is possible for the number of available iterations to be insufficient to determine the actual constraint placed on the field in a real run of the program. If this is the case, the analysis might not be sound.

Since hitting the bound when iterating is a problem, the program cannot make a definitive statement about commutativity if the final state is chosen by the analysis. The final state is then chosen just because it was the state at the bound, and not because is represent a valid state from that point forward. Since it could be that the program will *not* commute once enough iterations of the loop have been executed, the program will be deemed *not commuting* if the number of iterations hit the bound.

It is also possible to consider any loop that hits the bound to be commuting, if nothing before the bound caused it to become non-commuting. This could potentially be added as an option to the solver, leaving it to the user how to deal with bounds.

5.5.5 Effectful Statements

The effectful statements are `suspend` and `await`, and they are not supported by this analysis. Because the cause a scheduling point in the middle of the method, the method is no longer atomic.

The `suspend`-statements does nothing but suspend the current process, placing the current methods back in the scheduler queue, and the scheduler decides what to do next. The `await`-statements will also suspend the current process, but does so with a guard that must become `True` before the process is allowed to be rescheduled.

See Section 7.1.1 for a discussion on how this might be resolved in the future.

Chapter 6

Creating Constraints

This chapter is concerned with actual transforming the abstract syntax tree, represented with algebraic data types in Haskell, into constraints. SMT-Lib is a language specifically created to be a common language for input to SMT solvers, and serves as the target. The Haskell library used to encode constraints into SMT-Lib does neither require knowledge of the underlying language nor expose it to the user, unless explicitly told to do so. However, to fully understand *why* the constraints are constructed the way they are, it is necessary to first understand the idea that is used for identifying commuting methods. This chapter will first cover the general idea of how constraints can be used to determine commutativity, then a hard-coded example using SMT-Lib, and then have the tool generate its solution to the same problem. Solving the problem by hand will help familiarize the reader with both the mental model for SMT solving and usage of SMT-Lib, as well as provide a baseline for comparison with the generated code.

6.1 Construction of SMT-Lib constraints

The goal of this stage of the analysis is to create constraints that can be symbolically executed, in such a way that the analysis will consider two methods commuting when they actually *commute*, using the concept of commuting as defined in the previous chapter. While the constraint generation code uses reassigns variables in order to represent new states, the generated code itself uses Static Single Assignment (SSA) to represent changes to the same variable over time. The actual symbolic execution happens once the solver is invoked. Everything before that is constraint generation. However, in the same way one might reason about the runtime of a program while writing code, it is possible to reason about

the symbolic execution while writing constraints. And even though what is symbolically executed is a set of logical equations, we can consider the order they are constructed in as the order they will be *executed* in due to how each new constraint refers back to an earlier constraint.

When checking for commutativity between two methods, the following steps are executed:

1. A map of the shared fields will be created, where the name of each field refers to a free variable constrained only to be the correct sort, e.g. `Integer`. This same map is used in both runs, to ensure that the same initial variables are used in later constraints.

2. Each method gets its own map of local variables.

3. Generate constraints for one possible execution path of the two methods in question.

4. Now, swap the order of the two methods and generate a new set of constraints. This is a high-level view of the branching, and says nothing about whether additional branching occurs within each method.

5. The solver tries to satisfy the constraints, only succeeding if there is an assignment to the initial variables that cause the final states to be different. If the solver is unable to satisfy the constraints, it means no counter-examples were found.

If no counter-example is found, the methods are proved to commute.

Static Single Assignment

Static Single Assignment (SSA) is used by SBV when generating its SMT-Lib code. Every time a field is constrained in a new way, its new constraints are entirely the result of already known constraints, and so a new free variable is constructed, constrained immediately by some set of previous constraints.

Assumptions

We are making several useful assumptions about the programs we are analyzing, which we are justified in making because the syntax trees that we are operating on have first passed through the ABS parser and type checker.

- The program is syntacticly correct. It is not the task of the analysis tool to check correctness, and a valid syntax tree is assumed at every step.
- The program is well-typed. We get the program after the type checker phase of the compiler has approved it. This is very useful to us, as it means we do not have to worry about the program trying to create illegal constraints on types. Since Haskell is a statically typed programming language, this makes implementation easier.

The logical correctness of the program being analysed is left to the programmer.

6.2 Encoding First-Order Constraints by Hand

Before having the constraints constructed for us by the analysis tool, let us construct them ourselves. This will provide a better understanding of how constraints can be created to represent executable code, and provide a point of comparison for the later generated code.

Let us consider the `increment` and `setB` methods from Listing 3.1 again:

```
Int a = 0;
Bool b = False;

Unit increment() {
    if (b) {
        a = a + 1;
    }
}

Unit setB() {
    b = !b;
}
```

This example contains modification of a class field and branching with an `if`-statement.

