universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„New Symbolic Execution Pass for the CASM Language“

verfasst von / submitted by

Jakob Moosbrugger, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Univ.-Prof. Dr. Uwe Zdun

Acknowledgements

First, I would like to thank my supervisor Uwe Zdun for his support and the trust he put into me during this thesis. I would also like to thank Philipp Paulweber for his valuable input on the CASM language and supporting infrastructure. Furthermore, I am gratefule for the support and feedback at each step of the thesis.

To conclude, I cannot forget to thank my friends and family for their emotional and motivational support during the writing process of this thesis. Last but not least I want to thank my girlfriend for her support and motivation, when I had none left.

Abstract

Nowadays it is important that software works correctly as it is often used in safety-critical systems. One method for describing systems on a high level are Abstract State Machines (ASM). ASMs are an extension of Finite State Machines and are formally defined. As they are based on mathematical foundations, it is possible to prove certain properties of an ASM using formal methods. However, not only the modeled systems need to be error free but also compilers which are used to translate the high level representation to an executable file also must not introduce any errors. Translation Validation is one method to prove that a specific compilation did not introduce any errors. For translation validation a mathematical model of the input and the output of the compilation is built and then proven that they are identical.

This thesis takes the first step towards translation validation in the Corinthian Abstract State Machine (CASM) compiler, by presenting a framework that transforms an ASM written in the CASM language into a representation in the TPTP language. It is based on a previously introduced approach by Lezuo [48]. The TPTP language is designed as input for Automated Theorem Prover (ATP) systems. The transformation supports concolic execution. In order to ensure that numeric functions are not updated with symbolic values, this thesis presents a new system which checks whether such updates occur before the actual transformation takes place. If such updates occur the system promotes numeric functions to symbolic functions accordingly. Furthermore, a transformation from the TPTP model directly to the model of the Satisfiablity Modulus Theories (SMT) solver Z3 is presented.

Kurzfassung

Heutzutage ist es wichtig, dass Software korrekt arbeitet, da sie oft auch in sicherheitskritischen Bereichen verwendet wird. Eine Methode zur Beschreibung von Software sind Abstract State Machines (ASM). ASMs sind eine Erweiterung von Finite State Machines (FSM) und sind formal definiert. Da sie auf mathematischen Grundlagen basieren, ist es möglich, bestimmte Eigenschaften mittels formalen Methoden zu beweisen. Es müssen jedoch nicht nur die modellierten Systeme fehlerfrei sein, sondern auch die Compiler, welche verwendet werden, um die abstrakte Repräsentation in eine ausführbare Datei zu übersetzen. Diese dürfen keine Fehler einbringen. Translation Validation ist eine Möglichkeit zu zeigen, dass ein bestimmter Compilier-Vorgang fehlerfrei ist. Für Translation Validation wird ein mathematisches Modell der Input Spezifikation und des Outputs erstellt und gezeigt, dass die Modelle ident sind.

Diese Arbeit ist der erste Schritt in Richtung Translation Validation im Corinthian Abstract State Machine (CASM) Compiler, indem es ein Framework präsentiert, welches eine in CASM geschriebene ASM in ihre Repräsentation in der TPTP Sprache übersetzt. Die Arbeit basiert auf einem von Lezuo [48] vorgestellten Ansatz. Die TPTP Sprache ist als Input für Automated Theorem Prover (ATP) Systeme entworfen. Die Transformation unterstützt concolic execution. Um sicherzustellen, dass dabei numerische Funktionen nicht symbolischen Werten zugewiesen werden, wird in dieser Arbeit ein neues System vorgestellt, welches vor der Transformation das Auftreten solcher Zuweisungen prüft. Falls derartige Zuweisungen existieren, wird die entsprechende Funktion von numerisch zu symbolisch umgewandelt. Außerdem wird eine direkte Transformation von TPTP zu einem Model für den Satisfiablity Modulus Theories (SMT) Solver Z3 vorgestellt.

Contents

List of Tables

List of Figures

Listings

1. Introduction

Today's software is used in many different places for a variety of reasons and purposes. This includes safety critical systems where it is important that the software works exactly as specified. One method for writing software which conforms to a specification is to use Abstract State Machines (ASM). ASMs were first introduced by Gurevich [42] in the Lipari Guide. They are a generalization of finite state machines and aim to define systems on a high level [18]. The idea is that a system is specified using this formalized high level method and a computer can either directly execute that ASM or compile it to a binary. As ASMs are formalized, there has been a lot of effort put into model checking, meaning, to mathematically prove some properties of given ASMs [5, 2, 45, 58, 61, 74, 3, 33]. Automated Theorem Proving (ATP) is employed for that check, as it cannot be performed manually for any arbitrary ASM. The properties which are of interest are very diverse and range from deadlock freedom [14] to liveliness to general assertions [15] which must be true during some execution of the ASM.

However, even if the specification of a system and its implementation in any high level programming language is error free the resulting binary can be erroneous. This errors can be introduced by the compilation process, as the compilers itself contain errors [75, 76]. To prevent that, the compiler itself must be fully verified to be correct. This is a lot of work and each change in the compiler triggers a full re-verification of the whole compiler. Alternatively, translation validation can be used [60]. With translation validation, instead of proving that the compiler is error free, it is proven that a specific compilation is correct, i.e. the compilation process did not introduce any errors. This is done by building a model of the input program, then compiling it and building a model of the output binary. If the two models are semantically identical, the compilation is error free.

This thesis works towards translation validation for the Corinthian Abstract State Machine (CASM) language [1]. CASM is an implementation of an ASM. It has an interpreter and an optimizing compiler, which produces efficient binaries [50]. The compiler is written in C++. For more details on CASM see Section 2.1.

[1] `https://casm-lang.org`

1.1. Motivation

Model checking and translation validation are a desired addition to the CASM language and toolkit. However, the current implementation [48] for the transformation from CASM to a trace in TPTP must be improved. It does not work with the current CASM language, as it is not included in the state-of-the-art compiler and therefore cannot transform the current Abstract Syntax Tree (AST) or Intermediate Representation (IR).

1.2. Problem Statement

For this thesis a framework is implemented, which must be able to transform the AST of a CASM specification of the state-of-the-art compiler into a TPTP trace. The framework should produce structurally equivalent traces as the previous implementation by Lezuo [48]. Supported CASM language constructs should include sequential and block rules, conditional rules, let rules and call rules. Furthermore, a direct in memory translation to a SMT solver model of the trace should be possible.

A correct TPTP trace should conform to the following specification: The whole trace consists of First Order Formulae. ASM functions are represented as uninterpreted predicates and ASM instructions are predicates. A concolic transformation must be supported in which some ASM functions are interpreted as numeric, others as symbolic. Only one trace should be generated which differs from the implementation by Lezuo. See Section 6.4 for more information on the previous implementation.

1.3. Methological Approach

In order to support symbolic execution, a framework is created which can represent any valid TPTP model. This TPTP framework has to be able to parse and print these TPTP models. Second, a transformation from the TPTP framework to the Satisfiablity Modulus Theories (SMT) backend is built. Third, a transformation from the CASM-AST to the TPTP framework is built. This step follows the specification by Lezuo [48]. Lastly, a validation is added for symbolic consistency is added. Each step is evaluated and validated using unit tests.

1.4. Structure of the Thesis

The thesis at hand is structured as follows: Chapter 2 gives an overview of abstract state machines and the technologies used within this thesis. Chapter 3 describes the framework which was built on a high level. First, it discusses the overall architecture,

then it describes the transformation from the CASM language to the TPTP language and lastly it describes the problem of symbolic consistency and how it is handled within the framework. Chapter 4 presents implementation details of the framework. Chapter 5 presents an empirical evaluation of the framework. Chapter 6 discusses related work. First in the area of languages for model checkers and model checkers itself and afterwards of symbolic execution implementations, with a focus on symbolic execution of ASMs. Chapter 7 summarizes the thesis and outlines future research topics.

2. Background

This chapter briefly describes the technologies on which this thesis is based on including Abstract State Machines (ASM) in general and the Corinthian Abstract State Machine (CASM). The subsequent chapter gives a short introduction to symbolic execution as well as the Thousands of Problems for Theorems Provers (TPTP) library. These parts which are used for the present thesis will be described in more detail.

2.1. Abstract State Machines

Abstract State Machines were first introduced by Gurevich in 1995 [42] calling it "evolving algebra". ASMs can be used to model any program in terms of states and transition rules, similar to Finite State Machines (FSM). However, ASMs are strictly more expressive than FSMs. The initial definition of an ASM by Gurevich is almost identical to the definition of Börger 2003, who defines ASMs as:

> "An *abstract state machine* M consists of a signature Σ [...], a set of initial states for Σ, a set of rule declarations, and a distinguished rule name of arity zero called the *main rule name* of the machine. In a given state, a transition rule of an ASM produces for each variable assignment an update set." [18, p. 72]

In order to fully define an ASM, a signature, transition rules and an initial rule must be defined.

Signature. In this context, a signature is a finite collection of named n-ary functions, where n is a non-negativ natural number. The arity of a function is the number of arguments it has. Börger further distinguishes between *static* and *dynamic* functions. Static functions do not change during the lifetime of the ASM, whereas dynamic functions may change due to transition rules. The special case of a null-ary function can be compared to a constant and a variable in traditional programming languages, for the static and dynamic case respectively.

An exemplary signature Σ_{Colors} for an ASM for managing colors on a pallete may include the static constants `"Red"`, `"Blue"` and `"Green"` as available colors. The same

signature could also include a dynamic function `"colorOnBrush"`, which can be equal to `"Red"` at one point during the execution of the ASM and be equal to `"Green"` at another. The signature also includes a static binary function `"darker(c1, c2)"` as a means to get the darker of two colors. Another function in the signature is the `"usedLess(c1, c2)"` function, which should return the color which is used less. As this might change during the run of the ASM, it is a dynamic function.

State. The state of an ASM defines the interpretation of the function names in its signature Σ. A state of each ASM implicitly defines three values *true*, *false* and *undef* as pairwise distinct values [42, 18]. The values *true* and *false* are used for relation functions. Gurevich defines relation functions as functions which only ever return *true* or *false* [42]. This is in contrast to Börger, who allows relation functions to be partial and therefore also return the value *undef* [18]. Relation functions are often used in combination with conditional rules. *undef* is the default value for all functions in the ASM state, which are not explicitly defined otherwise.

Static functions are identical in all subsequent states, whereas dynamic functions may be different in later states.

In the context of the color example, the initial state of the ASM defines the functions `"Red"`, `"Blue"` and `"Green"` as pairwise distinct for the entire duration of the ASM execution. The function `"darker"` is defined to return the darker color for each possible pair of inputs using some arbitrary metric.

For the functions `"colorOnBrush"` and `"usedLess"` for all inputs, the value *undef* is assigned, as the brush is initially clean and no colors have been used so far. During the execution however, the values of these functions change as the state changes.

Rules. In order to transition between states, rules are used. Gurevich [42] defined a number of rules such as an update rule, a conditional rule, a block rule, an import rule, a choose rule and a forall rule. Börger [19] adds a skip rule, a let rule, a call rule, an iterate rule and a sequential rule. The semantics of these rules is depicted in Table 2.1. More complex rules are created by composition using the block and sequential rules.

Update-Set. As mentioned before, update rules produce updates to locations. A location l is a tuple of a function name f and the argument tuple $(a_1, \ldots, a_n)$ which has the same length as the arity of the function $l = (f, (a_1, \ldots, a_n))$. Hence, an update is a tuple of a location and a value to which the location is updated (l, v) [18].

Due to composition, it is possible for update rules to be only a part in a transition rule. In this case updates are not immediately applied to the state, instead, they are collected

Rule	Syntax (as used in Börger 2003 [18])	Meaning
skip	**skip**	Do nothing rule.
update	$f(a_1, \ldots, a_n) := l$	Update function f at $(a_1, \ldots, a_n)$ with the value l.
conditional	**if** φ **then** P **else** Q	If φ is *true*, execute rule P otherwise rule Q.
block	P **par** Q	Execute P and Q with parallel semantics, so updates in P don't influence updates in Q.
import	**import** v **do** P	Create new symbol v in Σ and execute P.
choose	**choose** x **with** φ **do** P	Choose x, which satisfies condition φ and execute P.
forall	**forall** x **with** φ **do** P	For all x satisfying condition φ execute P in parallel.
let	**let** $x = t$ **in** P	Assign the term t to x and execute P.
call	$r(a_1, \ldots, a_n)$	Call rule r with arguments $(a_1, \ldots, a_n)$.
iterate	**iterate**(R)	Execute R until its update set is empty.
sequential	P **seq** Q	Execute P which produces an intermediate state, then execute Q in that intermediate state.

Table 2.1.: Semantics of Rules

in an update-set. After executing the transitional rule, the update-set is applied to the state, creating the new state of the ASM.

If the same location is updated twice in a rule, the update set is inconsistent [42]. In the case of an inconsistent update-set, the state must not be changed. However, neither Gurevich nor Börger [18] define how to proceed when said case is encountered apart from not updating the state. They only advise to give some feedback to the programmer, either by a special function or by an error message.

There is an exception to the collection of updates in an update-set, however, when it comes to sequential rules. In sequential rules each update rule immediately creates a new intermediate state which is used for the following rule in the sequence rule.

Agent. An agent is a computation entity, comparable to threads from traditional programming [18]. Gurevich defines agents as entities which are able to move an algorithm from a state S_0 to another state S_1, S_2 etc. [42]. In a single agent ASM, the agent executes a rule sequentially and moves the states linearly forward (S_0, S_1, S_2, ...). This is done by executing the given transition rule, which produces the update-set, to change the state appropriately. In Börgers definition of ASMs, the initial transition rule for an agent is

set with the *main rule name* [18]. An ASM may contain multiple agents, which operate on different views of the state, so that not every agent sees the same state as another. Different agents can also execute different rules.

Corinthian Abstract State Machine

The Corinthian Abstract State Machine (CASM) is an implementation of an ASM. In the early stages, the CASM language was heavily influenced by CoreASM [32], as it tried to improve on the performance of CoreASM [49]. However, over time the languages diverged.

Functions and Types. CASM is a strong statically inferred typed language. The function type cannot change during execution. In the terms of Börger, the function definitions in CASM include both the signature of the ASM and the initial state. Each function can be of arbitrary arity and each parameter and return value can be any available type.

Concerning possible function types, CASM provides the common types of programming languages and their sub-types as built-in types and allows for custom enumeration types. The standard types include `Boolean`, `String`, `Integer` and `Decimal` for floating point values. Furthermore, it includes a `Rational` type. All numeric types can be subtyped to have only valid values in a given range. Enumeration types can be defined in the specification. The values of those types are different form all other types and from each other. Enumeration types do not define further properties such as ordering or the like. However, all types contain the `undef` value [54].

In future versions of CASM, traits will be support as a means of specifying composite types and behaviors. With traits, CASM enables the use of high level object-oriented abstractions, while still being able to separate structural and behavioral parts [56]. This is still under development, however.

Agents CASM supports single- and multi-agent specifications [55]. The agents are controlled by the special `program` function. For an agent to set its `program` function a special function `self` is available. This function is different for each agent.

An agent terminates if either its `program` function is set to `undef` or if the executed rule did not produce any updates. So if a rule produces updates but does not change the `program` function, the rule is executed again. The `program` function is initialized (Börgers *main rule name*) with the `init` definition.

An example CASM specification can be seen in Listing 2.1. In the example, one (`1`) is added to the function `a` and the function `b` gets updated with the result. In the next step, the functions `a` and `b` are swapped. After the execution, the functions `a` and `b` have the values of one (`1`) and zero (`0`) respectively.

The Lines 4 and 5 define two functions `a` and `b` with a null-ary type `Integer`. The function `a` further is initialized to the value zero (`0`). Since the function `b` is not initialized, it has the value `undef`. In Line 2, the start rule of a single agent is specified as the `addition` rule.

