Using Symbolic Execution to Transform Turbo Abstract State Machines into Basic Abstract State Machines*

Giuseppe Del Castillo[0009−0005−7020−6607]
giuseppedelcastillo@acm.org

Abstract. This paper introduces a transformation method that uses symbolic execution to eliminate sequential composition (`seq`) rules from turbo ASM rules by translating them into equivalent rules without `seq`. In some cases `iterate` rules can also be eliminated. A prototype tool that implements the transformation and some examples are presented in order to demonstrate the feasibility of the method. The material presented here is work in progress. The prototype tool is publicly available.

Keywords: Abstract State Machines · symbolic execution.

1 Introduction

Abstract State Machines (ASM) [6] are a well-established, rigorous state-based method for system design and analysis. The ASM method has been used to define the semantics of programming and modelling languages (Prolog, Java and UML state and activity diagrams, among others), specify and document hardware and software systems (including processor architectures, embedded control systems, communication protocols) and mathematically prove properties of the modelled languages and systems. In some cases theorem provers have been used to formally verify proofs of properties of ASM models (see the Prolog-WAM case study [19] or the Verifix project addressing the construction of formally verified compilers [11]; an overview of this line of research can be found in [4]). A limited degree of automated verification support has also been provided in ASM tools through interfaces to model checkers such as SMV / NuSMV or Spin [2, 8, 15].

The original definition of ASM by Gurevich [12] has been extended over the years in several ways to become the ASM method in its current form. An important extension is the "turbo ASM" introduced by Börger and Schmid [5], with rules for sequential composition, iteration and parameterized submachines. It provides a very expressive structuring mechanism, but the implications of its semantics, with non-observable intermediate steps and hidden states, are not straightforward, especially in combination with `par` rules of basic ASM (see [10]).

Therefore, in some situations, it may be desirable to transform a turbo ASM into a basic sequential ASM as defined in [12], for example when one wants to translate an ASM into the input language of a model checker. In such a language, models are typically specified in terms of the relation between current and next

* Extended version of a short paper accepted for publication in the proceedings of the ABZ 2024 conference (Springer Nature).

state in a sequence of states. This fits well with the computation model of basic ASM, but not with turbo ASM. In fact, none of the model checking interfaces that we are aware of supports turbo ASM.

In this paper, we introduce a method using symbolic execution to transform a subset of turbo ASM (not including submachines) into basic ASM. Furthermore, we present a prototype tool that implements the transformation, together with examples, in order to demonstrate the feasibility of the proposed method.

Symbolic execution is a rather old technique that dates back to the 1970s [13, 21] and has started seeing renewed interest with the advent of SMT solvers, which have increased its practical applicability (see [3] for an overview). To the author's knowledge, the main works investigating symbolic execution of ASM are [14, 18, 20].[1] The work by Schellhorn et al. [20] is not about direct symbolic execution of ASM rules, but defines a relational encoding of rules into a logical formula and a logical calculus based on symbolic execution then can be used for verifying properties of ASM models using interactive theorem provers such as KIV. The work by Lezuo [14] is similar to ours in that it directly applies classic techniques of symbolic execution to ASM, but has a different focus, as it produces symbolic traces to perform translation validation on compiler-generated code, rather than a human-readable transformed ASM rule than can be executed. Finally, [18] applies symbolic execution to a timed variant of ASM for the purpose of estimating worst-case execution time.

This paper is structured as follows: after summarising basic notions of ASM (Sect. 2), we define our symbolic execution framework (Sect. 3) and show how to use it for transforming turbo ASM into basic ASM (Sect. 4). Then, we present a prototype implementation as well as some examples and experimental results obtained using this tool (Sect. 5) and, finally, draw some conclusions (Sect. 6).