Declaring Class Fields

As discussed in the previous section, each class field is assigned a free variable with the correct sort. In the current example, there are two fields in this class to initialize. Their declarations have been included, and we do not have to infer the types ourselves. To create these two fields in SMT-Lib

we use `declare-const`, which takes a name and a type, and creates free variables restricted to that type. In this case, one `Int` and one `Bool`. Since mutation is not possible on constraints, new constraints are created that refer back to the previous versions for a particular variable.

```
(declare-const a0 Int)
(declare-const b0 Bool)
```

These two constants represent the initial values of the class fields we are interested in. Now we want to determine the constraints placed upon these constants by symbolically executing our methods in both possible orderings. As such, one possible run is `increment` first, followed by `setB`.

Run: `increment` → `setB`

First, we have to deal with an `if`-statement, and that means branching. In this case there is no alternative branch, which means that a negative test means the original state is kept. In a normal program, code for both branches would be generated, but only one branch would be executed depending on the test at runtime. For SMT solving both branches must be generated *and* symbolically executed, but the choice of state depends on the values during the symbolic execution. By the state, we mean the set of constraints that exist at the end of each branch. In this particular case, one state is the state from before the `if`-statement, and the other is the state where the `a` has been incremented.

The concept of branching is encoded into SMT solving with the `ite`-function, which is the symbolic equivalent of an `if-then-else`-expression. However, as mentioned in the previous paragraph, it does not decide on which branch to execute, only on which variables are kept. The constraints for each branch has to be created, and then encoded into the `ite`-function, along with test that is a symbolic boolean. Each branch of the `ite`-function will encode some version of each variable, and then it will create another new variable that picks the correct variable from one of the branches. The symbolic value of the test determines which branch these constraints are chosen from. For example, some variable $v0$ can be encoded with a new constraint as $v1$ in one branch and $v2$ in another branch. Then, some variable $v3$ will be created that is set to either $v1$ or $v2$ depending on the value of the boolean, and $v3$ is used as the representative of v from that point onward.

```
(declare-fun a1 () Int (ite b0 (+ 1 a0) a0))
```

This constraint will make `a1` hold the correct version of the state, which will depend on the value of `b0` while running the symbolic execution. This one statement describes the entire `increment`-method, generating one new variable which is part of the new state. This new variable must be accounted for when the constraints for the next method are formalized. The `setB`-method is a simple boolean negation, and is trivially encoded as follows:

```
(declare-fun b1 () Bool (not b0))
```

Again it is important to use the latest version of `b`, which is the initial `b0`. With these two statements we have created the new versions of both fields for one run, and its time to consider the alternative execution.

Run: `setB` $\rightarrow$ `increment`

Now to compare, the ordering of the methods is swapped and the `setB` method is executed first and the `increment` method second. We must remember to use the original free variables `a0` and `b0` when creating the new constraints.

First, the `setB`-method is the same statement as in the other run.

```
(declare-fun b2 () Bool (not b0))
```

Since `b1` was created in the previous branch, `b2` is used. However, notice that this is the exact same constraint as created for `b1`. To make this elaboration easier to follow, we will keep the two variables separate for now, but we will come back to how this is not necessary, and that it can be very useful to *not* use separate variables for identical constraints. Now, the constraints for the `increment`-method using our newly minted `b2` variable as our test.

```
(declare-fun a4 () Int (ite b2 (+ 1 a0) a0))
```

This is the final constraint that is formed from the two methods, and only the final constraint comparing the two states are needed.

Comparing the fields

In this case there are two fields to consider, from each *side* of the analysis. The boolean fields should be compared, and the integer fields should be compared. Again it is important to use the correct version of both fields

from each branch of the constraint generation. So given that there are two fields, there will be two constraints on the similarity of the fields. Both of these constraints can then be merged into a single constraint using the and-operator in SMT-Lib, which is true if all the constraints passed to it are true.

```
(assert (not (and (= b1 b2) (= a4 a2))))
(check-sat)
```

To make the solver determine if a model is satisfied or not, we must provide at least one constraint with a value that *must* be true in the model. We could try to use the and-constraint for this, but it would present a problem: the solver would be happy finding a single satisfying model, which would not prove that they are equal *for every set of initial values*, but only for a single initial assignment. Instead, we use the *negated* and-constraint. Now, instead of looking for a single model that proves the equality, it looks for a single model that *disproves* it. A single falsifying model is sufficient to show that the two methods do not commute. However, if no model is found, it means there are *no* counter-examples.

The call to `(check-sat)` will ask the solver to try to find a satisfying model. In this case, a model is found and we know that the two methods *do not commute.* Listing 6.1 has the entire program so far, without simplifications. The final call to `get-model` will output the model, given that a satisfying model was found.