When executing the rule `addition`, the function `b` is updated with the value of `a + 1` (Line 4) and the `program` function is updated with a reference to the `swap` rule (Line 9). Therefore, the ASM executes the `swap` rule in the next step. In the rule `swap`, the function `a` is updated with the value of the function `b` and `b` with the value of `a`. As `swap` is a block rule, it has parallel update semantics and the values are swapped correctly. The rule also updates the `program` function to `undef` (Line 15), which leads to the termination of the agent after the rule has been executed.

```
1  CASM
2  init addition
3
4  function a : -> Integer = { 0 }
5  function b : -> Integer
6
7  rule addition = {
8      b := a + 1
9      program( self ) := @swap
10 }
11
12 rule swap = {
13     a := b
14     b := a
15     program( self ) := undef
16 }
```

Listing 2.1: CASM Specification Example

Compiler

The CASM compiler performs model based transformations into different models which are used for different purposes [57]. The AST is the first model in the compilation. It

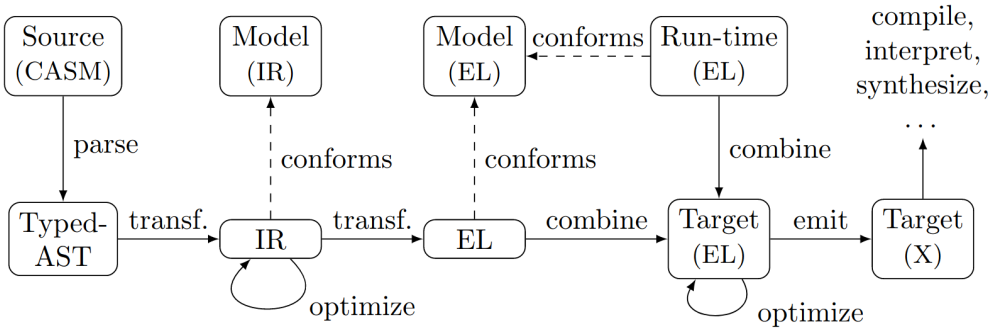

Figure 2.1.: CASM Compiler Models by Paulweber et al. [57]

is used for type inference, consistency checks etc.. The AST is aware of ASM semantics as is the Intermediate Representation (IR) layer. The IR is used to analyze, transform and optimize the specification with the knowledge of ASM-related properties [55]. After the IR, the specification is transformed into an Emitting Language (EL) which is ASM unaware. In this model, general optimizations can be performed, for example optimizing towards smaller file sizes or speed [57]. In the final transformation, the EL is transformed into an artifact in a target language.

2.2. Symbolic Execution

Symbolic execution, in contrast to numeric execution, does not use numeric values for its calculations, but symbols. Each symbol may be any valid value of its type domain, such that an integer symbol could be any whole number. Instead of results, an operation, such as addition, produces a logic formula which specifies the relationship between the input and the output symbols [46].

Where numeric execution follows a specific path through the program, symbolic execution evaluates all possible paths [6, 53]. Each path is preceded by the Path Condition (PC), which builds a complete model of the program. This is useful for software testing as strong guarantees can be given regarding reachability, assertions, value domains or the like. In contrast, numeric software testing methods such as tests or fuzzing under-approximate the analyzed property, which can lead to possibly incorrect results.

While symbolic execution can give strong guarantees, it is computationally expensive. To solve this problem, several strategies have been developed over the years [6]. These strategies can be divided into two groups, *forward* and *backward* symbolic execution.

Backward symbolic execution starts at a specified target point in the program and explores backwards until it reaches the starting point. While traversing backwards, a path condition is created and tested whether it can be met. This reduces the number of paths explored, because irrelevant paths are discarded as early as possible. However, a backwards symbolic execution requires an inter-procedural control flow graph to determine where a function was called from. Creating such a graph may be challenging [6].

Like numeric execution, Forward symbolic execution starts at the beginning of the program. In order to reduce the number of paths, symbolic and numeric execution are combined (*concolic* execution) [6, 23]. Concolic execution trades completeness with computation speed.

One approach for concolic execution is Dynamic Symbolic Execution (DSE) [40]. With this approach, symbolic and numeric execution are performed simultaneously. The numeric execution drives the execution, that is, when selecting paths, the numeric result

is considered. Simultaneously, a store with symbolic variables is updated. Whenever a path must be selected, the path which the numeric execution would take is used and the condition is added to the path condition. After a path is fully executed, a condition along the path is inverted and execution continues from this path forward, until the computation time has expired or the desired coverage is achieved.

2.3. Thousands of Problems for Theorem Provers

The Thousands of Problems for Theorem Provers (TPTP) is a library of logic problems for ATP systems [63]. It was built as a testing and evaluation library for ATP systems and contains problems in many different areas. TPTP problems are formulated in the TPTP language [67, 68]. The TPTP language allows problems and formulae to be written in various formats. Those formats range from *Clause Normal Form* (CNF) to *First Order Form* (FOF) [65], *Typed First Order Form* (TFF) [64] to *Typed Higher Order Form* (THF) [66]. The TPTP language can be parsed by many different SAT solvers because it is widely used as a benchmark or for testing.

First Order Formulae

In TPTP First Order Formulae (FOF) are used to represent theories in first order logic [43, 35, 51]. First order logic extends propositional logic with predicates and quantifiers. In TPTP unbound variables are called constants. All variables and constants are in the domain of individuals (`$i`). Elements in the domain of individuals are pairwise distinct and have no further semantic meaning. Predicates map from any number of arguments of type `$i` to a pseudo boolean type `$o`. The domain of `$o` contains only the values `$true` and `$false` and defines all operators in boolean algebra as expected.

When a TPTP file contains multiple FOF formulae, all formulae must be true for the file to be satisfiable. This is equivalent to connecting all formulae with a logical *and*.

Typed First Order Formulae

Typed First Order Formulae (TFF) extend FOF with types [64]. Each constant or variable has a specific type which is called sort. All sorts are non-empty, pairwise distinct domains. The equality predicate is ad-hoc polymorphic over all types, but an equation over different sorts is ill typed. TPTP also supports polymorphic predicates [16].

TPTP defines the numeric sorts of integers, rational and real numbers (`$int`, `$rat`, `$real`). For these sorts, a number of predicates are defined ranging from comparisons, like *greater* or *less*, to numeric algebra like *sum* or *multiply*. Similar to FOF, predicates map any

number of arguments of different types to the pseudo type `$o`. As `$o` is a pseudo type, cannot be used as an argument of a predicate.

3. Design

This chapter describes the design of the symbolic execution framework. First, it illustrates the general architecture, then it describes how a TPTP trace of a CASM specification looks like. Third, it explores how different CASM constructs translate to TPTP and lastly, it shows how the system deals with a numeric and symbolic function mismatch.

The symbolic execution framework is built in a modular fashion. Each computation step passes its results to the following unit through defined APIs. Therefore, each module is exchangeable with any implementation which provides the same API.

3.1. Architecture

The central data structure of the symbolic execution framework is the TPTP model. The overall architecture of the framework can be seen in Figure 3.1.

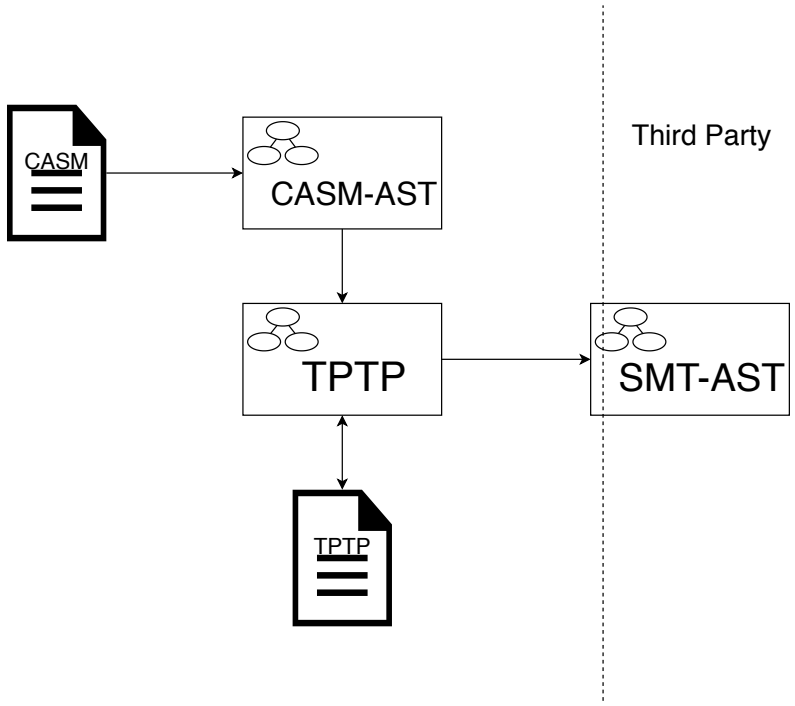

Figure 3.1.: Architecture of Symbolic Execution Framework

The TPTP model is the in-memory representation of a TPTP specification [63]. It can either be loaded from a file, or transformed from a CASM-AST. The TPTP model is more closely described in Section 3.1.1.

To execute a trace, the TPTP model is transformed into a model of a SMT solver. Alternatively, instead of directly calling the SMT solver, the TPTP model can just be dumped to a file. This allows for using SMT solvers for which no public API is available for transformation, but which provide a TPTP language parser.

This stands in contrast to the design by Lezuo [48]. In that design, as seen in Figure 3.2, the CASM-AST gets transformed directly into a TPTP file. This file, in turn, is then read by an external SMT solver.

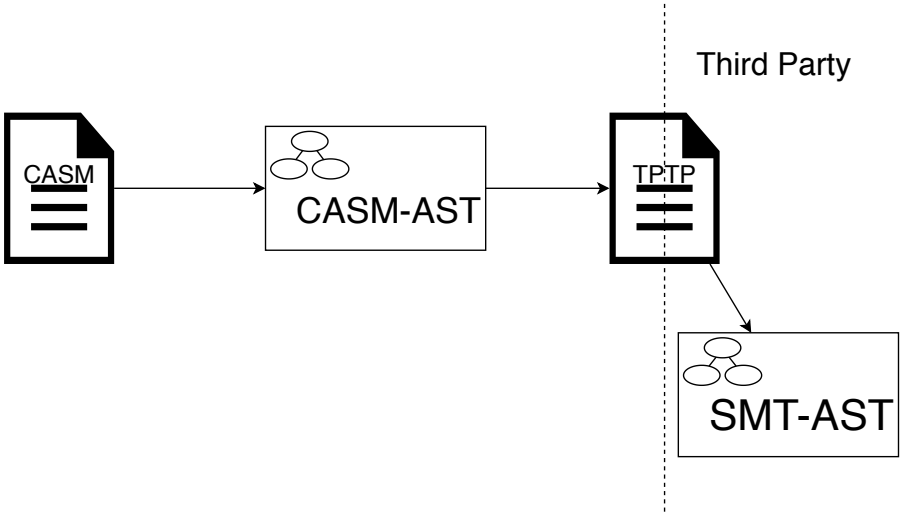

Figure 3.2.: Architecture of Symbolic Execution by Lezuo [48]

The approach of the model-to-model transformation, as built in this project, has the advantage that the model can be semantically and syntactically checked after the transformation has been completed.

3.1.1. TPTP Model

The TPTP model is very closely based on the language definition. This allows for usages different than this thesis, since any valid TPTP input can be represented correctly, not only the subset used in this project. Using TPTP has its disadvantages, however, since the TPTP language has only basic datatypes defined. This necessitates the explicit definition of dataypes, like bit vectors, if used. In contrast to TPTP, other frameworks, such as the SMB-Lib2.0 [9], implicitly define types and theories common to program verification.

3.1.2. Trace Description

A correct trace, as seen in Listing 3.1, consists of four parts. The first part includes the type declarations for calculation variables (Line 2 to 7), the second part contains the definitions for language operands (Line 8), the third part contains all function definitions (Line 9 to 11) and the fourth part is the execution trace itself (Line 12 to 21).

```
1  % specification:
2  tff(3,type,'%0':$int).
3  tff(5,type,'%1':$int).
4  tff(7,type,'%2':$int).
5  tff(11,type,'%3':$int).
6  tff(14,type,'%4':$int).
7  tff(17,type,'%5':$int).
8  tff(8,hypothesis,'#add#i':($int*$int*$int)>$o).
9  tff(0,hypothesis,'@a':($int*$int)>$o).
10 tff(1,hypothesis,'@b':($int*$int)>$o).
11 tff(2,hypothesis,'@c':($int*$int)>$o).
12 tff(4,hypothesis,'@a'(1,'%0')).
13 tff(6,hypothesis,'@b'(1,'%1')).
14 tff(9,hypothesis,'#add#i'('%0','%1','%2')).
15 tff(10,hypothesis,'@c'(2,'%2')).
16 tff(12,hypothesis,'@a'(1,'%3')).
17 tff(13,hypothesis,'@a'(0,'%3')).
18 tff(15,hypothesis,'@b'(1,'%4')).
19 tff(16,hypothesis,'@b'(0,'%4')).
20 tff(18,hypothesis,'@c'(2,'%5')).
21 tff(19,hypothesis,'@c'(0,'%5')).
```

Listing 3.1: TPTP Trace

All formulae are written as Typed First Order Formulae (*TFF*). This differs from the previous implementation, because as of TPTP version 7.0 [63] *TFF* and First Order Formulae (*FOF*) do not work together as expected.

In *FOF* all variables and constants are assumed to be in the same infinite domain. That domain, however, is distinct from any type domain in *TFF*. This implies that every constant and variable in a *TFF* formula is not equal to any constant or variable in a *FOF* formula.

Constant Definitions

Since each variable is used only for one purpose and a TPTP trace has no notion of execution time, these calculation variables are defined as constants, as seen in Listing 3.2. The definition for these constants needs to be prior to the first usage of the constant. If a constant is used ahead of its declaration, it must be assumed to be in the sort of individuals *$i*. It does not matter whether the sort could be inferred from the context [64], the type must be *$i*. To prevent wrong variable types and improve readability, all constant

definitions are collected at the beginning of the trace.

The sort of a constant is always a simple type, not a mapping type which would make it a function or a predicate. A constant therefore cannot have parameters, which is not needed however, as they are only used as temporary variables in calculations.

```
1 % specification:
2 tff(3,type,'%0':$int).
3 tff(5,type,'%1':$int).
4 tff(7,type,'%2':$int).
5 % <omitted>
```

Listing 3.2: TPTP Constant Definitions

Language Operands

To add calculation semantics to the trace, the operations must be defined. In TPTP this is easiest done using functors. A functor is akin to a function in the sense that it has multiple parameters and one return value. The number of parameters is the arity of a functor. Therefore all binary language operands of CASM use a ternary functor. It is ternary because two arguments are needed for the actual input values and the third argument is the result. As the functor is used as a predicate, the return type is always boolean.

Semantically, the functor works as follows. The functor is true if, and only if, $operand1 \circ operand2 = result$ is true, so Equation 3.1 is true for all possible inputs.

$$operator(op1, op2, result) = (op1 \circ op2 = result) \tag{3.1}$$

Since the functor is always in its own formula (see Listing 3.1, Line 14), it has to be true for all inputs for the trace to be satisfiable. This leads to the desired constraint that $operator1 \circ operator2 = result$.

The TPTP language supports first order polymorphic functors in *TFF* with the *TFF1* extension [16]. This is especially useful for container types like maps or lists, but is less useful for CASM language operators as not all CASM types have the same behavior for each type. An example is the slash (`/`) operator for the types `Integer` and `Rational`. In the case of the `Integer` type, the slash represents the floored division result and in the case of `Rational`, it is the mathematical division.

For that reason, a new functor is declared and defined for each use of a language operator with a different type. Function overloading, as known from general purpose languages like C++, where the type signature of a function is part of its name, is generally not possible in TPTP. As a result, each functor contains the type of its arguments in the name resulting in `'#<operator>#<type>'`.

While operators could be generated for all types, it is a lot of unnecessary work if the functors are not used and with user defined CASM types, it may not even be possible to do that. Hence, the generation of operator functors follows the lazy allocation principle and only defines operators for types which are used.

Function Declarations

CASM functions are mapped to TPTP functors, instead of constants or variables. This is because constants cannot change in the trace. CASM functions, however, may change in any rule, so the value is time dependent. Variables are not possible either, because the TPTP trace has no notion of time. Therefore, a variable cannot have a value for some formulae and another for different formulae.

Instead, a similar approach to the language operands is taken, which was introduced by Lezuo [48]. Each CASM function is mapped to a TPTP functor with 2 or more arguments. The first argument is always the time of the operation, the last argument is the constant which is to be equal to the function. All remaining arguments correspond to the arguments in an n-ary CASM function. Listing 3.3 and Listing 3.4 depict an exemplary transformation of various function declarations. The sample specification consists of three function definitions. These functions are updated with literal values in the rule `access`.

In TPTP, the return type is Boolean (`$o`) for all functors, similar to the language operands, as they are predicates. The arguments are different for all three functions. The null-ary CASM function `a` becomes a binary predicate in TPTP, the first argument being the time and the second the value. The unary CASM function `b` transforms into a ternary predicate in which the second argument in TPTP is the first argument in CASM. Lastly, the binary CASM function `c` turns into a 4-ary predicate in TPTP.

```
1  CASM
2  [symbolic] function a : -> Integer
3  [symbolic] function b : Integer -> Integer
4  [symbolic] function c : Boolean * Boolean -> Integer
5
6  rule access = {
7      a := 1
8      b(2) := 3
9      c(true, false) := 4
10 }
```

Listing 3.3: CASM Functions

The TPTP predicate is true if and only if the semantic meaning is correct for all input values. The semantics of a function access is that for a given time and arguments the function is equal to the value which is supplied as the last parameter. Examples of usage are shown in Listing 3.4 in Line 18.

```
% specification:
tff(0,hypothesis,'@a':($int*$int)>$o).
%                       ^    ^------- value to be equal
%                       |------------ time

tff(1,hypothesis,'@b':($int*$int*$int)>$o).
%                       ^    ^    ^------- value to be equal
%                       |    |------------ first argument
%                       |----------------- time

tff(2,hypothesis,'@c':($int*$o*$o*$int)>$o).
%                       ^    ^  ^  ^------- value to be equal
%                       |    |  |---------- second argument
%                       |    |------------- first argument
%                       |------------------ time

% <omitted>
tff(10,hypothesis,'@a'(2,1)).     % at time 2: a equals to 1
tff(15,hypothesis,'@b'(2,2,3)).   % at time 2: b of 2 equals to 3
tff(16,hypothesis,'@c'(2,$true,$false,4)). % at time 2: c of true,
    false equals to 4
% <omitted>
```

Listing 3.4: CASM Functions in TPTP

Since the semantic is equivalence and not assignment, the same access method can be used for reading values (see Section 3.2.1 for more details).

