About the Concolic Execution and Symbolic ASM Function Promotion in CASM

Philipp Paulweber[1]*, Jakob Moosbrugger**, and Uwe Zdun[2]

[1] Vienna University of Technology, Institute of Information Systems Engineering
Research Unit Compilers and Languages (CompLang)
Argentinierstraße 8, 1040 Vienna, Austria
ppaulweber@complang.tuwien.ac.at
[2] University of Vienna, Faculty of Computer Science
Research Group Software Architecture (SWA)
Währingerstraße 29, 1090 Vienna, Austria
uwe.zdun@univie.ac.at

Abstract. Abstract State Machines (ASMs) are a well-known state based formal method to describe systems at a very high level and can be executed either through a concrete or symbolic interpretation. By symbolically executing an ASM specification, certain properties can be checked by transforming the described ASM into a suitable input for model checkers or Automated Theorem Provers (ATPs). Due to the rather fast increasing state space, model checking and ATP solutions can lead to inefficient implementations of symbolic execution. More efficient state space and execution performance can be achieved by using a concolic execution approach. In this paper, we describe an improved concolic execution implementation for the Corinthian Abstract State Machine (CASM) language. We outline the transformation of a symbolically executed ASM specification to a single Thousands of Problems for Theorem Provers (TPTP) format. Furthermore, we introduce a compiler analysis to promote concrete ASM functions into symbolic ones in order to obtain symbolic consistency.

Keywords: Abstract State Machine, Concolic Execution, CASM, TPTP, Z3

1 Introduction

Due to the mathematical foundation of the Abstract State Machine (ASM) theory [1] [2], ASM specifications can be evaluated through either concrete or symbolic interpretation. All available ASM implementations offer a concrete execution, and some ASM implementations provide a symbolic execution based on model checking (e.g. Farahbod et al. [3] for CoreASM). Besides the approaches targeting model checking applications, some ASM implementations transform

* The work in this paper was carried out at the former affiliation[2]
** No affiliation

```
CASM

init test

[symbolic]
function x : -> Integer

[symbolic]
function y : -> Integer

rule test =
{
    if x = 0 then
        skip
    else
        y := 12 / x
    program( self ) := undef
}




// ...
```

Listing 1.1: `Example.casm`

```
tff(symbolNext, type, sym2: $int).
fof(id0,hypothesis,x(1,sym2)).
fof('Example.casm:13',hypothesis,sym2=0).
fof(id1,hypothesis,x(2,sym2)).
fof(final0,hypothesis,x(0,sym2)).
```

Listing 1.2: If-Then-Branch TPTP Trace of `Example.casm` by Lezuo [6]

```
tff(symbolNext, type, sym2: $int).
fof(id0,hypothesis,x(1,sym2)).
fof('Example.casm:13',hypothesis,sym2!=0).
tff(symbolNext, type, sym4: $int).
tff(symbolNext, type, sym5: $int).
fof(id1,hypothesis,y(1,sym5)).
fof(id2,hypothesis,x(2,sym2)).
fof(id3,hypothesis,y(2,sym4)).
fof(final0,hypothesis,x(0,sym2)).
fof(final1,hypothesis,y(0,sym4)).
```

Listing 1.3: Else-Branch TPTP Trace of `Example.casm` by Lezuo [6]

the specifications into Automated Theorem Provers (ATP) problems to check with off-the-shelve solver tools desired properties (e.g. Arcaini et al. [4] for AsmetaL with SMT solver Yices). A major disadvantage of such techniques is that for rather small ASM specifications, huge ATP input problems are generated which result into large states and long evaluation times of the underlying solver.