Listing 6.1: Hardcoded SMT-Lib example

```
(declare-const a0 Int)
(declare-const b0 Bool)

;; increment ->  setB
(define-fun a1 () Int (ite b0 (+ 1 a0) a0))
(define-fun b1 () Bool (not b0))

;; setB -> increment
(define-fun b2 () Bool (not b0))
(define-fun a2 () Int (ite b2 (+ 1 a0) a0))

(assert (not (and (= a4 a2) (= b1 b2))))
(check-sat)
(get-model)
```

The actual output of sending the program to a solver can be seen in Listing 6.2 on the next page. It states that a satisfying model was found, and then gives the specific model. This model shows one possible assignment of the original variables for which the final outcome will not

Listing 6.2: Hardcoded SMT-Lib example output

```
sat
(model
  (define-fun b0 () Bool
    false)
  (define-fun a0 () Int
    0)
)
```

be the same in both executions. This model can be verified manually in a straight forward manner. When the boolean value starts as false, running `increment` first will leave `a` unaltered, but when the order is switched the boolean is made true before the increment-method is called, and `a` is incremented. Another way to consider it is that the final value of `a2` is `a0`, the final value of `a4` is `(a0 + 1)`, and that there is not way to make `a0 = a0 + 1` true, since `a0` features on both sides of the equals sign.

6.3 Generating Constraints with Haskell

The Haskell library for creating SMT-Lib code, SBV, provides a domain specific language for formalizing first-order logic and producing constraints that can be passed to the SMT solver of your choice. For any specifics in this thesis, Z3 will be used as the SMT solver.

The SBV library holds an internal representation of the constraints and variables, which is constructed step-by-step as the analysis traverses the abstract syntax tree of an ABS program. When this assembling of constraints is done, the actual SMT-Lib code is generated from this internal representation. As a result, SBV can make smart choices to optimize the generated code. This places the analysis tool presented in this section in an interesting middle-position between the ABS compiler and the SMT-Lib code generation of SBV. Everything related to producing code is abstracted away by the library, and the code written specifically for the analysis tool can focus on a more general notion of constraints.

6.3.1 Tracking the State

If one looks back to the previous section where we manually constructed the constraints for a simple program, we had to keep track of the new versions of variables by suffixing them with a new number. The higher the number, the more recently created. However, keeping track of such variables would be tedious for longer programs, and something a

computer is much better suited for than humans. To generate code for arbitrary programs, we must abstract over the names of variables. The clear choice is a map, which lets us use the names of variables as given in the program being analysed as keys. However, rather than pointing to *values* like they do in the program, they refer to symbolic variables created for the analysis.

Furthermore, the AST retrieved from the ABS compiler differentiates between `FieldUse` and `VarUse` in the AST-nodes. As a result, we do not have to worry about keeping track of scoping or shadowing, as all the necessary information to handle the difference is provided for us. However, we do have to keep track of local variable names that are the same across methods. To do this, the state passed through the program is defined in Haskell as a `Map` of `Maps`.

```
data VarTable = Map String (Map String SVar)
```

The outer maps uses *names of methods* as keys, while the inner maps uses *names of variables* pointing to symbolic variables. In the outer map there is one special key `"fields"` which points to the map holding the class fields. This means that at the end of the analysis, it is simple to find all the fields from the two different executions and compare each field with its equally named neighbour.

One additional *trick* is to prefix the method names in the outer map with a number, `1_` and `2_` for example, in order to separate the two maps when comparing a method with itself. Otherwise, creating maps when comparing a method with itself would only create a single map, as identical keys are not accepted in a map.

Since ABS is a statically typed language, all variables once defined will not be assigned a different type at any point in the program. All of these aspects of ABS are helpful when creating symbolic variables and constraints in SBV.

6.3.2 Initializing the Analysis

Every pass of the analysis over two methods must first create the map of fields, as well as a map of the parameters passed to the method. SBV has methods to create new symbolic variables constrained to the specific types in question, including `Integer`, `Bool`, and `String`. We saw two examples of this in the manual code in the previous section. To do this in SBV is equally simple:

```
si <- sInteger "x"
```

The string passed to the function, `"x"`, provides an internal name for SBV, which SBV will display when printing the model to make it clearer which symbolic variables refer to what definitions. The variable on the left is how we refer to the symbolic variable created by the `sInteger`-function provided by SBV. Having created the symbolic variable, the next step is to place it into the map over variables.

```
go (Map.insert name (Numeric si) mp) rest
```

In the above statement, the new variable `si` is inserted into the map `mp`, with `name` as its key. The new map is passed as an argument to the `go`-function which takes the map and the rest of the parameters to initialize all the parameters.

The map of class fields and method parameters is passed as the starting point of the constraint generation for both symbolic executions. Haskell, as a pure functional programming language, does not allow the mutation of its values, and any *altering* of the map does not actually alter it, but rather return a new map with the new assignments. If one were to implement this in another language with different semantic, it is important that both branches of the analysis are working with an independent copy of the initial state, where they are referring to the same initial symbolic variables.

Having found a way to store class fields and local variables, the program needs to traverse over the statements in the methods, and produce the appropriate constraints on the fields and variables. For certain statements this is straight forward, but there are enough statements of sufficient complexity to warrant extra attention. The next part will cover most of these statements.