Execution Trace

The execution trace of the CASM specification itself is the last part of a TPTP trace (see in Listing 3.5). Each step in the computation is represented as separate formula. The calculation formulae use the operator functors and these functors, in turn, need constants as arguments. Therefore, an execution step consists of accessing the CASM functions, the operand functor and storing the result in a CASM function.

Part of the specifications for the TPTP trace is that the content of each CASM function at termination is modeled in TPTP as the content of the TPTP function at time zero (0).

```
%<omitted>
tff(4,hypothesis,'@a'(1,'%0')).
tff(6,hypothesis,'@b'(1,'%1')).
tff(9,hypothesis,'#add#i'('%0','%1','%2')).
tff(10,hypothesis,'@c'(2,'%2')).
tff(12,hypothesis,'@a'(1,'%3')).
tff(13,hypothesis,'@a'(0,'%3')).
tff(15,hypothesis,'@b'(1,'%4')).
tff(16,hypothesis,'@b'(0,'%4')).
tff(18,hypothesis,'@c'(2,'%5')).
tff(19,hypothesis,'@c'(0,'%5')).
```

Listing 3.5: TPTP Execution Trace

Listing 3.5 depicts only the execution trace part. From Line 2 to 3, the contents of `a` and `b` are saved in the TPTP constants `'%0'` and `'%1'` respectively. From these constants, `'%2'` is calculated (Line 4) and written to `c` in Line 5. In Line 6 to Line 11, the function accesses are listet that are required to set the last function value at time zero (0) according to the specification.

3.2. Transformation

The generation of a TPTP trace from a CASM specification involves multiple compiler passes, since the CASM compiler is built as a multi pass compiler [57, 55]. As illustrated in Figure 3.1 on page 13, the CASM specification is first transformed into an Abstract Syntax Tree (AST) by the CASM compiler front end. The CASM AST is then transformed into a in-memory TPTP model. Lastly, the TPTP model is written to a file as trace.

Transforming the CASM AST into the TPTP model, instead of transforming it directly into a text file, has the advantage, that it is more flexible, as it allows reordering of formulae or changing them after they have been first constructed. The model-to-model approach has limited syntax checking capabilities since it must fit into the model of the language. It also allows for modifications on the generated model for optimization purposes. Another benefit is that some form of source location can be added to each TPTP formulae. This, in combination with a direct transformation from the TPTP model into a model of a SAT solver, can be used to directly map errors in the SAT solver to the CASM specification. The model-to-model approach also increases resilience to changes in the TPTP language. Small changes in the syntax of TPTP do not have to be changed in the transformation from CASM to TPTP itself, but only in the transformation from the TPTP model to the TPTP file.

3.2.1. Model-to-Model Transformation

The transformation from a CASM specification to a TPTP trace uses a concolic (partly concrete, partly symbolic) approach. Function symbols which are not explicitly set to be symbolic by an appropriate function attribute, are evaluated numerically. While this puts constraints on the validity of the result, it allows for a simpler transformation and a smaller trace for the interesting cases.

However, this distinction between symbolic and numeric functions introduces the problem of symbolic consistency. It is not possible to update a numeric function with a symbolic value, as the value is unknown during the transformation. This problem is further discussed in Section 3.3.

Listing 3.6 depicts the CASM specification which produces the TPTP trace shown in

Listing 3.1 on Page 15.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  [symbolic] function b : -> Integer
7  [symbolic] function c : -> Integer
8
9  rule test =
10 {
11     c := a + b
12     program( self ) := undef
13 }
```

Listing 3.6: CASM: Symbolic Addition

For a function to be symbolic and be considered in the trace, the `symbolic` attribute must be added to the function definition. Without it, the function is considered numeric and will not appear in the TPTP trace as function, but only the function values as literals.

Unlike CASM functions, the TPTP functions do not support the `undef` value. However, this is not necessary as a value of `undef` is a numeric value. Due to the specification of `undef` variables, language operands with an `undef` value have a concrete result with `undef` values. That is, they either result in `undef`, further propagating the value through the computation, or result in the other operand, which may be symbolic.

But the calculation of that operand is done in the transformation step and not in the TPTP trace, so the `undef` value is not needed there. This has the limitation, however, that a symbolic function must not be `undef` at initialization, as otherwise symbolic and numeric execution may differ.

An example of this is illustrated in Equation 3.2 for the addition (+) operator [55]. The *symbolic* predicate denotes if the argument is either a symbolic function or a numeric one. The *symbolic_add* function does the addition symbolically and the + operator is a numeric addition.

$$add(A,B) = \begin{cases} undef, & \text{if } A = undef \vee B = undef \\ sym', & \text{if } (symbolic(A) \wedge B \neq undef) \vee (symbolic(B) \wedge A \neq undef) \\ A + B, & \text{otherwise} \end{cases} \tag{3.2}$$

If either of the arguments are `undef`, the result is `undef`. If neither is `undef`, but at least one is symbolic, a symbolic addition takes place. If both operands are numeric but not `undef`, a numeric addition is done. This decision can be made at transformation time, as

the cases are mutually exclusive, since $symbolic(A) \implies (A \neq undef)$.

In the scope of this thesis, some specific CASM functions are not supported to be symbolic. One example is the CASM function `program`, which is used by an executing agent to determine the next rule to be executed. To make this function symbolic would mean that the transformation must follow every possibility. This grows quickly out of hand, as can be seen in the example in Listing 3.7.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  function rules : Integer -> RuleRef< -> Void > = {
7      (0) -> @doSomething,
8      (1) -> @doSomethingElse
9  }
10
11 rule test =
12 {
13     program( self ) := rules(a)
14 }
15
16 rule doSomething = skip
17
18 rule doSomethingElse = skip
```

Listing 3.7: CASM: Symbolic Program Function

In this example, the `program` function depends on the value of the function `a`, which is symbolic. Hence, the value of the function `program` cannot be known at transformation time. To correctly transform the CASM specification, the TPTP trace would need to execute all possible rules, and add to each rule the condition "if function `a` has the value of φ, then do ...", where φ is the condition in which the rule is executed. If multiple of such conditional updates to the `program` function follow, the number of steps to compute increases exponentially. The same problem can be created using conditional rules (see Section 3.2.1 and Section 3.3.3). This splitting of the control flow can only be counteracted if there is a way to re-unify different control flows again later, but this is beyond the scope of this thesis.

Notion of Time in TPTP

A TPTP trace has no intrinsic time component since it consists only of logical formulae. A CASM execution, however, does have some notion of time, as different rules may be evaluated sequentially by updating the `program` function. In order to translate this behavior to TPTP, the *time* parameter is added for all functions [48]. The time is not needed for

calculations, since each TPTP constant is used exactly once. As defined by Lezuo [48], there are two special points in time. Time *1* is the step at which initialization happens and time *0* is the step at which the execution terminates. This provides an easy method of adding constraints on the start and termination state of any function, since the times are known for any CASM specification.

Accessing Functions

As briefly discussed in Section 3.1.2, the semantics of function functors in TPTP is based on equivalence. The functions in CASM, however, are updated or read so the data flow has a direction. For this transformation to work, the *time* parameter in the functors is key to differentiate between different states of the function. Updating a function is the same as setting the function value at the current time to be equivalent to the value to be set. This allows for only one update per step and function. However, this is insignificant, as it corresponds to the property of consistency (clash-free updates) to ASM functions.

That behavior can be seen in the TPTP trace in Listing 3.9 (Line 3). The corresponding CASM specification (Listing 3.8, Line 10) is an update rule of the `b` function. To a TPTP trace, this translates to the equivalence of the TPTP constant (`'%0'`) and the function `'@c'` at time 2, which is the first step after the initialization.

Reading a value in CASM, however, means in TPTP that the TPTP constant to be "written into" is equal to the function the last time it was written. In the TPTP trace this can be seen in Listing 3.9 (Line 2), where the TPTP constant `'%0'` is equal to the last time `b` was written. If a function has never been written before, the constant is equal to the function at initialization time (*1*).

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  [symbolic] function b : -> Integer
7
8  rule test =
9  {
10     b := a
11     program( self ) := undef
12 }
```

Listing 3.8: Accessing Functions

```
1  %<omitted>
2  tff(4,hypothesis,'@a'(1,'%0')).
3  tff(10,hypothesis,'@b'(2,'%0')).
4  %<omitted>
```

Listing 3.9: TPTP Accessing Functions

To sum up the TPTP trace in Listing 3.9, the TPTP constant `'%0'` is equal to `'@a'` at time 1 and also to `'@b'` at time 2. This results in function `'@a'` at time 1 to be equal to function `'@b'` at time 2, which is the semantics of the update rule in the CASM specification in Listing 3.8.

Let Rule

In a CASM specification, a *let rule* is a binding of a expression to a variable for a sub-rule. The transformation takes care of the renaming but does not produce the renamed variable, it uses the respective TPTP constants instead. Therefore, the example specification in Listing 3.10 produces exactly the same trace as Listing 3.6 on Page 20.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  [symbolic] function b : -> Integer
7  [symbolic] function c : -> Integer
8
9  rule test =
10 {
11     let d = a + b in
12         c := d
13     program( self ) := undef
14 }
```

Listing 3.10: Addition with Let

During the transformation, the result of the CASM expression `a + b` is a constant in TPTP, in this specific case `'%2'`. The transformation pass saves that constant with the information that the CASM function `d` is this constant. Since `d` is not available outside of the scope of the sub-rule, no time information is needed because all updates in the sub-rule occur at the same time as the enclosing rule. Every time `d` is referenced in the sub-rule, `'%2'` is used for the TPTP trace.

Sequential Rule

CASM supports the inclusion and nesting of sequential rules and block rules. Block rules have parallel semantics, whereas sequential rules have sequential semantics. CASM handles the sequential blocks by saving all function updates in a pseudo state which is then merged with the update-set of the enclosing block rule [50, 49]. When leaving the sequential rule, the last pseudo state is merged with the update-set of the surrounding block rule. All previous pseudo states are discarded as the effects of these states are present in the last pseudo state. When the outermost block rule is finished and the update-set is

applied to the global function state, none of the intermediary pseudo states are present and therefore are not explicitly written to the global state. This is why intermediary updates in sequential rules do not appear in TPTP traces as only the update of the global state is written.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  [symbolic] function b : -> Integer
7  [symbolic] function c : -> Integer
8
9  rule test =
10 {
11     c := a + b
12     {|
13         a := b + 1
14         a := a + c
15     |}
16     program( self ) := undef
17 }
18
19 rule alternative =
20 {
21     c := a + b
22     a := (b + 1) + c
23     program( self ) := undef
24 }
```

Listing 3.11: Sequential Rule

In Listing 3.11, the update (Line 13) to `a` is saved in a new pseudo state. Line 14 causes a second pseudo state to be created with the new value of `a`. After that, the sequential rule is finished and the last pseudo state (state of Line 14) is merged with the update-set of the surrounding block rule which contains only `c`. The merged update-set includes the new value for `c` and `a` as they have no conflicting updates. This merged update-set is then applied to the global state which is visible in the TPTP trace. All previous pseudo states are not visible in the TPTP trace. This interaction is illustrated in Figure 3.3. The update-set of the alternative rule (Line 22) is identical to the update-set of the original rule, as the pseudo states are discarded after the merge. Both rules therefore produce the same TPTP trace.

Conditional Rule

Conditional rules can fall into two cases. In the first case, the branch condition consists only of numerical values and functions. In said case, the decision as to which branch to take is made at transformation time. Therefore, the conditional rule is never visible in the TPTP

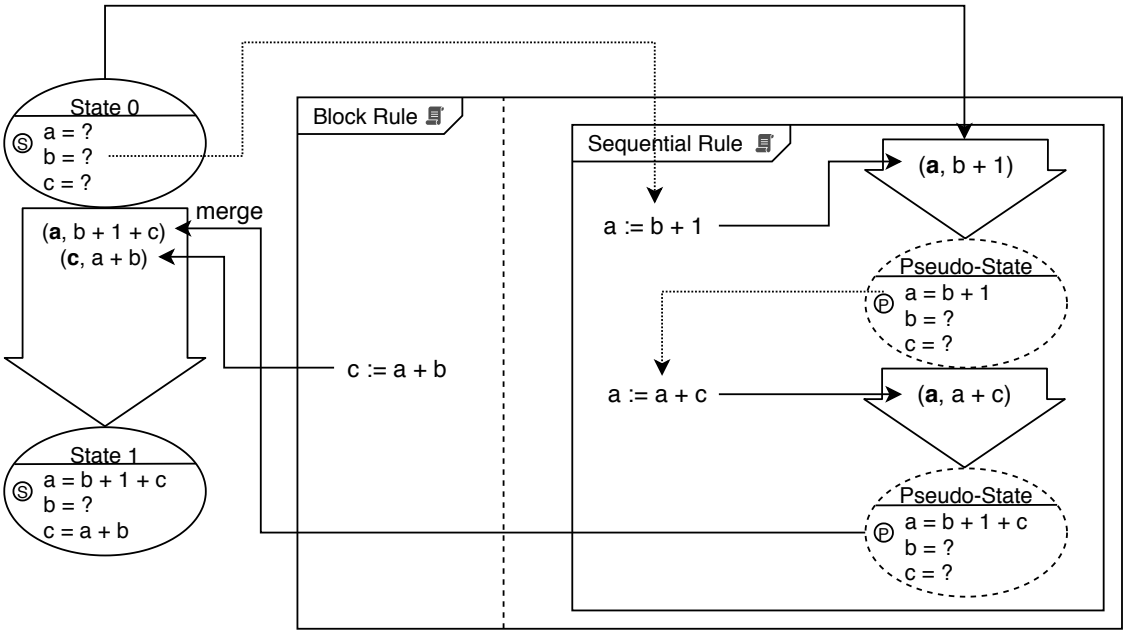

Figure 3.3.: States in Sequential Rule from Listing 3.11

trace, as only the relevant part of the specification is transformed while the other part is ignored. In the second case, if the branch condition includes some symbolic functions, then the rule is handled differently. In that case, the condition and both possibilities must be included in the TPTP trace, as the decision as to which branch of the conditional rule to take cannot be made at transformation time.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  [symbolic] function condition : -> Boolean
7
8  rule test =
9  {
10     if condition then {
11         a := 1
12     } else {
13         a := 2
14     }
15 }
```

Listing 3.12: Conditional Rule

Since one CASM specification should translate to exactly one trace, conditional rules are handled in a way that all TPTP formulae that are generated in a branch are prepended

with an implication of the correct condition for the branch.

The example in Listing 3.12 is translated to the TPTP trace in Listing 3.13. All rules from the if-branch get prepended with the implication that the condition is true, as seen in Line 3. In other words, if the condition is true, all rules from the if-branch must also be true. For the else branch, on the other hand, the implication that the condition is not true is added for the same reason.

Alternatively, one could use an equivalence instead of an implication as connective between condition and predicate. However, this would be incorrect as a predicate with the negated condition may happen to be true. This is exemplified by a CASM specification which reads the same function in both branches. The produced TPTP predicate is true in both cases.

```
1  % <omitted>
2  tff(9,hypothesis,'@condition'(1,'%1')).        % read condition
3  tff(10,hypothesis,'%1' => ('@a'(2,1))).        % if branch
4  tff(11,hypothesis,(~'%1') => ('@a'(2,2))).     % else branch
5  % <omitted>
```

Listing 3.13: CASM Conditional Rule in TPTP

Iterate Rule

Iterate rules are similar to conditional rules in that the next computation step in the CASM specification is dependent on functions. As with conditional rules, there are two cases, one where the iteration count is only dependent on numeric variables and one where the iteration count is dependent on symbolic variables.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  function b : -> Integer
7
8  rule test =
9  {
10     iterate {
11         if b < 10 {
12             b := b + 1
13             a := a * 2
14         }
15     }
16 }
```

Listing 3.14: Iterate Rule

Currently, only the first case is supported for transformation, since the other variant cannot find out whether and when the iterate rule terminates.

If only numeric functions are used in determining whether the iterate rule terminates, then this can be evaluated at transformation time, see Listing 3.14. The TPTP trace therefore contains the iterate rule in an unrolled fashion (Listing 3.15).