To overcome this problem, a *concolic execution* [5] can be used to reduce the number of symbolic path conditions by performing a mixed concrete and symbolic interpretation. Branches inside an evaluation are driven by concrete results and only symbolic states of interest are tracked in the output trace which directly optimizes the results. Therefore, concolic execution [5] trades completeness for computation speed. So far, only Lezuo [6] described a concolic execution approach for ASM specifications. Based on a prototype version of the Corinthian Abstract State Machine (CASM) language[3] [7], the described concolic execution performed a model-to-text transformation by emitting directly multiple Thousands of Problems for Theorem Provers (TPTP) [8] traces of the symbolically executed specification. A downside of Lezuos' [6] approach is that for each conditional rule (path condition) the generated TPTP trace gets forked into an *if-then* and *else* part resulting into two TPTP specifications which are emitted during the symbolic execution of an ASM specification.

Listing 1.1 depicts an example CASM specification consisting of two functions – `x` and `y` – and a named rule `test` with a block rule, conditional rule, skip rule, and two update rules. This specification represents the running example which was used by Lezuo [6] to describe a *division-by-zero-free* ASM specification expressed in the latest CASM language syntax. Both functions – `x` and `y` – are set explicitly to `symbolic` in order to determine a TPTP trace showing that the function `y` gets only updated with a non-zero Integer value of function `x`. Two TPTP traces are generated by using Lezuos' [6] implemented (closed source) symbolic execution. Listing 1.2 depicts the *if-then* part and the Listing

[3] For the CASM syntax description, see: `https://casm-lang.org/syntax`

1.3 depicts the *else* part. Based on this traces, a language user can use an external ATP solver Z3 [9] or vanHelsing [10] and prove the *division-by-zero-free* property for the functions `y` and `x` by analyzing each TPTP trace.

We present in this paper an improved version of the concolic execution for the (open-source) CASM language and implementation. Based on the concolic execution definition by Lezuo [6], we provide two major improvements in the current presented implementation state: (1) the concolic execution generates a single TPTP trace and does not generate forked TPTP traces for each path condition (see Section 2); and (2) a language user only has to set ASM functions of interest to `symbolic` and each ASM function is automatically promoted to `symbolic` if there exists a path which updates that ASM function (see Section 3). Furthermore, we do not directly generate TPTP traces through a model-to-text transformation. We have implemented an abstraction of the TPTP model and provided an in-memory model-to-model transformation. This design decision allows us to directly (re)use in the CASM compiler the transformed TPTP instance either for further analysis, in-memory evaluation, or emitting to a textual representation in order to use an external solver.

2 CASM Concolic Execution and TPTP Model

CASM is a concrete ASM implementation with a strongly typed inferred specification language. The concolic execution is implemented as forward symbolic execution by reusing and extending the Abstract Syntax Tree (AST) based concrete execution[4]. Due to the CASM compiler design [11], the symbolic constant, calculation, and environment handling is directly implemented on the CASM Intermediate Representation (IR) level[5]. Our own TPTP implementation[6] supports in-memory model-to-model transformation based on the SMT/SAT solver Z3 [9] to invoke a Z3-based evaluation without external tooling.

Since each ASM function can be explicitly selected to be evaluated as symbolic state (annotation syntax), a complete selection of all available ASM functions inside a specification would enable a full symbolic execution of the provided specification. So far we support all basic ASM rules in the transformation except for symbolic `iterate` rules consisting of symbolic path conditions. Listing 1.4 depicts the same *division-by-zero-free* running example as shown in Listing 1.1 with one small change. In this listing the function `y` is not explicitly set to `symbolic`, because the function of interest we want to analyze is the function `x`. Function `y` gets implicitly set to `symbolic` through a novel compiler analysis pass (see Section 3) in order to provide symbolic consistency for the specified update to function `y` where function `x` is used in the division operation (see Listing 1.4 at Line 16). Listing 1.5 corresponds to the result TPTP trace of the concolic execution. A first look at this TPTP trace gives the impression that it is longer than both TPTP traces combined of the previous implementation depicted in

[4] For CASM front-end, see: `https://github.com/casm-lang/libcasm-fe/pull/206`
[5] For CASM mid-end, see: `https://github.com/casm-lang/libcasm-ir/pull/29`
[6] For TPTP model, see: `https://github.com/casm-lang/libtptp/pull/5`

```
CASM

init test

[symbolic]
function x : -> Integer

// concrete, not set symbolic
function y : -> Integer

rule test =
{
    if x = 0 then
        skip
    else
        y := 12 / x
    program( self ) := undef
}
```