Symbolic and Monadic Types in Haskell

Due to the static types in Haskell, the SBV specific types are wrapped into a type so that the map that contains the state can hold different symbolic types at the same time. We know that the types will match since we get the program after it has been type-checked by the ABS compiler, but Haskell does not know this, and so we have to be somewhat extra accommodating. This means that at the leaves values are unwrapped for use and re-wrapped for storage in the state. This is visible in the code example above where `Numeric` is wrapped around the symbolic integer before inserting it into the map. This is only necessary in order for Haskell to type check and be happy about holding different types of symbolic variables in the same data structure.

In the Haskell examples to come, there will be some strange syntax for

those unfamiliar with the language. Here are some resources to familiarize oneself with the language.[1,2]

The `state`-monadic pattern is used extensively to avoid having to pass around the map of variables explicitly. Instead, it is passed *implicitly*, and it is possible to realize the current state into a variable with `get`, and to set the new state with `put`. Both these functions will be frequently used in the examples. The benefit of using monads in this case is that our code is free of side effects. Instead, every function is pure which makes it easy to reason about what our code does.

Because we are using multiple monads inside each other to get both state as one effect, and the possibility of throwing errors as another, we have to use the keyword `lift` in order to access the *inner* monad. The inner monad in these examples is `State`, which is why its methods `get` and `put` are *lifted*.

6.3.3 Assignment

The first case is an assignment, into a variable `a`, and where `a` also features on the right side.

```
a = a + 1
```

In terms of the abstract syntax tree, this is an `Assignment Statement`, with a `BinaryExpression` on the right side.

Assignments

The syntax tree differentiates between assignments and declarations, but they are fairly similarly implemented since they do pretty much the same thing. First, look at the expression on the right side of the assignment, determine what constraint it corresponds to. Second, find the correct variable in the map of variables, and *reassign* that variable in the map. Here is the Haskell code for doing this with SBV.

```
(AssignStmt (PureExp (FieldUse v)) exp) -> do
  ret <- generateExp (name, exp)
  mp <- lift get
  let fs = mp Map.! "fields"
  lift $ put $ Map.insert "fields" (Map.insert v ret fs) mp
```

[1] http://learnyouahaskell.com/chapters
[2] https://haskell.mooc.fi/

It is important to note that *reassign* in this context means to return a new map with the `a` pointing to a new constraint and the other variables the same as the previous map.

This is part of a larger function that looks at all the possible statements in the program, and this particular code pattern matches on an `AssignStmt`, with a `FieldUse` as the value being assigned to. This means that this particular variable refers to a *class field*, which are the variables we really care about. The `generateExp` function call is responsible for creating the constraints on the right side of the assignment, analysing the `pure expression` on the right hand side. The function ends with `put`-ting the updated map as the new state of the analysis.

The call to `generateExp` will pattern match on the following code:

```
BinaryExp op l r -> do
  l' <- generatePureExp (name, l)
  r' <- generatePureExp (name, r)
  pure $ case (l', r') of
    (Boolean a, Boolean b) ->
      case op of
        (...)
    (Numeric x, Numeric y) ->
      case op of
        (...)
        Addition -> Numeric $ x + y
        Subtraction -> Numeric $ x - y
        Multiplication -> Numeric $ x * y
        (...)
```

Parts of the function have been omitted for the sake of clarity, as there are more operators following a similar pattern.

Here we see one instance of the power of the SMT solvers, as we are able to use the already defined operators in the theory of arithmetic to directly encode the addition of two variables as constraints. It is this resulting constraint of adding two symbolic variables that is returned to the previous functions, and set as the new value of `a` in this case. The further calls to `generatePureExp` at the top of the function are responsible for fetching the left and right side of the binary expression. The left side of our current expression is the `a`, and requires a lookup in the map. It will pattern match on the following line, where `gets` is a specialization of `get` which only returns a part of the state. In this case, the symbolic constraint the current version of `a` refers to.

```
(FieldUse s) -> gets ((Map.! s) . (Map.! "fields"))
```

For the right-hand side of the binary expression, the literal `1` returns a

symbolic constant `1`. Once the program has fetched both sides of the arithmetic expression, it matches on the `Addition` operator. The `+`-operator has been overloaded by SBV to work on symbolic integers, and to returns a new symbolic constraint. This final symbolic expression is then returned and eventually placed as the new constraint on `a` in the map.

Pure expressions are *pure* because they depend only on what is passed to them as arguments. This makes them by far the most simple language construct to model, and so modeling them becomes fairly accurate. When we cover constructs with side effects it becomes much more difficult.