```
1  % <omitted>
2  tff(9,hypothesis,'@a'(1,'%0')).            % iteration 0
3  tff(10,hypothesis,'#mul#i'('%0',2,'%1')).
4  tff(11,hypothesis,'@a'(2,'%1')).
5  tff(12,hypothesis,'@a'(2,'%2')).           % iteration 1
6  tff(13,hypothesis,'#mul#i'('%2',2,'%3')).
7  tff(14,hypothesis,'@a'(3,'%3')).
8  tff(15,hypothesis,'@a'(3,'%4')).           % iteration 2
9  tff(16,hypothesis,'#mul#i'('%4',2,'%5')).
10 tff(17,hypothesis,'@a'(4,'%5')).
11 tff(18,hypothesis,'@a'(4,'%6')).           % iteration 3
12 tff(19,hypothesis,'#mul#i'('%6',2,'%7')).
13 tff(20,hypothesis,'@a'(5,'%7')).
14 % <omitted>
```

Listing 3.15: CASM Iterate Rules in TPTP

3.2.2. Trace Generation

The transformation step from the CASM-AST to the TPTP model, as described in the previous section, produces the TPTP models in memory. To output a file, the TPTP model must be transformed to text. This is achieved by traversing the whole AST in order and dump all tokens into a stream.

However, a problem regarding brackets in binary formulae occurs as seen in in Figure 3.4. It consists of an equivalence between two binary formulae. While the two binary formulae farther down the AST do not produce errors as their arguments are constants, the equivalence between those formulae would require brackets around its arguments. The output without the brackets can be seen in Listing 3.16, Line 1. This is an illegal TPTP formula, as only the *and* (`&`) and *or* (`|`) connectives have defined associativity. All other connectives such as *equivalence*, *implication*, etc. must explicitly define their associativity using terms surrounded by brackets. Therefore, the correct output would contain brackets as seen in Line 2.

```
1  tff(1,hypothesis,a<=>b<=>b<~>c).              % incorrect result
2  tff(1,hypothesis, (a <=> b) <=> (b <~> c)).   % correct output
3  tff(1,hypothesis, (((a) <=> (b)) <=> ((b) <~> (c)))).   % defensive
       output
```

Listing 3.16: CASM Iterate Rules in TPTP

One possibility to mitigate that problem is to wrap each part of the formula in brackets, as seen in Line 3. While adding more brackets to the formula solves the aforementioned

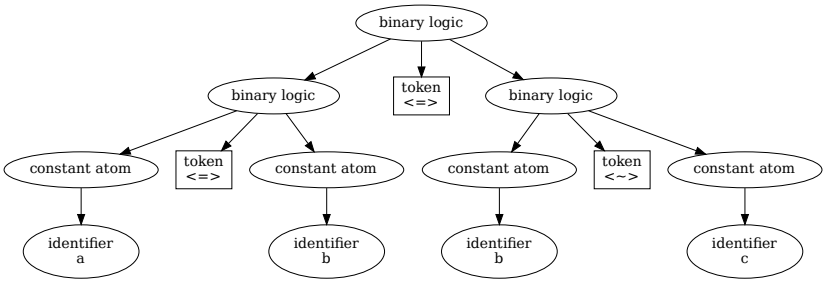

Figure 3.4.: TPTP Model: Binary Formulae

problem, it reduces readability.

It is up to the creator of the model, in this case the model-to-model transformation as discussed in the previous section, to explicitly set appropriate brackets.

3.3. Symbolic Consistency

Symbolic consistency is achieved if all function updates with symbolic values are updating symbolic functions. While updating a symbolic function with a numeric value is possible (Listing 3.17, Line 9), it is impossible to update a numeric function with a symbolic value (Listing 3.17, Line 10). The symbolic value can be any value in the type domain and is unknown at transformation time. In order to update a numeric function, however, a concrete value is needed. For this reason, the symbolic consistency must be checked before executing and, if needed, numeric functions must be set symbolic.

```
1  CASM
2
3  init test
4
5  function a : -> Integer
6  [symbolic] function b : -> Integer
7
8  rule test = {
9      b := a // OK
10     a := b // Error: numeric function is updated with symbolic value
11 }
```

Listing 3.17: Symbolic Consistency: Updates

If an update of a numeric function with a symbolic value is encountered, there are two possible ways to proceed. The first is to simply abort execution and report the error to the user to manually fix the symbolic status of the CASM specification. The second is to "promote" the numeric function to be symbolic and continue the transformation with that

function being symbolic.

The implementation by Lezuo [48] made use of the first approach, in which the transformation failed. In the present thesis, the second approach is utilized which means all updates are checked for symbolic consistency. If an inconsistency is found, the symbolic status of the function is updated.

The intuitive approach would be to promote a function to symbolic if a conflict, as described above, occurs and restart the symbolic execution step. However, this might result in very different traces if a numeric function is updated with a symbolic value in one branch of a conditional rule, but not in the other. In this case, as seen in Listing 3.18, the symbolic status depends on the numeric value of a function during transformation.

```
1  CASM
2
3  init test
4
5  function a : -> Integer
6  [symbolic] function b : -> Integer
7
8  rule test = {
9      if a = 1 then { // condition is initially numeric,
10                     // evaluated at transform time
11         a := 2      // a is numeric
12     } else {
13         a := b      // a is symbolic, causes conditional rule to
14                     // be evaluated symbolically
15     }
16 }
```

Listing 3.18: Symbolic Consistency during Execution

In this example (Listing 3.18), if the condition is true, `a` is updated with `2` but no symbolic trace is generated, as no symbolic functions are used or updated. If the condition is false, `a` is updated with a symbolic value and therefore gets promoted to be symbolic. After the promotion, the execution is restarted, and the condition is evaluated symbolically, generating both branches in the TPTP trace.

For reliable results, both branches of the conditional rule must be evaluated in order to determine the symbolic status of a function. However, this is unfeasible during the transformation step, as the branch, which is only executed because of the symbolic status of functions, may produce updates. These updates must be ignored and side effects of sub-rules like the `print` rule must be prevented.

To avoid the need to conditionally prevent side effects, the analysis for symbolic consistency is done before the transformation step. In this step no actual calculations are performed, only the symbolic status of functions and expressions is recorded. This is a very cheap operation, as nearly all language operators return a symbolic value if either

one of its arguments is symbolic.

If any symbolic conflict is encountered, the offending function is promoted to be symbolic and the check restarts. This is due to the fact that this newly symbolic function may cause other functions to have a symbolic conflict at any time in the specification.

3.3.1. Consistency with Called Rules

In order to achieve that all code paths are followed within reasonable checking time, each rule is evaluated in isolation. This is possible by registering the call with the symbolic status of its arguments, but not evaluating the called rule immediately. This creates a rule-call-graph.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  function b : -> Integer
7  function c : -> Integer
8  function d : -> Integer
9  function e : -> Integer
10
11 rule test = {
12     callable(1, 0)
13     callable(a, 1)
14     d := b + 1
15 }
16
17 rule callable(param1 : Integer, param2 : Integer) = {
18     b := param1
19     c := param2
20 }
21
22 rule notCalled = {
23     e := a + 1
24 }
```

Listing 3.19: Symbolic Consistency: Callable Example

After all calls have been registered, all arguments, which were not called symbolically, are set to be called with only numerical values, and the checking of the callable rule can be completed.

This use of a rule-call-graph has the advantage that all rules that are not used in a specification are not tested and therefore cannot influence the result.

In the specification in Listing 3.19 only the function `a` is considered symbolic, all other functions are numeric functions.

After the symbolic consistency check, the functions `a`, `b` and `d` are all symbolic, as they are updated with a symbolic value at one point or another. `b` is updated in Line 18, after

`callable` is called with a symbolic first parameter in Line 13. Since `b` is symbolic, the update of `d` in Line 14 results in `d` being promoted.

The function `c` remains numeric, as `callable` is only called with numeric values in the second parameter. The function `e` also remains numeric, as the rule `notCalled` is never evaluated in the specification. For a more detailed description of how the symbolic consistency pass works see Section 4.3.5.

3.3.2. Rules with Return Values

If a rule has a return type other than `Void`, the returned value can also be symbolic or numeric. As rules are not necessarily pure, in the sense that they may be influenced from the global function state, the symbolic status is determined on a rule basis, not a call basis.

This means that even a simple rule, as shown in Listing 3.20, which is pure, always returns a symbolic or numeric value, depending on the how the function is called.

```
1  // <omitted>
2  rule callable(p1 : Integer, p2 : Integer) -> Integer = {
3      result := p1 + p2
4  }
```

Listing 3.20: Symbolic Consistency: Function with Return Value

If at any point in the specification the rule `callable` is called with at least one symbolic argument, all calls to the rule return a symbolic value. Even if both arguments are numeric in another call, the rule returns a symbolic value.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  function b : -> Integer
7  function c : -> Integer
8
9  rule test = {
10     b := callable(1, 2)
11     c := callable(a, 1)
12 }
13
14 // <omitted definition of callable>
```

Listing 3.21: Symbolic Consistency: Return Value Symbolic

For example, in the specification in Listing 3.21, both `b` and `c` are promoted to symbolic, whereby, as shown in the specification in Listing 3.22, the function `d` remains numeric

because `callable` is only called with numeric arguments.

```
CASM

init test

function d : -> Integer
function e : -> Integer

rule test = {
    d := callable(3, e)
}

// <omitted definition of callable>
```

Listing 3.22: Symbolic Consistency: Return Value Numeric

For simplicity and consistency, these rules for symbolic consistency also apply to derived functions, which by definition are pure.

3.3.3. Conditional Rules

Conditional Rules, as explained in Section 3.2.1, must take into account two cases. In the first case, when the condition only depends on numeric functions and in the second case, when the condition depends on symbolic variables.

In both cases, however, both branches must be considered for symbolic consistency. The difference is in how numeric assignments are handled.

```
CASM

init test

[symbolic] function a : -> Integer
function b : -> Integer
function c : -> Integer
function d : -> Integer
function e : -> Integer

rule test = {
    if b < 0 then { // numeric condition
        c := a // symbolic update
        d := 1 // numeric update
    } else {
        d := 2 // numeric update
    }

    if a < 0 then { // symbolic condition
        e := 1 // numeric update
    } else {
        e := 2 // numeric update
    }
}
```

Listing 3.23: Symbolic Consistency: Conditional Rule

In the case of the numeric condition, numeric updates are handled as usual, because at transformation time, the correct branch is known and therefore the numeric update takes place as usual.

In the symbolic condition case, however, all updates in both branches are considered symbolic updates, as it cannot be determined which numeric update of which branch is to be done out at transformation time. This is only known when the symbolic trace is solved.

Listing 3.23 illustrates this behavior. After the symbolic consistency check, the functions `a`, `c` and `e` are symbolic, while the functions `b` and `d` remain numeric.

The conditional rule in Line 12 has a numeric condition because `b` is numeric. Therefore, all updates in the conditional rule are treated as usual. `c` is updated with a symbolic value which makes it symbolic. `d` is updated with numeric values in both branches so it remains numeric.

However, the conditional rule in Line 19 has a symbolic condition as `a` is symbolic. Although all updates to `e` in both branches are numeric, `e` is promoted to symbolic because the branch to be executed cannot be determined at transformation time.

The special case in which a function is updated in both branches of the conditional rule with the same numeric value is not specifically addressed. It also results in a symbolic update in a conditional rule with symbolic condition, despite the known update value at transformation time.

Unknown Contexts

There are some cases in which conditional rules combined with return values from rules cannot determine the correct symbolic status of a function, as shown in Listing 3.24.

```
1  CASM
2
3  init test
4
5  function a : -> Integer
6
7  rule test = {
8      if increment(a) < 10 then {
9          a := increment(a)
10     }
11 }
12
13 rule increment(p1: Integer) -> Integer = {
14     result := p1 + 1
15 }
```

Listing 3.24: Symbolic Consistency: Conditional Rule

A symbolic execution with the function `a` being numeric is possible, as it is neither

updated with a symbolic function, nor updated in a conditional rule with symbolic condition or called with symbolic arguments.

However, the symbolic consistency check only works locally on one rule at a time. During the rule `test`, the status of the condition is unknown because it depends on the return value of the rule `increment`.

The return value of `increment`, however, depends on the argument which is passed. As `increment` is called in Line 9 in an unknown context, the context is defensively considered to be symbolic. Therefore, `increment` is called in a symbolic context. Since it is now called with a symbolic argument, it returns a symbolic value. Hence, function `a` is promoted to symbolic as it is updated with a symbolic value.

3.3.4. Functions with Symbolic Arguments

Another way of promoting a function to symbolic is by calling it with a symbolic argument as the function cannot be resolved at transformation time.

In Listing 3.25 both functions `b` and `c` are promoted to symbolic by calling with symbolic arguments. The context in which they are called is irrelevant for that promotion.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  function b : Integer -> Integer
7  function c : Integer -> Integer
8
9  rule test = {
10     b(a) := 1
11     print(c(a))
12 }
```

Listing 3.25: Symbolic Consistency: Symbolic Function Arguments

4. Implementation

This chapter covers the implementation details of the symbolic execution framework. Firstly, it describes the infrastructure provided by the CASM compiler[1] in which it is located. It will then go on to discuss the transformation step and finally, it illustrates the details of the symbolic consistency check.

4.1. Passes

A compilation of a CASM specification is divided into many small steps, each of which is handled by a pass. Each pass has a specific task with defined input and output states. Most passes require a CASM-AST as input and output a CASM-AST with some additional information. One such pass is the `TypeInferencePass`. It takes a CASM-AST and outputs a CASM-AST with type information added to all nodes.

In order to get the desired output, the sequence of the passes is important. Therefore, a pass also contains information on passes that ought to be run before and after. For example, the `SymbolResolverPass` requires the `SourceToAstPass` and must be scheduled after the `AttributionPass`. The CASM interpreter specifies a pass to be run, for example the `SymbolicExecutionPass`, and a starting point which in most cases is an input file. It is then up to the pass system to find a series of passes to be run to go from the input data to the desired output via the specified pass.

4.2. Execution

The `SymbolicExecutionPass` builds on the numeric execution by delegating all functions that do not explicitly differ from numeric execution. The execution, both symbolic and numeric, is based on agents [55]. Each agent executes one rule at a time which is controlled by the `program` function in CASM.

[1] `https://github.com/casm-lang/`

```
1 CASM
2 init test
3
4 function a : -> Integer
5 function b : -> Integer = { 4 }
6 function c : Integer -> Integer = { (0) -> 1, (1) -> 3 }
7 function d : Integer -> Integer defined { 2 } =
8     { (0) -> 1, (1) -> 3 }
9 // <omitted>
```

Listing 4.1: State Initialization

The sequence of operations is identical in each execution. In the beginning, the global function state is initialized with all function definitions in the CASM specification with their respective initialization values. This initialization is lazy in the sense that only functions with initialized values are inserted. In the example in Listing 4.1, the functions `b`, `c` and `d` are in the global state after initialization. `b` is present in the global state with the value `4`. Functions `c` and `d` are both present with the values `1` and `3`, respectively for the indices `0` and `1`. The difference between `c` and `d` is the value of not initialized function values.

During the initialization of the general functions, the `program` function is also initialized. Its value is set with the `init` statement and defines the first rule which an agent should execute. The `program` function does not have to be a null-ary function, but can be an n-ary function with different domains for each dimension [55]. Such an n-ary function instructs the execution environment to create multiple agents.

After the initialization step is completed, the agents execute the rule according to their program function. The agent passes the current rule to its corresponding visitor [36], which is either numeric or symbolic. This depends on whether the execution is defined as symbolic or numeric by the execution pass. That visitor then traverses the CASM-AST of the specified rule and performs the calculation steps defined in the rule. For the most part, symbolic and numeric execution, and therefore their visitors, are the same. They only differ when accessing and updating functions and certain control flow structures such as conditional or iterate rules.

After having completed the execution of the rule, the agents' update-sets are collected, merged and applied to the global state. This update set can also include the `program` function. If it does not include the `program` function, the function remains unchanged and the agent will execute the same rule again in the next step. After merging the update-set into the global state, the agents compute the next step. This continues until either the `program` function of all agents is set to `undef` or the update-set is empty. If either of those conditions are met, the computation is completed.

4.2.1. Symbolic Environment

The symbolic environment represents the TPTP trace during the transformation. Each transformation has exactly one instance of such an environment. Furthermore, it is the single source of truth regarding the TPTP trace. Each addition to the trace is sent to the environment, which handles all the necessary steps to build a valid TPTP model. The symbolic environment has many responsibilities, ranging from creating appropriate function definitions to generating valid formula names and outputting the TPTP formulae in the correct order. It is located in the Intermediate Representation (IR) layer [55] of the compiler so that it can be used for transforming IR code. All functions the symbolic environment provides are performed lazy as to avoid unnecessary memory usage and overhead.

Time Management

As discussed in Section 3.2.1, the notion of time is important in the transformation process from CASM to TPTP. Correct time management is central to the symbolic environment. The environment has an internal time counter which starts at `1` and is updated each time the agents start their next step. So each time a function is updated, the symbolic environment adds the current time to the formula. When accessing a function, however, not the current time is relevant, but the last time the function was updated. In addition to saving the update, the environment also saves the time and location of each update in a hash map. Thus, when accessing a function, it can be checked whether and when the function has been updated. If no update is found, the function is accessed at initialization time (`1`).