Listing 1.4: `Example.casm`

```
 1 tff(2,type,'%0':$int).
 2 tff(4,type,'%1':$o).
 3 tff(6,type,'%2':$int).
 4 tff(8,type,'%3':$int).
 5 tff(12,type,'%4':$int).
 6 tff(15,type,'%5':$int).
 7 tff(9,hypothesis,'#div#i':($int*$int*$int)>$o).
 8 tff(0,hypothesis,'@x':($int*$int)>$o).
 9 tff(1,hypothesis,'@y':($int*$int)>$o).
10 tff(3,hypothesis,'@x'(1,'%0')).
11 tff(5,hypothesis,'%1'<=>('%0'=0)).
12 tff(7,hypothesis,~'%1'=>('@x'(1,'%2'))).
13 tff(10,hypothesis,~'%1'=>('#div#i'(12,'%2','%3'))).
14 tff(11,hypothesis,~'%1'=>('@y'(2,'%3'))).
15 tff(13,hypothesis,'@x'(1,'%4')).
16 tff(14,hypothesis,'@x'(0,'%4')).
17 tff(16,hypothesis,'@y'(2,'%5')).
18 tff(17,hypothesis,'@y'(0,'%5')).
```

Listing 1.5: TPTP Trace of `Example.casm`

Listing 1.2 and Listing 1.3, but besides the path condition fork there is a huge difference in the form of the trace representation itself. Lezuos' [6] implementation uses mixed First Order Form (FOF) and Typed First Order Form (TFF) formulae to represent the state evolving which fully complies to the deprecated TPTP versions before 7.0 [8]. Since the latest major revision 7 of TPTP the mixing of FOF and TFF does not work anymore, because variables and constants in FOF formulae are assumed to be in the same infinite domain, which is not the case for any type in a TFF formulae [8]. The later implies that each variable or constant in a TFF formulae is not equal to any variable or constant in a FOF formula. Therefore, we generate a fully typed TPTP trace by using only TFF formulae in the trace result. A transformed TPTP trace consists of four parts: (1) type declarations for intermediate calculations (see Listing 1.5 Line 1 to 6); (2) language operand definitions (see Listing 1.5 Line 7); (3) all function definitions (see Listing 1.5 Line 8 to 9); and (4) the actual trace itself (see Listing 1.5 Line 10-18). Since in TPTP each variable can only be used once and there exists no notion of time, each ASM function gets mapped to a TPTP predicate with 2 or more arguments where the first argument represents an Integer based time. Similar to the definition by Lezuo [6], we use time at `1` to represent the initialization of ASM functions. Time at `0` equals the termination of an ASM execution. This encoding provides an elegant way to describe start and termination constraints, since the times are known before the concolic execution starts. Furthermore, since CASM supports block rules (parallel execution semantics) and sequential rules (sequential execution semantics) the handling of parallelism is an important issue. The evolving of function states (ASM steps) is encoded in the time value of each function in the first argument. Sequential rule computations which create *pseudo update-sets* [7] are not shown and tracked in the TPTP trace except for the remaining update to functions.

3 ASM Function Promotion and Symbolic Consistency

Due to the possibility that some ASM functions in a CASM specification can be marked as `symbolic`, the concolic execution can reach an interpretation of

```
casmi: info: promoting function 'y' to be symbolic, because function is
updated with symbolic value.
Example.casm:16:8..16:19
    y := 12 / x
    ^---------^
```

Listing 1.6: CLI Tool Information of ASM Symbolic Function Promotion

the ASM specification where a symbolic value or calculation could be used in an update rule to a concrete ASM function. This would abort the concolic execution and would lead to an execution error, because the symbolic consistency is violated. Therefore, we implemented a symbolic consistency analysis in the compiler pass pipeline which analyses in advance which concrete ASM functions will be updated by symbolic values. Note that updating a symbolic ASM function with a concrete value (e.g. a numeric value) is possible and does not violate symbolic consistency.