6.3.4 Branching and Iteration

There are two forms of branching in our language, as part of a two-way branch in the `if`-statement, and a loop in the `while`-statement. In either case, the solver cannot determine which branch to *execute* during its run, and require both (or all) branches to be fully executed, and it will pick the correct version of the state by looking at the test as part of the symbolic execution.

```
(IfStmt co th (Just el)) -> do
  cond' <- generatePureExp (name, co)
  let Boolean cond = cond'
  preIf <- lift get
  traverse_ (generateStatement . (name,)) th
  postThen <- lift get
  lift $ put preIf
  traverse_ (generateStatement . (name,)) el
  postElse <- lift get
  lift $ put $ Map.unionWith
               (Map.unionWith (merge cond))
                              postThen postElse
```

The intuition for how branching is implemented is this:

1. Save the state before the `if`-test (into `preIf`), and get the current constraints used for the condition (into `cond`).
2. Run the first branch (first `traverse_`), and save the resulting state (into `postIf`).
3. Reinstate the state from before the first branch (using `lift $ put preIf`), and run the second branch (second `traverse_`), saving the resulting state (into `postElse`).
4. Use the condition to pick the correct state for the continuation of the program. The merge function calls `ite` for every variable in the map.

Just like a normal program has to generate code for both branches of execution, we have to generate constraints for both branches. And for normal programs this means that the condition can pick one of the branches to execute, leaving the other as *dead code* in the sense that it does not run. In the case of symbolic execution, it means that the constraints created for the branch that is not chosen can be of any value. They will not have any impact on the final constraints.

This idea of saving the state and reinstating it is presented for a path that branches in only two possible paths. However, the underlying language does not support more than two branches, and this approach is sufficient. A nested `if`-statement is easy to model, and will naturally happen as the recursive calls on each statement in the body of the `if`-statement are made.

A `while`-loop can be modeled as an `if`-test where the alternative branch does nothing. To simulate running the loop multiple times the first branch is branched again using another `if`-test, and so on, nesting `if`-statements inside each other. Since it is not possible to consider every possible continuation in the case of the `while`-loop, it is necessary to place an upper bound on how many times the loop is allowed to create a branch. This also runs into potential problems when the loop has a large body of statements, but the number of constraints grow only linearly in size with each additional branching. This is due to every branching having to create the body as one branch, with the other branch passing without doing anything.

Just as in a normal program, once the condition is false, it will stay false for the rest of the symbolic execution of that `while`-statement. Even if the body of the loop has to be symbolically executed additional times, the same state will always be chosen and thus the condition will not be changed.

For a discussion on what happens when the last state produced by the `while`-loop is chosen, because loop was ended because of the bound and not the test being false, see Section 5.5.4 on page 41.

6.3.5 Effectul Statements and Expressions

There are some statements that are less straight forward to implement. This is mostly due to how they cause side-effects, or cause execution in the normal program that cannot be modelled symbolically. Some of these constructs can nevertheless be simulated in our analysis to some degree, while others will cause it to stop and return that no answer can be determined. This latter statement means that nothing specific can be said about the commutativity of the two methods. By not saying anything

about the methods, they should be treated as non-commuting.

Several effectful statements in ABS cause *scheduling points*, wherein the method suspends itself and some other block of code in the same class is scheduled. As a result, the method is no longer atomic. For a discussion on this, see Section 7.1.1.

A `SyncCall` is a statement in which an asynchronous call is made and then immediately awaited upon, resulting in a scheduling point for the callee.

```
(SyncCall obj method params) ->
  throwError $ UnsupportedType
    "generateEffExp: SyncCall not supported"
```

Since this marks an end of the constraint generation for this particular analysis, the analysis will *throw an exception*. Strictly speaking, no exceptions are thrown, but the practical difference is small. The underlying Haskell machinery passes an error value through the rest of the computation, which can then be checked at the end of the construction and before the actual symbolic execution.

The specific scheduling points to cover are listed in the documentation.[3]

However, there are some effectful expressions that do not trigger a scheduling point, and these statements can in fact be modeled, albeit with some caveats.

New Expressions

The `New`-expression creates a new object, and returns a reference to this new object. However, while creating new objects multiple times might be pointless, it does commute. There is nothing about creating an object multiple times that affects the state of the current object. Either way, the class has a reference to a fresh object.

`New`-expressions are handled by testing the inputs to the call. If they are the same in every case, they will be considered commuting. If swapping execution order changes what arguments are passed to the initialization, the calls will be considered not commuting. This is done by creating a new unique field in the `Fields`-map, which will then be tested at the end of the symbolic execution by comparing the arguments passed to both calls.

[3] https://abs-models.org/manual/#await-stmt

Asynchronous Call Expressions

Any asynchronous call cannot, by definition, alter any state on the callee object and is therefore trivial to deal with by themselves. They do return a `Future`, which will contain some value that can be accessed by waiting for its resolution in the called object. If this waiting is done by suspending, the method will not be analysed. However, if the method uses a `get`-call to wait, this will instead block the current thread until the value is available, and atomicity is retained.

Since nothing is known about the resolved value from a future, given that it came from another method, is it modeled as a free variable. It is important that this free variable is the same in both branches of the symbolic execution, or it would be trivial for the solver to assign them different values and probably find a counter-example. Furthermore, the arguments passed to an asynchronous call are tested in both branches to ensure they are the same in both instances. Different asynchronous calls would case different state in another object, and probably different return values into the future.