Definition Generation

Every time a new symbol is created or a new instruction is executed, the symbolic environment creates the definition for the symbol or functor in TPTP and saves it into a separate list. The separation of definitions and the general execution trace allows all definitions to come first in the generated trace, as discussed in Section 3.1.2. This has two purposes: First, it enhances readability, since all the definitions can be skipped at once. Second, in TPTP the definition must be prior to the usage. Otherwise a type of the functor is assumed to be `($i * $i * ... * $i)> $o` which may be incorrect [16].

While symbols are used only once, instructions may be used multiple times. To avoid having multiple definitions of the same instruction, the symbolic environment keeps track of all generated definitions by their name and creates new definitions only if needed.

4.2.2. Accessing and Updating Functions

Accessing and updating functions works slightly different during symbolic execution in comparison to just numeric execution. In numeric execution, a function access always checks whether a value for the function with the given arguments exists. First, the update-set is searched if the function has already been updated with sequential semantics. Such an update has precedence over the global state [50, 49]. If the function does not exist in the update-set or was updated in a parallel context, the global state is searched. In case a valid value has been found in either of those two steps, it is used for further calculations. If no value has been found, the default value for the function is returned. Usually, that value is `undef` but it can be specified with the `defined` clause during function definition.

If, for instance, in Listing 4.1 the functions `c` or `d` are accessed at `1`, both function would return the value `3`. However, if these functions were to be accessed at `2`, function `c` would have the value `undef` while the function `d` would have the value `2`.

During symbolic execution, numeric values behave the same as all values during numeric execution. Symbolic functions, however, never check the global state as they do not exist in the global state, since updates are never written into the global state. Instead, they create a new symbol and a function access in the TPTP trace using the symbolic environment. This symbol is then passed on to further calculations.

The mechanisms to update numeric and symbolic function are very similar. If a function is updated in numeric execution, the update location, consisting of the function and all the passed argument values, and the value itself are stored in an update-set. During symbolic execution, a symbolic value is added to the update-set and, additionally, a function update is created in the TPTP trace. The presence of the symbol in the update-set serves two purposes: First, it is used if it is repeatedly accessed in a sequential rule. Second, it is used for determining whether an update-clash occurred so that both, numeric and symbolic execution, have the same behavior [50]. When merging update-sets, however, symbolic function updates are ignored.

4.2.3. Conditional Rules

The rules for numeric conditional rules are straight forward. The condition is evaluated and if it is defined and true, the *then* rule is evaluated, otherwise the *else* rule is evaluated. During symbolic execution, and after having evaluated the condition, it is first checked whether the evaluation resulted in a symbol. This happens if, at any point in the condition, a symbolic function is referenced. If the value of the condition is numeric, the same procedure, as mentioned beforehand, is applied. However, if it is a symbolic value, both branches are evaluated. The symbolic environment is instructed to create a new path condition from the condition symbol. This path condition prepends all further TPTP

formula, added to the trace, with the condition symbol until the path condition is destructed again. With the path condition in place the *then* rule is evaluated. After the evaluation the path condition for the *then* rule is destructed and another path condition with the inverted condition is created for the *else* rule which is evaluated afterwards.

4.2.4. Iterate Rules

Iterate rules during numeric execution are defined as repeating the sub-rule as long as the update-set produced by the sub-rule is not empty. Iterate rules where the loop count depends on symbolic functions, are not supported. Therefore, this case is not checked and the symbolic execution is identical to numeric execution. If the loop count depends on symbolic functions, the symbolic execution will run that rule indefinitely. This is because symbolic calculations always create updates as they only create new symbols.

4.2.5. Calculation Instructions

In the CASM compiler, calculations are performed using instructions [55]. Those instructions are defined in the IR and work with IR constants. Symbolic values are a kind of IR constants. So if a instruction encounters a symbolic value, it creates a new symbol and generates the corresponding TPTP formulae. This new symbol is then returned as the result of the calculation. There are two kinds of formulae which are generated. These are the general instruction functors and specialized formulae which use TPTP defined functors.

General instructions are all written in the form: `#<function>#<type-info>(<p1>, <p2>, <result>)`. An example for such an instruction is the *add* instruction. Its name and declaration is generated by the symbolic environment when the instruction is used for the first time, however, its behavior is not yet defined.

The specialized formulae make use of TPTP defined functors. They do not have a fixed scheme as general instructions and are mostly used for comparisons. However, they include the result symbol in such a way, that it contains the result of the instruction.

In Listing 4.2, there are two examples of such specialized formulae. In both cases the resulting symbol (`res`) must be equivalent to the return value of the TPTP defined functor. These functors have fixed semantics and are polymorphic over some types [63].

```
1 tff(1,hypothesis,res<=>(p1=p2)). % uses '=' infix operator
2 tff(2,hypothesis,res<=>($less(p1,p2))). % uses '$less' functor
```

Listing 4.2: Symbolic Instructions: Equivalence and Less

For all types or instructions which are not supported by the defined functors, the general

approach must be used.

4.2.6. Let Rules

Let assignments in the numeric execution are handled by storing the IR constant at a predetermined index in the rule frame. Each time the variable is accessed, the constant at the specified index is retrieved and used. The symbolic execution works in the same way. As discussed before, symbols are a kind of IR constant, so they can be stored at the same index the value is usually saved. While retrieving the value, the symbol is passed on and used for calculations. During this entire procedure, no interaction with the TPTP trace takes place, so that in the TPTP trace formulations with let rules are indistinguishable to formulations without let rules.

4.3. Symbolic Consistency Pass

As discussed in Section 3.3, the symbolic consistency pass checks if any function updates produce a symbolic conflict. All other information like types or actual values are ignored, as they are not important for the symbolic status.

Each function, rule parameter or expression can have one of three statuses, `SYMBOLIC`, `NUMERIC` or `UNKNOWN`. At the beginning, all functions are either `SYMBOLIC` or `NUMERIC`, depending on their `symbolic` attribute. All parameters, expressions and rule return values are initially `UNKNOWN`.

For an unknown parameter to become symbolic, it only needs to be passed a symbolic value, same as return values and expressions. To be set to numeric, however, a parameter must never be assigned a symbolic value and all calls to its rule must be found. CASM instructions result in a numeric value if none of the arguments is symbolic or unknown and the expression has the symbolic status of the instruction result. Return values have the symbolic status of their expression.

A rule is in one of four stages during the evaluation: *init*, *started*, *evaluated* and *finished*. If it is encountered during execution, it has the status *init* as it has not yet been evaluated in any form, but is part of the Rule-Call-Graph (RCG). When the evaluation of a rule starts, the status is *started*, indicating, that it has started evaluation. In case the rule evaluation is aborted due to an unknown conditional rule, it remains in the status *started*. If a rule finishes evaluation, but has an unknown return value or any unknown function updates, as it might be dependent on a as of yet unknown parameter, the rule is marked as *evaluated*. The status *evaluated* means that a rule must be evaluated again after all parameters have been defined `SYMBOLIC` or `NUMERIC`, but all calls to other rules have been recorded. In a *finished* rule all updates and the return value are resolved to either symbolic

or numeric.

4.3.1. Building the Rule-Call-Graph

The Rule-Call-Graph (RCG) is created step-by-step. At the beginning, the RCG consists only of the rules defined by the `init` statement. From all the rules in the RCG, a rule is chosen for evaluation. The rule selection process is described in detail in Section 4.3.2.

If another rule is directly called during the evaluation, it is added to the RCG. In case it is not yet part of the graph, it is added with the status *init* and all parameters and return value are set to `UNKNOWN`. Every rule has a reference to all rules by which it s called and a reference to all rules which it calls.

After the rule has been added to the RCG, the arguments are resolved. All arguments which are symbolic are marked as symbolic. All numeric arguments remain unknown, since they may only be marked as numeric after all rule calls have been recorded.

To check whether all rule calls have been recorded, the status of all rules in the RCG is checked after each rule evaluation. If all rules are either in the state of *evaluated* or *finished*, all rule calls have been recorded, so all unknown arguments are now set to be numeric.

4.3.2. Choosing a Rule

In order to evaluate the next rule, all rules in the RCG are checked to see whether they meet the criteria for evaluation. The earlier a rule is in its evaluation state, the higher its priority. A rule in the *init* state therefore has priority over a rule in the *started* state which, in turn, has priority over a rule in the *evaluated* state. A secondary criteria is the status of the arguments of each rule. A rule which has all arguments defined as either symbolic or numeric has precedence over a rule in the same state with unknown arguments.

The rules in Listing 4.3 serve as an example. The dependencies of the rules are not important for the selection, therefore they are omitted. The order of evaluation would be `b`, `a`, `c`, `f`, `d`. Since rule `f` has already finished, it is omitted.

```
1 rule a(UNKNOWN, SYMBOLIC) -> UNKNOWN : init
2 rule b() -> UNKNOWN                  : init
3 rule c(UNKNOWN, SYMBOLIC) -> UNKNOWN : started
4 rule d(UNKNOWN) -> NUMERIC           : evaluated
5 rule e() -> NUMERIC                  : finished
6 rule f(SYMBOLIC) -> UNKNOWN          : evaluated
```

Listing 4.3: Rule Choose Order

4.3.3. Aborting Rule Evaluation

In the `SymbolicConsistencyPass` there are two cases, in which the evaluation of a rule must be aborted. The first case, as briefly discussed in Section 3.3.3, concerns conditional rules. If the condition expression in the conditional rule is `UNKNOWN`, the correct symbolic context cannot be created. In this case the evaluation of the top-level rule is aborted. The second case refers to the encounter of a symbolic conflict.

In either case, execution is aborted by throwing an exception, which quickly unwinds the stack built by the visitor pattern. The exception thrown by the conditional rule is caught by the code which schedules the evaluation of rules. If such an exception is encountered, a new rule is evaluated.

The symbolic conflict exception is caught by the code which starts the visitor. When such an exception is caught, the offending function is set to be symbolic and the entire visitor is restarted.

4.3.4. Built-in Rules

CASM has a few built-in rules which may be called during execution. Each built-in rule has a fixed behavior regarding symbolic parameters, which is not dependent on the CASM specification, but is constant.

In the `SymbolicExecutionPass` this is handled by a table lookup. The lookup is implemented using a hash map, that has the rule name as key. The corresponding value is a lambda function which takes an array of symbolic statuses and returns a symbolic status. The arguments to the lambda function are the symbolic status of the arguments passed to the built-in rule. The return value of the lambda function is the symbolic status of the value which the built-in rule returns.

Most built-in functions have the same behavior as instructions, returning `SYMBOLIC` if any of the arguments are `SYMBOLIC`. But there are rules like `print` with return type `Void` or the method `symbolic` which returns a `NUMERIC` value in any case. Rules with the return type `Void` always return `NUMERIC` values in the context of symbolic consistency. That way, these rules do not need to be handled specially.

4.3.5. Example Symbolic Consistency

In the following section, the detailed sequence of operations in the `SymbolicConsistencyPass` is illustrated using an example. The CASM specification is shown in Listing 4.4.

At the start of the analysis, only the rule `test` is considered as called within the specification, as it is the initial rule. Therefore, the RCG only consists of the rule `test` (Figure 4.1). Since no other rule is available, the rule `test` is analyzed next. The call

to `callable` with the arguments `(1, 0)`(Line 12) is registered in the RCG (Figure 4.2) with two numeric arguments and the analysis continues with Line 12. In that statement `callable` is called with `(a, 1)` and is therefore registered as a call with the first argument being symbolic and the second argument numeric (Figure 4.3). The next statement to be evaluated is in Line 14. Since the function `b` is numeric, the expression `b + 1` is numeric. This expression is the update of the numeric function `d`, which remains numeric. The rule `test` is now fully analyzed.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  function b : -> Integer
7  function c : -> Integer
8  function d : -> Integer
9  function e : -> Integer
10
11 rule test = {
12     callable(1, 0)
13     callable(a, 1)
14     d := b + 1
15 }
16
17 rule callable(param1 : Integer, param2 : Integer) = {
18     b := param1
19     c := param2
20 }
21
22 rule notCalled = {
23     e := a + 1
24 }
```

Listing 4.4: Symbolic Consistency Pass: Example

The next rule to be evaluated is now `callable` as it is the only remaining function that has not yet been evaluated in the RCG (Figure 4.4). It is called with the first argument that is symbolic. The second argument is still unknown, as not all rules are evaluated at least once. For example, the `callable` rule could call itself recursively with a symbolic value as second parameter, which would make that second parameter symbolic. The statement in Line 18 is analyzed next. Since `param1` is symbolic and `b` is numeric, a conflict occurs. Therfore, function `b` is promoted to symbolic and the consistency check is restarted.

The check starts again at rule `test`. The calls to `callable` (Line 12-13) are identical to the first time, producing the same RCG (Figure 4.3). Line 14, however, differs as `b` is now symbolic and the expression `b + 1` is now also symbolic. This is the update to numeric function `d`, which in turn is promoted to symbolic.

In the third check, the `test` rule is analyzed as before, with the exception that `d` has

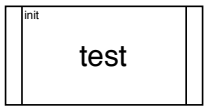

Figure 4.1.: RCG: Init

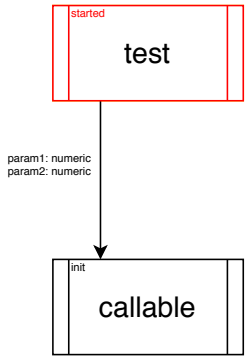

Figure 4.2.: RCG: First Call

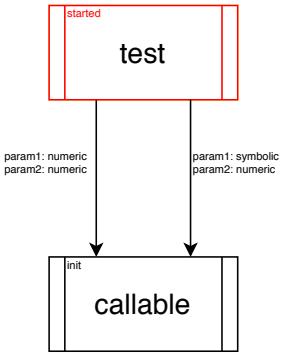

Figure 4.3.: RCG: Second Call

already been set to symbolic so that no conflict occurs. Next, the rule `callable` is analyzed (Figure 4.4). As it was in the first pass, `param1` is symbolic and `param2` is unknown. The next analyzed statement is in Line 18. Both functions `param1` and `b` are symbolic, so the update does not produce a conflict.

In Line 19 there is no change in the symbolic status of `c` due to the unknown status of the `param2` function. Although the rule `callable` has been fully evaluated, an unknown update remains. Therefore, the rule is not yet completed which leads to the RCG in Figure 4.5. Now all the rules in the RCG have been processed at least once. This means that the specification no longer contains any unregistered rule calls. At this point, all parameters of rule calls, which have been unknown up until now (`param2` of rule `callable`) are set to numeric values.

The next rule to be analyzed is `callable`, as it is the only rule that has not completely been evaluated. The symbolic status of `b` and `param1` in Line 18 is still symbolic and therefore identical to the last evaluation. Function `c` in Line 19 is updated with the `param2` function. However, `param2` is now numeric and therefore, `c` is updated with a numeric value. Updating a numeric function with a numeric value is possible, so `c` remains numeric. Rule `callable` is now fully analyzed, leading to the RCG in Figure 4.6.

As there are no further rules to be analyzed, the consistency check is complete. If rule `notCalled` is analyzed, the update in Line 23 would be a symbolic conflict. However, as the rule is never called, it is not evaluated. Therefore function `e` remains numeric.

The result of the symbolic consistency check is depicted in Listing 4.5. The functions `b` and `d` are promoted to symbolic and the functions `c` and `e` remain numeric.