The symbolic consistency pass is an AST-based compiler analysis pass and checks if any function update produces a symbolic conflict. Each function, rule parameters, and expression AST node gets annotated by the analysis which labels the nodes either *symbolic*, *concrete*, or *unknown*.

Depending on the annotated functions through the annotation syntax, all functions are labeled either *symbolic* or *concrete* and all other nodes in the AST are labeled *unknown* at the beginning of the analysis. Since CASM supports named rule calls, each possible rule call hierarchy starting from the `init` statement has to be evaluated in order to determine symbolic consistency. The analysis derives in a step-by-step manner a Rule Call Graph (RCG) where each callable rule has to go through four states – *init*, *started*, *evaluated*, and *finished*. The resulting RCG is used to derive the final symbolic function promotion which assures symbolic consistency.

We implemented a proper reporting of ASM functions which are promoted to symbolic. Listing 1.6 depicts a console output of our CASM interpreter Command Line Interface (CLI) tool named `casmi`[7] which evaluated in concolic/symbolic execution mode the `Example.casm` specification shown in Listing 1.4 and outputs an information message that function `y` gets promoted to a symbolic ASM function.

4 Conclusion

In this paper, we describe an improved ASM based concolic execution approach which is implemented for the CASM language and its framework.

Novel about this contribution is that the transformation of an ASM specification towards a TPTP model instance is performed through an in-memory model-to-model transformation which allows either further in-memory analysis, optimization, and evaluation of the TPTP instance or a flexible model-to-text transformation into a TPTP textual representation. Furthermore, the implemented approach only generates a single TPTP trace and promotes non-symbolic

[7] For CLI tool `casmi`, see: `https://github.com/casm-lang/casmi/pull/12`

ASM functions to symbolic ones if the symbolic consistency is violated which is determined in advance through a symbolic consistency pass.

With our new concolic execution approch we aim at a complete translation validation of the CASM compiler implementation itself by checking each internal transformation step of the intermediate models [11]. Moreover, due to the introduction of state and behavioral separation in the CASM language [12], we are currently investigating the ability of automated semantic checking for imported ASM rules from loaded libraries or modules.

Acknowledgements We would like to thank Andreas Krall[1] for proof-reading the paper and Emmanuel Pescosta for several concolic execution discussions.

References

[1] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide - Specification and Validation Methods," pp. 9–36, New York, NY, USA: Oxford University Press, Inc., 1995.

[2] E. Börger and A. Raschke, *Modeling Companion for Software Practitioners.* Springer, 2018.

[3] R. Farahbod, U. Glässer, and G. Ma, "Model Checking CoreASM Specifications," in *Proceedings of the 14th International ASM Workshop (ASM'07)*, Citeseer, 2007.

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

[5] 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.

[6] R. Lezuo, *Scalable Translation Validation; Tools, Techniques and Framework.* PhD thesis, 2014. Wien, Techn. Univ., Diss.

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

[8] G. Sutcliffe, "The TPTP Problem Library and Associated Infrastructure," *Journal of Automated Reasoning*, pp. 1–20, 2017.

[9] 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*, pp. 337–340, Springer, 2008.

[10] R. Lezuo, I. Dragan, G. Barany, and A. Krall, "vanHelsing: A Fast Proof Checker for Debuggable Compiler Verification," in *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 167–174, IEEE, 2015.

[11] P. Paulweber, E. Pescosta, and U. Zdun, "CASM-IR: Uniform ASM-Based Intermediate Representation for Model Specification, Execution, and Transformation," in *ABZ 2018*, LNCS 10817, pp. 39–54, Springer, 2018.

[12] 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 *International Conference on Rigorous State-Based Methods*, Lecture Notes in Computer Science 12071, pp. 237–243, Springer, 2020.