Await and Suspend

Any statement that makes a method non-atomic is not supported by this analysis. See discussion in Section 7.1.1. As a result, all such statements and expressions will cause the program to propagate an error up the stack, informing the user of why commutativity cannot be determined for this particular pair of methods. Furthermore, once the analysis has determined that a pair of methods cannot be analysed, the analysis cannot fall back on a read-write analysis, since this analysis also relies on the atomicity of the methods.

Get Expressions

`Get`-expressions cause the current thread to block while waiting for the value of a future to resolve. This does not cause a scheduling point, and are perfectly fine to analyse. In fact, there is nothing *to* analyse, as the expressions just blocks until the value is ready, causing no effects on the current state. It does alter a `Future<a>` into just an `a`, but from the perspective of the analysis this is not very important, unless it forms part of a larger statement.

6.4 Output from SBV

Listing 6.3: Code generated by SBV passed to Z3 with response for the `increment` and `setB` methods

```
[GOOD] ; Automatically generated by SBV. Do not edit.
[GOOD] (set-option :print-success true)
[GOOD] (set-option :global-declarations true)
[GOOD] (set-option :smtlib2_compliant true)
[GOOD] (set-option :diagnostic-output-channel "stdout")
[GOOD] (set-option :produce-models true)
[GOOD] (set-logic ALL) ; has unbounded values, using catch-all.
[GOOD] ; --- uninterpreted sorts ---
[GOOD] ; --- tuples ---
[GOOD] ; --- sums ---
[GOOD] ; --- literal constants ---
[GOOD] (define-fun s2 () Int 1)
[GOOD] ; --- skolem constants ---
[GOOD] (declare-fun s0 () Int) ; tracks user variable "a"
[GOOD] (declare-fun s1 () Bool) ; tracks user variable "b"
[GOOD] ; --- constant tables ---
[GOOD] ; --- skolemized tables ---
[GOOD] ; --- arrays ---
[GOOD] ; --- uninterpreted constants ---
[GOOD] ; --- user given axioms ---
[GOOD] ; --- formula ---
[GOOD] (define-fun s3 () Int (+ s0 s2))
[GOOD] (define-fun s4 () Int (ite s1 s3 s0))
[GOOD] (define-fun s5 () Bool (not s1))
[GOOD] (define-fun s6 () Int (ite s5 s3 s0))
[GOOD] (define-fun s7 () Bool (= s4 s6))
[GOOD] (assert (not s7))
[SEND] (check-sat)
[RECV] sat
[SEND] (get-value (s0))
[RECV] ((s0 0))
[SEND] (get-value (s1))
[RECV] ((s1 false))
*** Solver   : Z3
*** Exit code: ExitSuccess
```

Listing 6.3 shows the code passed to the solver, along with the response from the solver at the end. The first few lines prints some of the options that are set for the current run, and then the sections for various solver constructs that are not used in this particular problem.

In the `literal constants`-section, we can see the 1 from the right hand side of the addition, `a + 1`. Then, the free variables, here named skolem constants, and the name of the variable they correspond to in a comment. This is determined by the name passed in as an argument to

the `sInteger` and `sBool`-functions. Then, in the *formula*-section are the actual constraints placed on the variables, that was created as the analysis traversed the abstract syntax tree.

After all the constraints have been created, `check-sat` is called. The response from the solver is `sat`, followed by the actual counter-example.

It is interesting to compare the constraints to those that were handmade in Section 6.2 on page 45. As a human it can be more natural to inline various constructs like each side of the `ite`-statements. SBV does not do this, instead choosing to give every atomic statement its own variable. Another feature of SBV is that it does not create duplicate constraints. In the example coded by hand there are two versions of the negated boolean. In part this was to be explicit about how information flows through a program when it is created by hand, and it would have been fairly simple to also code by hand without duplicating. However, for more advanced constraints, and in situations with many more constraints than in this example, it is good to know that SBV will make such optimizations on its own.

By having more constraints depend on the same variable, the solver will be faster at finding counter-examples. Rather than having to backtrack all the way to the beginning, it might have sufficient information to determine whether a set of constraints is unsatisfiable at an earlier stage.

The strict use of Static Single Assignment for *every* constraint is probably the most useful takeaway from printing the output of SBV, as doing so can help speed up the solver on longer inputs.

Chapter 7

Evaluation and Conclusion

This chapter aims to evaluate the success of the analysis tool, by looking at its ability to identify commuting methods compared with the more naive read-write analysis. There will also be a discussion around the ergonomics of using an SMT solver to generate constraints on top of an abstract syntax tree.

All the examples are from the github repository for the ABS language[1].

The results from running the analysis and considering the number of commuting methods found by the SMT solver and by the read-write analysis suggests the following:

1. The solver is able to determine commutativity to some degree, finding 132 pairs of commuting methods, 21 non-commuting methods, and not being able to say anything useful about 87 methods.

2. The read-write analysis is really good at determining commutativity, and doubly so when considering how simple it is. From the 153 pairs for which the solver gave a definitive answer, the read-write analysis found 130 commuting pairs. However, out of the 87 inconclusive solver results, it finds another 70 commuting pairs, for a total of 200 commuting pairs.