```
1 [symbolic] function a : -> Integer
2 [symbolic] function b : -> Integer
3 function c : -> Integer
4 [symbolic] function d : -> Integer
5 function e : -> Integer
```

Listing 4.5: Symbolic Consistency Pass: Example – Functions after Analysis

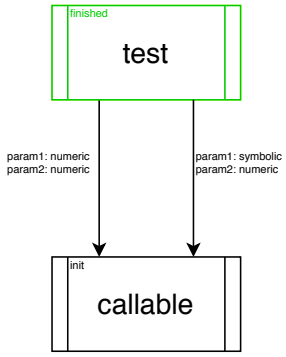

Figure 4.4.: RCG: `test` Finished

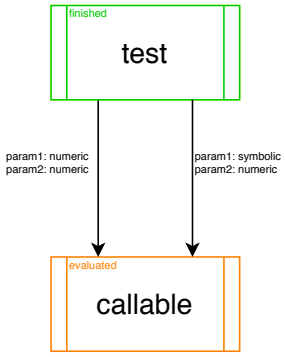

Figure 4.5.: RCG: Evaluated `callable`

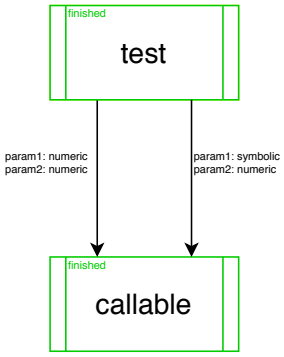

Figure 4.6.: RCG: Finished

4.4. TPTP Library

As mentioned in Section 3.1.1, the TPTP framework represents its models similar to the TPTP grammar. A TPTP file can therefore be transformed into the model without complicated transformations and vice versa. For that reason and due to the fact that a TPTP-AST grows complex very fast, the previous chapters showed TPTP files, despite the fact that transformations generate TPTP models and not files.

The TPTP language posses some form of hierarchy in its formulas. The outermost level is the *logic level*, which knows only the `$o` type and represents the propositional logic parts of the language. The CNF formulae consist mostly of elements from this level. The next level is the *term level*. In this level are term structures like conditional terms or let terms. The third level is the *atom level*. At this level there are atomic structures such as literals, identifiers, tuples. TPTP further divides the atomic level into its plain, defined and system atoms. Plain atoms are free to use by the problem creator. The defined atoms are reserved for language defined elements such as the `$sum` functor or the `$true` value. The system atoms are system specific elements. An ATP system could use system atoms to define theories like arrays or the like. Problems which use such elements are generally not portable anymore. The framework only uses one atom class, but provides a property which states the type of the atom. The last layer is the *type layer*, in which types are specified. As the layers are hierarchical, a type is also an atom, which is also a term, which is also a logic element. This allows for a logical formula to include an identifier as an argument to a logic operand like *and*. In the framework of this thesis, this hierarchical structure is achieved through inheritance.

A shortened class diagram can be seen in Figure 4.7. The complete class diagram can be seen in the appendix (A.1). It only includes the abstract base class of every TPTP layer type and some of the auxiliary classes. The base class of any AST element is the

`Node` class. The `GeneralTerm` and `Annotation` classes model the additional info, which can be added to each formula. The `Token` class represents any token in the AST. It has a factory class `TokenBuilder` which contains static methods to generate any token which occurs in the TPTP grammar. The `Definition` and `Formula` subclasses are used to store the formulae. One of the `Definition` subclasses handles the *import* rule of TPTP. This rule allows the inclusion of additional TPTP files. This is often used to provide the same axioms to several different problems. It is currently not used in the transformation of this thesis. The other subclass stores the tokens and a `Formula` class of a formula statement. The `Formula` class contains the `Logic` object and optionally an `Annotation` object.

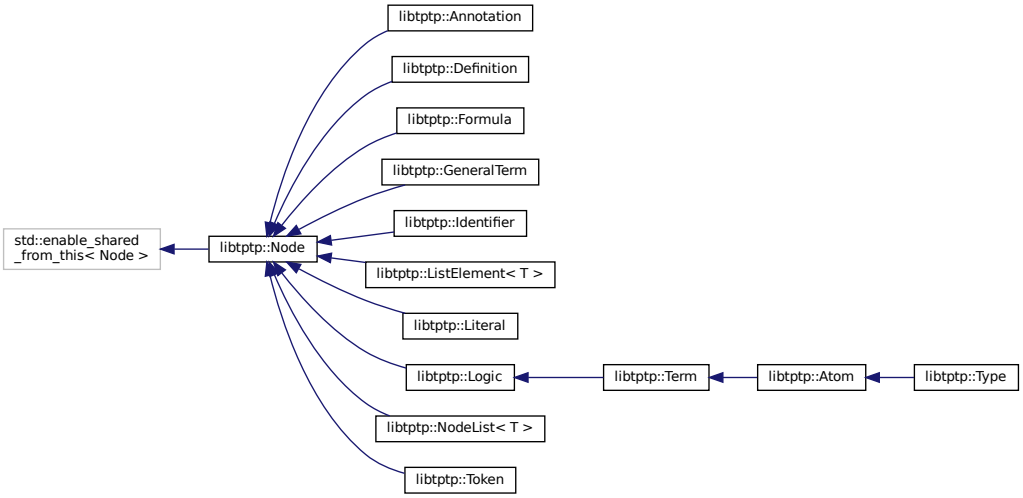

Figure 4.7.: TPTP Framework: Reduced Class Diagram of Node

The information which can be attached to any formula is called *useful information* in TPTP. The *useful information* field has no defined meaning and can be used by an ATP to append some information. Usually the information includes some form of why any formula was generated in a solution output, for example. In the context of this thesis, source location of the original CASM file could be added during the transformation to the TPTP model to generate output in CASM terms.

Visitors are used to manipulate the AST. The currently implemented visitors and their dependencies can be seen in Figure 4.8. The `Visitor` class is an abstract base class for all other visitors and only defines the interface. The `RecursiveVisitor` traverses the AST, but has no other functionality. The class is not abstract, so any visitor which is a subclass does not need to implement methods for AST nodes which have no special function. An example

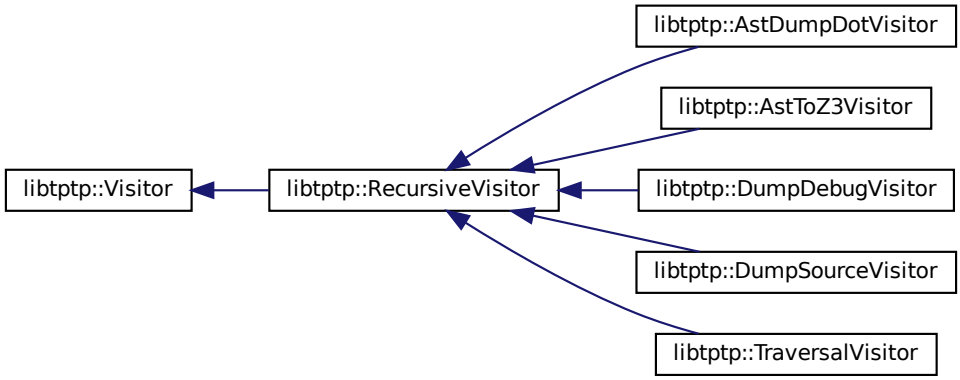

Figure 4.8.: TPTP Framework: Reduced Class Diagram of Visitors

for this is the `DumpSourceVisitor` as it only needs to implement methods for the Literal, Identifier and Token classes to dump the whole AST to a file. The `AstDumpDotVisitor` and `DumpDebugVisitor` are only used for debugging purposes, as the former dumps the whole AST as GraphViz [37] dot graph and the latter dumps only the node types which are present in the AST. The `TraversalVisitor` traverses the whole AST and calls a `callback` function with the current node either in preorder or postorder. Inorder is not supported in the visitor, as most AST nodes have multiple child nodes and inorder sequence is therefore not defined. However, it is currently unused. The `AstToZ3Visitor` transforms the AST into a model for the SMT solver Z3. Only FOF and TFF formulae are currently supported.

4.4.1. Transformation to SMT Solver

The transformation from the TPTP to the Z3 model takes place in the `AstToZ3Visitor`. The visitor traverses the AST and generates the Z3 objects using the C++ API of Z3. Currently only the FOF and TFF parts of TPTP can be translated to Z3. While most of the transformation is straight forward, there are a few edge cases.

Z3 does not support polymorphic sorts, but TPTP does. Therefore, the transformation pass must bridge that gap. For this purpose, lazy sort instances are created for each interpretation of the parametric sort. In quantified formulae where the quantifier is over all sorts, this ad hoc realization is currently not possible. Therefore, quantified formulae over sorts are not supported.

TPTP enables the differentiation of formulae by type, however, Z3 does not support that,

as it is largely unimportant. Therefore, all formula types are ignored, except `conjecture` and `negated_conjecture`. Those two types are used to indicate formulae which must be true (*universal* quantifier) for all interpretations of all constants. Unbound constants in the Z3 model, however, have an implicit *exists* quantifier. This means that Z3 returns *satisfiable* if it can find any interpretation of unbound constants in which all formulae are true. To get the desired result, `conjecture`s are negated, and if any model can be found (there exists an interpretation, in which the inverted conjecture is satisfied), the non inverted conjecture is not satisfied. In case the negated conjecture cannot be satisfied in any interpretation, it follows that the non negated conjecture is true for all interpretations.

Quantified logic in TPTP is directly translated to `forall` and `exists` constructs in Z3 respectively. This enables the use of quantified formulae with infinite types like integer or real. Whether or not a solution can be found therefore depends entirely on the capabilities of Z3 and not on the transformation.

5. Evaluation and Discussion

This chapter discusses the evaluation of the framework created. First, it presents the types of tests that have been performed on the different parts of the framework and, second, it examines the performance impact of the symbolic execution on the numeric CASM interpreter. A direct comparison with the implementation of Lezuo [48] is not possible as this implementation is proprietary.

5.1. TPTP Framework

In order to validate the TPTP framework itself, testcases were created which parse problems from the TPTP library and dump them again. The dumped output is then compared to the original input problem. As the TPTP framework does not preserve whitespaces or comments, these have to be excluded in the comparison.

The selected problems are chosen randomly from the TPTP library with the goal of covering different parts of the TPTP syntax. There are at least two problems for each type of formula. One criterion for the test selection is that at least some problems have a high formula depth. Other criteria are that all numeric types occur at least once, all types of general data in the useful information field are present, quantifier formula exist and problem defined predicates and types are present.

These parse tests uncovered several bugs in the TPTP framework, in the parser, in the dumping pass and in the underlying CASM compiler standard library. These include parsing errors in the Decimal datatype and null pointer errors. A total of 13 TPTP problems are used for these tests, of which 8 run successfully. Two of those not working are due to the bug in the Decimal datatype and the other three are because of an unimplemented type in the THF part of the language which is currently unused.

The tests show that with the exception of a THF type, the whole TPTP language can be correctly parsed and stored in an in-memory model. Dumping an in-memory model to a TPTP file also works as specified, except for semantically irrelevant data such as whitespaces or comments. These cannot be correctly printed, as they are not stored in the in-memory model.

5.2. Z3 Transformation

In order to test the Z3 transformation, small (one to five formulae) TPTP traces are used which cover the supported features of the transformation. The focus of the testcases, however, is on TFF formulae, as they are used in the CASM to TPTP transformation. The traces are small because the correct transformation from TPTP to the Z3 model is central and not the execution speed of Z3. The testcases cover basic FOF and TFF formulae, quantified TFF formulae, predicate functions and polymorphic types.

The testcases are parsed, transformed to Z3 and solved using Z3. However, the models which Z3 outputs, cannot be checked for correctness, as Z3 outputs its models in the SMT lib 2.0 language, for which no parser is implemented. A text based comparison is fragile, as any change in Z3 could result in an equally valid but different model output which would falsify the tests. Therefore, it is only checked whether or not the trace is satisfiable by Z3.

The results show that the TPTP traces are correctly translated into the Z3 model. To verify that the transformation is correct, the solutions from Z3 have been manually validated. However, the transformation is limited to first order formulae. As only first order formulae are needed for the CASM models, this limitation is acceptable.

5.3. CASM to TPTP Transformation

The tests for the transformation of CASM to TPTP are similar to the tests for the TPTP framework. A small CASM code is parsed and transformed to TPTP, and then compared to an expected TPTP trace. The expected trace was created by the first run of the CASM to TPTP transformation of the CASM code in question and manually validated.

The testsuite of the transformation pass covers the basic elements of CASM. These include integer arithmetic, boolean algebra, let rules, sequential rules, conditional rules and rule calls.

In this process the Z3 model is not used. In this way, TPTP traces can be created even if the linking to Z3 does not work or Z3 is unavailable or there has been a version change.

Currently, the transformation uses the pseudo boolean type `$o` as its boolean type, as all operators of the boolean algebra are already defined. This means that the produced traces do not completely comply with the TPTP standard in case a predicate with a boolean parameter exists in the trace. However, similar to the Z3 TPTP parser, this thesis treats the `$o` type as a complete type. This is due to the fact that while a boolean type could be defined in TPTP, the overhead that is added to the traces is rather big since all boolean operands must also be defined. Furthermore, Z3 cannot use its internal boolean type to accelerate the calculation.

5.3.1. Limitations

Integration tests of CASM code to Z3 are currently not useful, as, on the one hand, the semantics of the function and instruction predicates in the TPTP traces are not yet defined. Therefore, any formula is satisfiable, if the predicates are interpreted as true regardless of the input values. On the other hand, there is no way to add conjectures in the CASM specification, and translation validation constraints are only useful when there are models at two different optimization points, which is currently not the case.

Furthermore, loops that are dependent on symbolic variables to determine their loop count are not supported. Also, the symbolic choose rule is not working, as there is currently no syntax in CASM that allows to choose a symbolic value from a numeric defined range.

5.3.2. Addition Case Study

A test case is presented as an example to illustrate the validation process. The analysis is done in the same manner for each testcase in the test suite. The CASM specification in Listing 5.1 produces the TPTP trace in Listing 5.2. All formulae are in TFF form, which is different from the implementation by Lezuo that mixed TFF and FOF forms.

```
1  CASM
2
3  init test
4
5  [symbolic] function a : -> Integer
6  [symbolic] function b : -> Integer
7  [symbolic] function c : -> Integer
8
9  rule test =
10 {
11     c := a + b
12     program( self ) := undef
13 }
```

Listing 5.1: CASM: Addition Program

However, as already mentioned, the definitions for the semantics of the function and instruction predicates are missing, making it unfit for solving. All symbols which occur in the trace are correctly defined at the start of the trace. Half of those symbols in the trace result from meeting the condition that time zero is the last step of the ASM. Therefore, the number of symbols used can be reduced by reusing symbols. Similarly, all definitions for function predicates and instruction predicates are present and the control flow corresponds to the CASM specification. Lastly, the value of all functions in the last ASM step is set to be at time zero, as per specification. Therefore, the transformation pass produced a correct TPTP trace. Once the produced trace is validated, it is used as the expected value

for the output and automatically tested.

```
% specification:
tff(3,type,'%0':$int).
tff(5,type,'%1':$int).
tff(7,type,'%2':$int).
tff(11,type,'%3':$int).
tff(14,type,'%4':$int).
tff(17,type,'%5':$int).
tff(8,hypothesis,'#add#i':($int*$int*$int)>$o).
tff(0,hypothesis,'@a':($int*$int)>$o).
tff(1,hypothesis,'@b':($int*$int)>$o).
tff(2,hypothesis,'@c':($int*$int)>$o).
tff(4,hypothesis,'@a'(1,'%0')).
tff(6,hypothesis,'@b'(1,'%1')).
tff(9,hypothesis,'#add#i'('%0','%1','%2')).
tff(10,hypothesis,'@c'(2,'%2')).
tff(12,hypothesis,'@a'(1,'%3')).
tff(13,hypothesis,'@a'(0,'%3')).
tff(15,hypothesis,'@b'(1,'%4')).
tff(16,hypothesis,'@b'(0,'%4')).
tff(18,hypothesis,'@c'(2,'%5')).
tff(19,hypothesis,'@c'(0,'%5')).
$
```

Listing 5.2: TPTP: Addition Trace

As the transformation is instruction based and therefore mostly ignores rule boundaries, the CASM specifications in Listing 5.4 and Listing 5.3 produce exactly the same trace as in Listing 5.1.

```
CASM
init test

[symbolic] function a : -> Integer
[symbolic] function b : -> Integer
[symbolic] function c : -> Integer

rule test =
{
    assign( a + b )
    program( self ) := undef
}
rule assign( d: Integer ) =
{
    c := d
}
```

Listing 5.3: CASM: Addition Program with Let

A direct comparison with AsmToSMT [5] and ASM2Bogor [61] is difficult as AsmToSMT is currently not working[1] and the ASMs presented in both papers rely on choose rules.

[1] https://github.com/asmeta/asmeta/issues/17

However, as the trace in this thesis is instruction based, it has more symbols and is therefore generally longer than AsmToSMT and ASM2Bogor. How and to what extent this longer trace affects the solving time is not yet clear and needs to be evaluated further.

```
1  CASM
2  init test
3
4  [symbolic] function a : -> Integer
5  [symbolic] function b : -> Integer
6  [symbolic] function c : -> Integer
7
8  rule test =
9  {
10     let d = a + b in
11         c := d
12     program( self ) := undef
13 }
```

Listing 5.4: CASM: Addition Program with Let

5.3.3. Conditional Rules

One significant improvement of this implementation compared to the previous implementation by Lezuo is that with conditional rules, the trace generation is not forked. Instead, both branches with their respecitve path conditions are in the same trace. The CASM specification in Listing 5.5 transforms to the TPTP trace in Listing 5.6. The condition is represented by the `'%1'` symbol and both updates to `b` are present, guarded by their respective condition. A longer CASM specification with nested conditional rules can be found in the appendix (A.1, A.2)

```
1  CASM
2  init test
3
4  [symbolic] function a : -> Integer
5  [symbolic] function b : -> Integer
6
7  rule test =
8  {
9      if a = 0 then
10         b := 1
11     else
12         b := 2
13     program( self ) := undef
14 }
```

Listing 5.5: CASM: Conditional Rules

```
% specification:
tff(2,type,'%0':$int).
tff(4,type,'%1':$o).
tff(8,type,'%2':$int).
tff(11,type,'%3':$int).
tff(0,hypothesis,'@a':($int*$int)>$o).
tff(1,hypothesis,'@b':($int*$int)>$o).
tff(3,hypothesis,'@a'(1,'%0')).
tff(5,hypothesis,'%1'<=>('%0'=0)).
tff(6,hypothesis,'%1'=>('@b'(2,1))).
tff(7,hypothesis,~'%1'=>('@b'(2,2))).
tff(9,hypothesis,'@a'(1,'%2')).
tff(10,hypothesis,'@a'(0,'%2')).
tff(12,hypothesis,'@b'(2,'%3')).
tff(13,hypothesis,'@b'(0,'%3')).
```

Listing 5.6: TPTP: Conditional Rules

Similar to the addition example, all symbols and function predicates are defined with their correct types. The condition of the Conditional rule is present and the sub-rules of the then and else branch are correctly prepended by the path condition.

To summarize the results of these tests: the transformation pass correctly transforms the supported CASM specifications according to the specification by Lezuo [48]. The fully supported CASM rules, however, are limited to update rules, parallel and sequential block rules, let rules and conditional rules. Other rules, like iterate, choose and forall rules are partially supported when used with numeric functions.

5.4. Symbolic Consistency

The symbolic consistency algorithm has been discussed in detail in Section 4.3. To validate its results, testcases which cover most of the CASM capabilities have been created manually. All test cases start with only one symbolic function, and this symbolic status is propagated through CASM expressions.

One of these tests can be seen in Listing 5.7. Function `b` and `c` get set to symbolic, as at least one of their element is symbolic. Function `h` and `d` are promoted to symbolic. `h` is symbolic because `x` is a range with an symbolic cardinality. `d` is symbolic because whether the value ten (10) is in the range from `0` to `a` is also determined by the value of `a`. Functions `e`, `f`, `g` and `i` all remain numeric, as their updates consist only of numeric values. Function `j` is symbolic because the condition of the `holds` clause depends on the symbolic variable `x`. Function `k`, however, stays numeric, as the `holds` clause is independent of the symbolic variable. The function `l` also remains numeric because `x` is chosen from a numeric range. When running the test, the symbolic consistency pass is invoked and afterwards the symbolic status of the functions is compared with the expected values.

```
CASM
init test

[symbolic] function a : -> Integer
function b : -> (Integer,Integer)
function c : -> List< Integer >
function d : -> Boolean
function e : -> (Integer, Integer)
function f : -> List< Integer >
function g : -> Integer'[0..10]
function h : -> Integer
function i : -> Integer
function j : -> Boolean
function k : -> Boolean
function l : -> Integer

rule test =
{
    b := (a, 1)
    c := [a, a, 1, 2, 3]
    let x = [0..a] in
        h := |x|
    d := exists x in [0..a] with x = 10
    e := (0, 1)
    f := [4, 5, 1, 2, 3]
    g := 5
    let x = [0..10] in
        i := |x|
    j := forall x in [0..a] holds x < 5
    k := forall x in [0..a] holds true
    choose x in [0..10] do
        l := x
    program( self ) := undef
}
```

Listing 5.7: Symbolic Consistency Test

When creating the testcases for the symbolic consistency pass, a CASM specification with the syntax to be tested is created and then the expected symbolic status of all functions is determined manually. The expected values for the symbolic status in the testcases are determined analogous to the previous example, by examining all update rules and by checking whether the expression is symbolic.

However, as the generated testcases are synthetic and evaluated manually, their validity is limited. They are short because each test checks specific syntactic constructs. Their length also prevents conclusive statements about the run time performance, as the runtime scales with the size of the specification.

Despite aforementioned limitations, the testcases show that the symbolic consistency pass correctly updates the symbolic status of functions, which would produce symbolic conflicts. They also show that functions which are used only numerically, remain numeric. ASMs which would run indefinitely, as they contain infinite recursion and infinite loops,

are calculated instantly. This is facilitated by the rule-by-rule approach of the consistency check.

5.5. Execution Speed Comparison of the Numeric Interpreter

The CASM compiler has an interpreter, which runs the interpretation of the ASM on the CASM-AST, without compiling it. As the symbolic execution framework supports concolic execution, some changes have been made to the numeric execution pass of the interpreter. To assess whether these changes affect normal numeric execution, the execution times of some benchmark problems of the CASM interpreter without the symbolic execution framework are compared to the execution times of the interpreter with the symbolic execution framework. The tested problems are a bubblesort, a fibonacci, a convay game of life implementation, array accesses in nested loops (sieve) and a program with many pseudo-states (updateset)[2]. The time measurement has been done using the hyperfine[3] tool on an Arch Linux [4] machine. Each program is run at least ten times for accurate results.

The results can be seen in Table 5.1. The left column contains the result without the symbolic execution framework, the right column contains symbolic execution framework. The measured runtimes are within the σ distance from each other, therefore there is no measurable difference.

Specification	$t_{no\,sym}$(s)	$t_{with\,sym}$(s)
Bubblesort	$(89.1 \pm 15.6) \cdot 10^{-3}$	$(97.0 \pm 17.1) \cdot 10^{-3}$
Bubblesort large	112.771 ± 5.203	107.922 ± 1.204
Fibonacci	$(253.6 \pm 11.1) \cdot 10^{-3}$	$(253.0 \pm 13.0) \cdot 10^{-3}$
Convay	39.598 ± 1.298	41.365 ± 1.640
Sieve	$(810.9 \pm 13.7) \cdot 10^{-3}$	$(866.5 \pm 52.2) \cdot 10^{-3}$
Updateset	167.979 ± 6.591	176.698 ± 10.430

Table 5.1.: Results of Numeric Execution Time of CASM Interpreter

[2] `https://github.com/casm-lang/libcasm-tc/tree/master/benchmark`
[3] `https://github.com/sharkdp/hyperfine`
[4] uname -a: Linux 5.10.4-arch2-1 #1 SMP PREEMPT Fri, 01 Jan 2021 05:29:53 +0000 x86_64 GNU/Linux, Intel(R) Core(TM) i7-4810MQ CPU @ 2.80GHz

6. Related Work

This section provides an overview of the related work in the field of symbolic execution with a focus on the symbolic execution of ASMs. First, different languages for execution traces are discussed, followed by an overview of satisfyability modulus theory (SMT) solvers. Finally, various symbolic execution implementations are presented.

6.1. Modeling Language

This thesis uses the TPTP language as the modeling language for the symbolic traces [63]. It is a general purpose language for ATP problems and is supported by many theorem provers. This makes it easy to switch theorem provers. A more detailed description of TPTP can be found in Section 2.3.

The SMT lib 2.0 [8, 11] is similar to TPTP. Satisfiablity Modulus Theories (SMT) is an area of problems related to the determination of the satisfyability of first order formulae with regards to some background theories [2]. These background theories do not need to be formulated in first order form. An example for such a background theory is the theory of natural numbers [10]. The SMT lib 2.0 provides a language for expressing problems and benchmarks for SMT solvers [9]. It is developed with considering the field of software verification. As such, it contains many theories relevant to software verification like arrays, bitfields, strings, and numbers.

6.2. SMT Solvers

In this thesis, the generation of proofs is deferred to established SMT solvers. The default solver for this thesis is Z3 [27]. Z3 is an efficient SMT solver developed by Microsoft Research. It is able to parse SMT lib 2.0 and the FOF and TFF parts of TPTP. Additionally, Z3 has API interfaces for different languages. The most important interface for this thesis is the C++ interface as it allows a direct transformation from the internal TPTP model of the compiler to the model of Z3.

Alternative state-of-the-art SMT solvers include MathSat [20], Beagle [12], CVC4 [7], Yices [30], Barcelogic SMT solver [17], OpenSMT [21] and Vampire [47]. In comparison, Z3 has similar performance to CVC4, Yices and MathSat [72, 34]. However, only Beagle,

Z3 and Vampire support inputs in TPTP format. Compared to the latter options, Z3 is the most common choice as it has a wide variety of problems that it can solve. In addition, it is licensed under the MIT license, which enables it to be easily integrated into the CASM compiler. Therefore it is chosen as the default solver.

The field of SMT solvers is rapidly developing with many advancements every year. While there are already powerful methods for solving satisfyability in propositional logic or quantifier-free linear real and integer arithmetic, solvers for checking logic with quantifier-free non-linear real and integer arithmetic are still under active development [1].

6.3. Symbolic Execution Implementations

A big variety of symbolic execution implementations already exist. Most prominently are KLEE [22], S2E [24], DART [40], SAGE [41] and PEX [70]. All of those symbolic execution engines are built with a specific goal in mind and are therefore tailored towards these needs [29]. KLEE, S2E and DART, for example, are designed to generate high coverage test cases [71]. For fast runtimes, these tools use concolic execution. KLEE and DART are using dynamic symbolic execution (DSE), while S2E uses selective symbolic execution. SAGE and Angr [62] are tailored towards binary analysis and finding security vulnerabilities. Other symbolic execution implementations are restricted to some languages, for example, PEX can only execute .Net code and Symbolic PathFinder (SPF) [52] can only execute Java byetcode.

In the field of ASMs, different approaches of symbolic execution have been explored. Most of these symbolic execution implementations are designed towards model checking.

Gargatini et al. [38] use symbolic execution to generate high coverage test cases. They transform an ASM specification into PROMELA, a language for the model checker Spin [44]. Functions in ASM are translated into PROMELA variable pairs. One variable is for the current value, the other for the value in the next step. However, only null-ary ASM functions are supported. Rules are transformed into a "flattened" form of guarded updates. Test cases are generated from this specification in order to obtain the best possible coverage.

Del Castillo and Winter [28, 74, 73] built tools into transform ASMs from the ASM Workbench into a model for the SMV model checker. Their tools support n-ary functions with a finite location space. Their transformation first translates the ASM into an ASM_0 form, which has only null-ary functions, and then translates that ASM_0 into the SMV language by "flattening" the transition rules to guarded update statements. The model checks are added in *Computation Tree Logic* (CTL) [26] form.

Kardos [45] presents a model checker for the AsmL language. It explores the state space

of the ASM and checks the model properties on the fly. The model properties are written in CTL* form, a superset of CTL and *Linear Temporal Logic* (LTL) [59].

Tang et al. [69] propose a model check framework based on *Answer Set Programming* (ASP). The ASM, written in ASM-SL, is transformed into an ASP program for k ASM steps. Model checks are written in LTL form and are also transformed into an ASP program. An answer set is calculated using a bounded model checker. Similar to this thesis, ASM functions are represented as predicates with a time argument.

Farahbod et al. [33] built a model checker for the CoreASM language [32]. CoreASMs model checking supports distributed ASMs, arbitrary n-ary functions and all ASM rules except import and extend, as these could potentially create an infinite state model. The CoreASM language is extended with a signature and a property plugin. The signature plugin allows for functions to be typed. This is not necessary in CASM, as already all functions are strongly typed. The property plugin in CoreASM adds the syntax for model checks. These properties must be formulated in LTL. The translation from CoreASM into PROMELA is similar to the translation from CASM to TPTP. In both cases, the AST is traversed and update rules are generated for symbolic variables. However, in CoreASM functions are represented as integer constants in PROMELA, with current and next state locations.

Beckers et al. [13] use the CoreASM simulation to generate a state space of the ASM. They use the model checker [mc]square to check CTL properties and generate counter examples. This prevents the loss of expressiveness due to translation, but suffers from the inefficiencies of needing to create the state space. Peng et al. [58] improve this approach with their ASM-SPV tool, as it reduces the memory consumed.

Gargatini et al. [39] propose AsmetaSMV [3], the model checking tool for the ASMETA Toolset. It translates AsmetaL specifictions into the input for the NumSMV model checker [25]. Its approach to the translation is similar to the approach of CoreASM. AsmetaL functions are translated into NumSMV variables. Null-ary functions are directly translated into variables and n-ary functions are decomposed to multiple variables, each representing one location of the function. This allows only functions with bound parameters to be transformed, as unbound parameters would produce infinite variables. Properties are specified either in LTL or CTL form.

Rafe et al. [61] present ASM2Bogor, a tool to convert ASMs written in the AsmetaL language to BIR, the input language of the Bogor model checker [31]. It improves on AsmetaSMV by supporting more ASMs, as BIR supports n-ary functions. Therefore, functions with unbound parameters can correctly be transformed. Similar to AsmetaSMV, model checks are specified directly in the ASM specification in LTL form.

Arcaini et al. [5] built a framework for transforming ASMs in the ASMETA Toolset

into a model for the SMT solver Yices. Their approach is similar to this thesis in that they use uninterpreted functions in the SMT model to represent ASM functions. Rules are transformed to guarded update statements. However, concolic execution is not supported and all functions are treated as symbolic. Arcaini et al. [4] present a method for comparing two ASMs and proving whether they are identical using the SMT solver Yices.

6.4. CASM Implementation

This thesis is a direct successor to the symbolic execution implementation for CASM by Lezuo [48]. His implementation transforms a CASM specification directly into a TPTP text file which then must be used as input for an external SMT or SAT solver. It supports concolic execution. If an ASM function is not marked as `symbolic`, it is treated as a numeric function and therefore is evaluated numerically. That is, if it is part of any instruction with symbolic functions, it will appear as literal in the TPTP trace. If it is only used in the condition of conditional rules without any symbolic functions, it will not appear in the trace, and neither will the conditional rule, as only one branch is transformed.

In Lezuos implementation, ASM functions are represented as uninterpreted predicates. The signature of the predicate includes the time and the current value. The function $f(D_1, \ldots, D_n) \to D$ is therefore translated into the predicate $f(\$int, D_1, \ldots, D_n, D) \to \$o$, where $\$int$ and $\$o$ are the TPTP types for integer and the pseudo boolean type respectively. These TPTP predicates represent the state of the ASM at all times. The time argument is an integer and represents the number of ASM steps since the start of the ASM. Each transformation starts at time one (`1`). The time of the last ASM step is defined as zero (`0`). This allows for easy addition of start and end constraints, as the time argument is known in advance.

Arguments to these predicates are symbols which are represented as unbound TPTP constants. These symbols are lazily allocated as required and regardless of time as they are only once used as arguments for instruction predicates.

ASM instructions are also represented as predicates in TPTP. The operator $D_1 \circ D_2 \to D$ is transformed into the predicate $op(D_1, D_2, D) \to \$o$, where op is the name of the instruction. The ASM expression `2 * z`, where `z` is a null-ary function is translated into the trace in Listing 6.1 and the resulting symbol of the expression is `sym2`.