Perhaps the most surprising part about the results is how many of the pairs actually commute.

Given the large amount of commuting pairs, the fact that the read-write analysis found so many could suggest that a commutativity analysis was not the best property to test for. The existence of a naive and sound analysis, which is very easy to implement, means that the more complex analysis have to be *very* good in order to be a better option. Properties that

[1] https://github.com/abstools/absexamples

lack a naive option could present a better alternative use of the power of an SMT solver.

Table 7.1: Results of Analysis on All Example Programs

Analysis Type	Commuting	Non-Commuting	Inconclusive
SMT Solver	132	21	87
Read-Write	200	40	0

Table 7.2: Results of Analysis On Conclusive SMT results

Analysis Type	Commuting	Non-Commuting
SMT Solver	132	21
Read-Write	130	23

To stay on the safe side, the current version of the analysis will consider any loop that runs beyond the upper bound to be non-commuting, regardless of content. However, the read-write does not even analyse any actual looping, as it looks only at individual statements. This causes the methods with `while`-loops to be deemed non-commuting by the solver, only because the bound is hit. See the discussion in Section 5.5.4 on the alternative of letting the user decide what to do when hitting the bound, for a possible way to have the solver recognize these as commuting.

The support for built-in operations on numbers, strings, and booleans was expected to be more useful than it turned out to be in the examples. Given how easy it is to construct examples using these features that are deemed commuting by the solver, but not by the read-write analysis, it is somewhat surprising how few there are in the gathered examples. It is possible that the examples being created specifically to show off various features of ABS, use these features heavily, and that they are not very representative of all programs one might want to create.

7.1 Implementation of SMT analysis

We have created a prototype implementation of an analysis that can determine the commutativity of methods in ABS. These methods are restricted to a non-trivial subset of common programming operations. The implementation of the analysis tool in Haskell has provided valuable experience generating code for an SMT solver.

For the constructs that are supported, such as arithmetic and strings, the implementation is very straight forward, often being nothing more than a call to the corresponding function in the solver.

7.1.1 Support for ABS Features and Possible Future Work

Table 7.3: Overview of ABS features covered by the Analysis Tool

Feature	Level of Support
Integers	Supported
Booleans	Supported
String	Supported
Expressions	Supported
Assignment	Supported
Synchronous/Asynchronous Calls	Supported
Get Expressions	Supported
New expressions	Rudimentary Support
Functions	Rudimentary Support
Custom/Compound Data Types	Not supported
Await/Suspend	Not Supported

A few more ABS features are not covered because they did not appear in any of the examples. This includes `foreach`-statements, `throw`-statements, `assert`-statements, `switch`-statements, and `try-catch`-statements. The lack of these constructs in the example programs might be an indicator that the sample was not as representative as it could be. However, many of them can be expressed in using different syntax, and it might reflect programmer preference or that certain ways to write ABS is more natural.

Functions

There is potential for substantially more support for ABS functions. Since functions are pure, they could be supported to a similar degree as expressions. Currently, the return value from functions is completely free, and the solver ensures that function calls are made with the same arguments.

Custom Data Types

As custom data types are not supported, any program heavily using them will not be very well analysed.

In fact, without support for custom data types, any class with custom data types as fields will simply not be analysed at all, unless we limit the analysis to methods that do not read or write to those fields. Responsible for most of the 87 inconclusive results, custom and compound data types

provide an opportunity for future work. There is rudimentary support for algebraic data types in Z3, but to *generate* data types was not something we were able to implement.

Delineate Atomic Blocks by Scheduling Points

This analysis has only considered methods for which the entirety of the method is one atomic block. Any *scheduling point* inside the method body, such as suspend statements or await statements will force the analysis to stop and deem the current pair as `Unknown` in terms of commutativity. One possible way to improve our analysis is to more precisely delineate atomic blocks by splitting up methods on internal scheduling points.

Some challenges to consider:

- This would require the analysis to name each of these blocks in a uniform manner to ensure that the analysis refers to the same blocks of code every time.
- Methods split into multiple blocks would need some way of transferring what is known about variables from the previous elements in the method. A block that starts in the middle of a method should have get the knowledge about constraints on local variables placed by previous blocks in the method. This could be particularly difficult when the block is split inside a loop, and would seem to require that the methods be analysed not separately as in our analysis, but in some sort of ordering that allows the transfer of local constraints.
- Blocks that start inside a loop can have have different constraints placed on local variables depending on how many iterations of the loop have been executed so far. It might be necessary to analyse the same block multiple times with different starting constraints. If there are enough blocks of this nature, it could dramatically increase the complexity of the analysis.

One feasible solution for splitting methods on scheduling points is to make a syntactic transformation of the syntax tree, and split methods into multiple methods, and then have the local variables from the earlier block in the method passed as arguments to the next block.