```
1 fof(1,hypothesis, z(1, sym1)).
2 fof(2,hypothesis, mul(2, sym1, sym2)).
```

Listing 6.1: Multiplication Trace by Lezuo [48]

In the implementation by Lezuo et. al, conditional rules cause the trace to fork. One

trace contains the *then* branch, the other the *else* branch. So for each conditional rule in the CASM specification, the number of trace files grows linearly.

7. Conclusion

This thesis presents a symbolic execution framework to transform a CASM specification into a TPTP trace and then translates this into a Z3 model. It improves on the implementation by Lezuo [48] by being implemented in the state-of-the-art CASM compiler and only producing one TPTP trace in a specification with conditional rules. However, the framework is limited to ASMs without symbolic loop conditions.

The transformation supports concolic execution, which can result in smaller TPTP traces compared to a full symbolic transformation. A full symbolic transformation, however, is possible. A user of the framework can therefore reduce or improve completeness in favor of proving speed from within the CASM specification.

The resulting TPTP traces are solver independent and can be exported without invoking any internal SMT solver. Alternatively to exporting the TPTP model, this thesis presents a model-to-model transformation from TPTP to Z3, which can be directly integrated into the framework.

One significant contribution of this thesis is a symbolic consistency pass. In the case of a concolic execution, the symbolic consistency pass checks whether any numeric functions are updated with symbolic values. If that is the case, the function is also made symbolic to ensure consistency (see Section 3.3).

7.1. Future Work

In the follwing section, various research topics are proposed which could be researched to improve the symbolic execution framework in the thesis at hand.

Predicate Semantics The automatic generation and inclusion of formulae for the semantics of the function and instruction predicates would allow for the traces to be solved by Z3.

Model Checking in CASM A language construct could be added to the CASM language, which allows model assertions to be added. These assertions could then be translated into model checks in the symbolic trace.

Transformation of CASM Intermediate Representation By translating the CASM-IR, optimized ASM models could be symbolically compared with the input ASM from the AST representation. This would be another step towards translation validation.

CASM Syntax Support CASM supports local functions which are only available during a rule and its called rules. For this purpose, a sub-ASM is created and the states are updated accordingly. Loops are another CASM construct that are currently unsupported and would improve the framework. This is done by creating a sub-ASM and updating the states accordingly. Another possibility concerning CASM Syntax Support would be the full implementation of the choose rule. Supported function types are currently only boolean and numeric functions. Building Support for string and composite types is another possible improvement.

Custom Boolean Type Using a custom defined Boolean type might be beneficial, as it would conform to the TPTP standard. However, performance implications while transforming and solving have to be taken into account.

TPTP Typed Higher Order Form The full support for the TPTP typed higher order formulae would improve the TPTP framework of the thesis. If it fully supports the transformation into Z3, it could alternatively be used as a TPTP front end for Z3.

Symbol Reuse A system which reuses symbolic constants in the TPTP trace would greatly reduce the length of the traces and potentially accelerate the solving process of Z3.

Performance Evaluation Comparing transformation times and solving times of the SMT solver to similar systems could point towards possible performance improvements.

Model Feedback Another direction to improve the symbolic execution pass could be to include source location information in the TPTP formulae and parsing the calculated Z3 model to provide feedback in the CASM language.

Bibliography

[1] E. Abraham, "Building bridges between symbolic computation and satisfiability checking," in *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–6. [Online]. Available: https://doi-org.uaccess.univie.ac.at/10.1145/2755996.2756636

[2] E. Abraham and G. Kremer, "Satisfiability checking: Theory and applications," in *International Conference on Software Engineering and Formal Methods*. Springer, 2016, pp. 9–23.

[3] P. Arcaini, A. Gargantini, and E. Riccobene, "AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications," in *International Conference on Abstract State Machines, Alloy, B and Z*. Springer, 2010, pp. 61–74.

[4] P. Arcaini, A. Gargantini, and E. Riccobene, "SMT-Based Automatic Proof of ASM Model Refinement," in *Software Engineering and Formal Methods*, R. De Nicola and E. Kühn, Eds. Cham: Springer International Publishing, 2016, pp. 253–269.

[5] P. Arcaini, A. Gargantini, and E. Riccobene, "SMT for state-based formal methods: the ASM case study." in *AFM@ NFM*, 2017, pp. 1–18.

[6] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 50, 2018.

[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177.

[8] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," www.SMT-LIB.org, 2016.

[9] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening, Eds., 2010.

[10] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[11] P. Baumgartner, "SMTtoTPTP – A Converter for Theorem Proving Formats," in *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds. Cham: Springer International Publishing, 2015, pp. 285–294.

[12] P. Baumgartner, J. Bax, and U. Waldmann, "Beagle – a hierarchic superposition theorem prover," in *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds. Cham: Springer International Publishing, 2015, pp. 367–377.

[13] J. Beckers, D. Klünder, S. Kowalewski, and B. Schlich, "Direct support for model checking abstract state machines by utilizing simulation," in *Abstract State Machines, B and Z*, E. Börger, M. Butler, J. P. Bowen, and P. Boca, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 112–124.

[14] A. Bianchi, S. Pizzutilo, and G. Vessio, "Towards an asm-based characterization of the deadlock-freedom property." in *ICSOFT-PT*, 2016, pp. 123–130.

[15] A. Bianchi, S. Pizzutilo, and G. Vessio, "An asm-based characterisation of starvation-free systems," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 33, no. 1, pp. 35–51, 2018.

[16] J. Blanchette and A. Paskevich, "TFF1: The TPTP Typed First-order Form with Rank-1 Polymorphism," in *Proceedings of the 24th International Conference on Automated Deduction*, ser. Lecture Notes in Artificial Intelligence, M. Bonacina, Ed., no. 7898. Springer-Verlag, 2013, pp. 414–420.

[17] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The barcelogic smt solver," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 294–298.

[18] E. Borger and R. F. Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Berlin, Heidelberg: Springer-Verlag, 2003.

[19] E. Börger and J. Schmid, "Composition and submachine concepts for sequential asms," in *International Workshop on Computer Science Logic*. Springer, 2000, pp. 41–60.

[20] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The mathsat 4 smt solver," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 299–303.

[21] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The opensmt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 150–153.

[22] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[23] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later." *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[24] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[25] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[26] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*. Springer, 1981, pp. 52–71.

[27] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[28] G. Del Castillo and K. Winter, "Model checking support for the asm high-level language," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Graf and M. Schwartzbach, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 331–346.

[29] S. Duraibi, A. M. Alashjaee, and J. Song, "A survey of symbolic execution tools," *International Journal of Computer Science and Security (IJCSS)*, vol. 13, no. 6, p. 244, 2019.

[30] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 737–744.

[31] M. B. Dwyer, J. Hatcliff *et al.*, "Bogor: A flexible framework for creating software model checkers," in *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*. IEEE, 2006, pp. 3–22.

[32] R. Farahbod, V. Gervasi, and U. Glässer, "Coreasm: An extensible asm execution engine," *Fundamenta Informaticae*, vol. 77, no. 1-2, pp. 71–103, 2007.

[33] R. Farahbod, U. Glässer, and G. Ma, "Model checking coreasm specifications," in *Proceedings of the 14th International ASM Workshop (ASM'07)*. Citeseer, 2007.

[34] M. Y. R. Gadelha, L. C. Cordeiro, and D. A. Nicole, "Encoding floating-point numbers using the smt theory in esbmc: An empirical evaluation over the sv-comp benchmarks," in *Formal Methods: Foundations and Applications*, S. Cavalheiro and J. Fiadeiro, Eds. Cham: Springer International Publishing, 2017, pp. 91–106.

[35] J. H. Gallier, *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications, 2015.

[36] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[37] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *SOFTWARE - PRACTICE AND EXPERIENCE*, vol. 30, no. 11, pp. 1203–1233, 2000.

[38] A. Gargantini, E. Riccobene, and S. Rinzivillo, "Using spin to generate tests from asm specifications," in *Abstract State Machines 2003*, E. Börger, A. Gargantini, and E. Riccobene, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 263–277.

[39] A. M. Gargantini, E. Riccobene, and P. Scandurra, "A metamodel-based language and a simulation engine for abstract state machines," *Journal of Universal Computer Science*, vol. 14, no. 12, pp. 1949–1983, 2008.

[40] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[41] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[42] Y. Gurevich and E. Börger, "Evolving algebras 1993: Lipari guide," *Evolving Algebras*, p. 40, 1995.

[43] W. Hodges, "Classical logic i: first order logic," *The Blackwell Guide to Philosophical Logic. Blackwell*, 2001.

[44] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[45] M. Kardos, "An approach to model checking asml specifications." in *Abstract State Machines*, 2005, pp. 289–304.

[46] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[47] E. Kotelnikov, L. Kovács, G. Reger, and A. Voronkov, "The vampire and the fool," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, ser. CPP 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 37–48. [Online]. Available: https://doi-org.uaccess.univie.ac.at/10.1145/2854065.2854071

[48] R. Lezuo, "Scalable translation validation," Ph.D. dissertation, PhD thesis, Vienna University of Technology, 2014.

[49] R. Lezuo, G. Barany, and A. Krall, "CASM: Implementing an Abstract State Machine based Programming Language," in *Software Engineering Conference (Workshops), Arbeitstagung Programmiersprachen, ATPS 2013*, 2013, pp. 75–90.

[50] R. Lezuo, P. Paulweber, and A. Krall, "CASM - Optimized Compilation of Abstract State Machines," in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES 2014*. ACM, 2014, pp. 13–22.

[51] E. Mendelson, *Introduction to mathematical logic*. CRC press, 2009.

[52] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 179–180.

[53] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International journal on software tools for technology transfer*, vol. 11, no. 4, p. 339, 2009.

[54] P. Paulweber, *An optimizing compiler for the abstract state machine language CASM*, 2014. [Online]. Available: https://resolver.obvsg.at/urn:nbn:at:at-ubtuw:1-64768

[55] P. Paulweber, E. Pescosta, and U. Zdun, "CASM-IR: Uniform ASM-Based Intermediate Representation for Model Specification, Execution, and Transformation," in *6th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, ABZ 2018*, ser. Lecture Notes in Computer Science 10817. Springer, 2018, pp. 39–54.

[56] P. Paulweber, E. Pescosta, and U. Zdun, "Structuring the state and behavior of asms: Introducing a trait-based construct for abstract state machine languages," in *Rigorous*

State-Based Methods, A. Raschke, D. Méry, and F. Houdek, Eds. Cham: Springer International Publishing, 2020, pp. 237–243.

[57] P. Paulweber and U. Zdun, "A Model-Based Transformation Approach to Reuse and Retarget CASM Specifications," in *5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, ABZ 2016*, ser. Lecture Notes in Computer Science 9675. Springer, 2016, pp. 250–255.

[58] J. Peng, F. Liu, Z. Zhao, D. Huang, and R. Xue, "Asm-spv: A model checker for security protocols," in *2010 Sixth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*. IEEE, 2010, pp. 458–461.

[59] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE, 1977, pp. 46–57.

[60] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998, pp. 151–166.

[61] V. Rafe and S. Doostali, "Asm2bogor: An approach for verification of models specified through asmeta language," *Journal of Visual Languages & Computing*, vol. 23, no. 5, pp. 287–298, 2012.

[62] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[63] G. Sutcliffe, "The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0," *Journal of Automated Reasoning*, vol. 59, no. 4, pp. 483–502, 2017.

[64] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner, "The TPTP Typed First-order Form with Arithmetic," in *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, ser. Lecture Notes in Artificial Intelligence, N. Bjørner and A. Voronkov, Eds., no. 7180. Springer-Verlag, 2012, pp. 406–419.

[65] G. Sutcliffe, "The TPTP problem library and associated infrastructure," *Journal of Automated Reasoning*, vol. 43, no. 4, p. 337, 2009.

[66] G. Sutcliffe and C. Benzmueller, "Automated reasoning in higher-order logic using the tptp thf infrastructure," *Journal of Formalized Reasoning*, vol. 3, no. 1, pp. 1–27, Apr. 2010. [Online]. Available: https://jfr.unibo.it/article/view/1710

[67] G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder, "Using the tptp language for writing derivations and finite interpretations," in *International Joint Conference on Automated Reasoning*. Springer, 2006, pp. 67–81.

[68] G. Sutcliffe, J. Zimmer, and S. Schulz, "TSTP data-exchange formats for automated theorem proving tools," in *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, W. Zhang and V. Sorge, Eds., vol. 112. IOS Press, 2004, p. 201.

[69] C. K. F. Tang and E. Ternovska, "Model checking abstract state machines with answer set programming," in *Logic for Programming, Artificial Intelligence, and Reasoning*, G. Sutcliffe and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 443–458.

[70] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*. Springer, 2008, pp. 134–153.

[71] X. Wang, L. Zhang, and P. Tanofsky, "Experience report: How is dynamic symbolic execution different from manual testing? a study on klee," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 199–210.

[72] T. Weber, S. Conchon, D. Déharbe, M. Heizmann, A. Niemetz, and G. Reger, "The smt competition 2015–2018," in *Journal on Satisfiability, Boolean Modeling and Computation*. IOS Press, 2019, pp. 221–259.

[73] K. Winter, "Towards a methodology for model checking asm: Lessons learned from the flash case study," in *International Workshop on Abstract State Machines*. Springer, 2000, pp. 341–360.

[74] K. Winter, "Model checking abstract state machines," Doctoral Thesis, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Berlin, 2001. [Online]. Available: http://dx.doi.org/10.14279/depositonce-423

[75] X. Yang, *Random testing of open source C compilers*. The University of Utah, 2015.

[76] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

Acronyms

API Application Programming Interface.

ASM Abstract State Machine.

ASP Answer Set Programming.

AST Abstract Syntax Tree.

CASM Corinthian Abstract State Machine.

CNF Clause Normal Form.

CTL Computation Tree Logic.

CTL* Superset of CTL and LTL.

DSE Dynamic Symbolic Execution.

EL Emitting Language.

FOF First Order Form.

FSM Finite State Machine.

GCD Greatest Common Divisor.

IR Intermediate Representation.

LCM Least Common Multiple.

LTL Linear Temporal Logic.

RCG Rule Call Graph.

S2E Selective Symbolic Execution.

SAT Boolean Satisfiability.

SMT Satisfiability Modulus Theories.

TFF Typed First Order Form.

THF Typed Higher Order Form.

TPTP Thousands of Problems for Theorem Provers.

A. Appendix

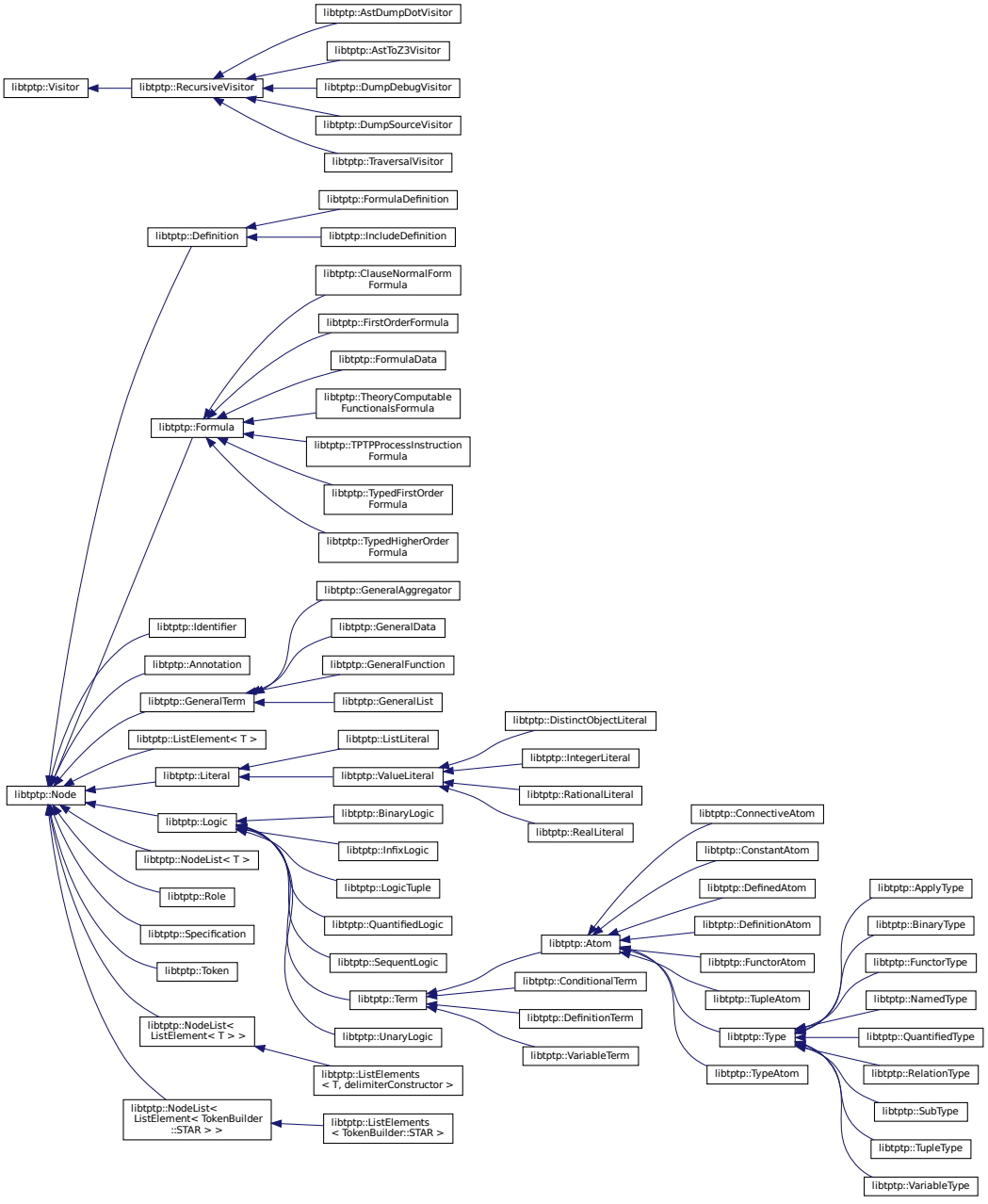

Figure A.1.: Complete TPTP Class Diagram

```
CASM
init test

[symbolic] function a : -> Integer
[symbolic] function b : -> Integer
[symbolic] function c : -> Integer

rule test =
{
    if a = 0 then {
        if c != 0 then
            if a = 0 then
                b := 1
    }
    else
        b := 2

    program( self ) := undef
}
```

Listing A.1: CASM: Conditional Rules Extended

```
% specification:
tff(3,type,'%0':$int).
tff(5,type,'%1':$o).
tff(7,type,'%2':$int).
tff(9,type,'%3':$o).
tff(11,type,'%4':$int).
tff(13,type,'%5':$o).
tff(17,type,'%6':$int).
tff(20,type,'%7':$int).
tff(23,type,'%8':$int).
tff(0,hypothesis,'@a':($int*$int)>$o).
tff(1,hypothesis,'@b':($int*$int)>$o).
tff(2,hypothesis,'@c':($int*$int)>$o).
tff(4,hypothesis,'@a'(1,'%0')).
tff(6,hypothesis,'%1'<=>('%0'=0)).
tff(8,hypothesis,'%1'=>('@c'(1,'%2'))).
tff(10,hypothesis,'%1'=>('%3'<=>('%2'!=0))).
tff(12,hypothesis,'%1'=>('%3'=>('@a'(1,'%4')))).
tff(14,hypothesis,'%1'=>('%3'=>('%5'<=>('%4'=0)))).
tff(15,hypothesis,'%1'=>('%3'=>('%5'=>('@b'(2,1))))).
tff(16,hypothesis,~'%1'=>('@b'(2,2))).
tff(18,hypothesis,'@a'(1,'%6')).
tff(19,hypothesis,'@a'(0,'%6')).
tff(21,hypothesis,'@b'(2,'%7')).
tff(22,hypothesis,'@b'(0,'%7')).
tff(24,hypothesis,'@c'(1,'%8')).
tff(25,hypothesis,'@c'(0,'%8')).
```

Listing A.2: TPTP: Conditional Rules Extended