For example, a method `m` can be split into methods `m#1` and `m#2`, with the local variables in `m#1` passed to `m#2` as arguments. If this transformation happens before the entry point of the tool presented in this thesis, it should work without any modification. However, doing this for loops seems to create a similar problem as symbolically executing loops, and probably requires some upper bound on the methods created.

7.2 Conclusion

The main objective of this thesis was to:

> *Investigate the suitability of SMT solving to determine the commutativity of methods in an active object language, such as ABS.*

We are completely convinced about the power of SMT solvers as a tool for analysing programming languages. Its suitability for doing commutativity analysis on its own is solid, but questionable in the presence of the simpler read-write analysis. If used for problems better suited for the strengths of the solver, it should provide a much better experience and more impressive results.

Furthermore, a better understanding of the tools or more time available for implementation could provide a better result, given the inherent complexity.

The shortcomings of the tool itself relate in part to the difficulty of translating an entire language into symbolic constraints from scratch, and the complexity of the tools involved, and should not be underestimated by someone who decides to take it on.

As solvers become better, analyses of this type should become even more attractive, and further work on different program properties is encouraged.

Bibliography

[1] Elvira Albert et al. "Formal modeling and analysis of resource management for cloud architectures: an industrial case study using Real-Time ABS". In: *Serv. Oriented Comput. Appl.* 8.4 (2014), pp. 323–339. DOI: 10.1007/s11761-013-0148-0. URL: https://doi.org/10.1007/s11761-013-0148-0.

[2] Bowen Alpern and Fred B. Schneider. "Recognizing Safety and Liveness". In: *Distributed Comput.* 2.3 (1987), pp. 117–126. DOI: 10.1007/BF01782772. URL: https://doi.org/10.1007/BF01782772.

[3] Roger Antonsen. *Logiske Metoder: Kunsten å tenke abstrakt*. Oslo: Universitetsforlaget, 2014. ISBN: 978-82-150-2274-1.

[4] Edward A. Ashcroft. "Proving Assertions about Parallel Programs". In: *J. Comput. Syst. Sci.* 10.1 (1975), pp. 110–135. DOI: 10.1016/S0022-0000(75)80018-3. URL: https://doi.org/10.1016/S0022-0000(75)80018-3.

[5] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (2018).

[6] Clark W. Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8%5C_11.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017.

[8] Nikolaos Bezirgiannis et al. "Implementing SOS with Active Objects: A Case Study of a Multicore Memory System". In: *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Reiner Hähnle and Wil M. P. van der Aalst. Vol. 11424. Lecture Notes in Computer Science. Springer, 2019, pp. 332–350. DOI: 10.1007/978-3-030-16722-6_20. URL: https://doi.org/10.1007/978-3-030-16722-6%5C_20.

[9] Miquel Bofill, Josep Suy, and Mateu Villaret. "A System for Solving Constraint Satisfaction Problems with SMT". In: *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Ofer Strichman and Stefan Szeider. Vol. 6175. Lecture Notes in Computer Science. Springer, 2010, pp. 300–305. DOI: 10.1007/978-3-642-14186-7_25. URL: https://doi.org/10.1007/978-3-642-14186-7%5C_25.

[10] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[11] L Erkök. "SBV: SMT based verification in Haskell". In: *Software library* (2019).

[12] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Vol. 1032. Lecture Notes in Computer Science. Springer, 1996. ISBN: 3-540-60761-7. DOI: 10.1007/3-540-60761-7. URL: https://doi.org/10.1007/3-540-60761-7.

[13] Einar Broch Johnsen et al. "ABS: A Core Language for Abstract Behavioral Specification". In: *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*. Ed. by Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue. Vol. 6957. Lecture Notes in Computer Science. Springer, 2010, pp. 142–164. DOI: 10.1007/978-3-642-25271-6_8. URL: https://doi.org/10.1007/978-3-642-25271-6%5C_8.

[14] Eduard Kamburjan, Reiner Hähnle, and Sebastian Schön. "Formal modeling and analysis of railway operations with active objects". In: *Sci. Comput. Program.* 166 (2018), pp. 167–193. DOI: 10.1016/j.scico.2018.07.001. URL: https://doi.org/10.1016/j.scico.2018.07.001.

[15] Daniel Kroening and Ofer Strichman. "Equality logic and uninterpreted functions". In: *Decision Procedures*. Springer, 2008, pp. 59–80.

[16] Leslie Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Trans. Software Eng.* 3.2 (1977), pp. 125–143. DOI: 10.1109/TSE.1977.229904. URL: https://doi.org/10.1109/TSE.1977.229904.

[17] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008.*

Proceedings. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24.

[18] Vugranam C Sreedhar et al. "Translating out of static single assignment form". In: *International Static Analysis Symposium*. Springer. 1999, pp. 194–210.

[19] Brian A. Wichmann et al. "Industrial perspective on static analysis". In: *Softw. Eng. J.* 10.2 (1995), pp. 69–75. DOI: 10.1049/sej.1995.0010. URL: https://doi.org/10.1049/sej.1995.0010.