2 Basic Notions of Abstract State Machines

In this section we summarise essential notions of single-agent, deterministic ASM and introduce a few notational conventions. Only the minimal subset of ASM needed to illustrate the symbolic execution techniques of this paper is presented. The limitations of this subset are: only static, controlled and monitored functions are allowed; there are no variables nor variable-binding constructs, only ground terms; there are no logic formulas, but terms of *Boolean* type are used instead; only six kinds of transition rules are used: `skip`, update, conditional, `seq`, `par`, `iterate`. Furthermore, we use a typed variant of the ASM language.[2]

[1] Previous works such as the Verifix project [11] used symbolic execution techniques in ad hoc manner, but did not address the systematic symbolic execution of ASM rules. Recently, [9] has described a novel application of symbolic execution of control state ASM, but is not concerned with how to implement the symbolic execution.

[2] Strong typing is necessary for an effective use of SMT solvers to support symbolic execution. A type system which does not use *undef* to model partial functions, such the one described in [22], while being a significant departure from the standard ASM definition, may further facilitate the use of SMT solvers.

2.1 Computational Model

Abstract State Machines formalise a state-based model of computation, where computations (*runs*) are finite or infinite sequences of states $S_0 \to S_1 \to \ldots$ starting from an initial state S_0, where each state S_{i+1} is obtained by updating the state S_i in a way that is determined by evaluating (in state S_i) a transition rule P, called the *program*. In symbols[3]: $S_{i+1} = S_i + \{\!|P|\!\}_{S_i}$.

A run terminates in a final state S_n if $\{\!|P|\!\}_{S_n}$ yields no updates, i.e. an empty update set (termination with success), or conflicting updates, i.e. an inconsistent update set (termination with failure). If there is no such final state, the run is an infinite sequence of states. The precise meaning of states, updates, update sets and transition rules is defined below.

States The *states* are algebras over a given *signature* Σ (Σ-algebras for short). A signature consists of a set of *basic types* and a set of *function names*. Each function name has a fixed arity n and type $T_1, \ldots, T_n \to T$, where the T_i and T are basic types (written $f : T_1, \ldots, T_n \to T$; if f has arity 0, $f :\to T$ is shortened to $f : T$). A Σ-algebra (state) S consists of:

(i) a nonempty set $\mathcal{T}^S$ for each basic type T in Σ (the *carrier set* of basic type T in S; any element $x \in \mathcal{T}$ is called a *value* of type T, written $x : T$), and
(ii) a function $\mathbf{f}_S : \mathcal{T}_1^S \times \ldots \times \mathcal{T}_n^S \to \mathcal{T}^S$ for each function name $f : T_1, \ldots, T_n \to T$ in Σ (the *interpretation* of function name f in S).

A function name f in Σ can be declared to be of a given *kind*, which can be[4]:

- *static*, meaning that the interpretation of f is fixed and never changes during a run,
- *controlled*, meaning that the interpretation of f can change during a run due to updates resulting from the execution of transition rules (see below), or
- *monitored*, meaning that the interpretation of f can change during a run in a way that is determined by the environment.

Any signature Σ must contain a basic type *Boolean*, static nullary function names (i.e. constants) *true* : *Boolean* and *false* : *Boolean*, the usual Boolean operators ($\neg$, $\wedge$, $\vee$, etc.) and equality ($= : T \times T \to$ *Boolean* for any T in Σ). Furthermore, a signature Σ may contain a (possibly infinite) number of *special constants*, i.e. static nullary function names of basic types denoting elements of the respective carrier set according to usual conventions (e.g., $0, 1, \ldots :$ *Integer*).

When no ambiguity arises, the state S is not explicitly mentioned, e.g. we write $\mathcal{T}$ instead of $\mathcal{T}^S$ for carrier sets and $\mathbf{f}$ instead of $\mathbf{f}_S$ for static functions,

[3] The unusual notation $\{\!|\cdot|\!\}$ is used here to refer to the semantics of terms and rules (evaluation) because the notation $[\![\cdot]\!]$ will be used later to denote *symbolic* evaluation. The semantics of a term t in state S is a value $x = \{\!|t|\!\}_S$. The semantics of a rule R in state S is an update set $U = \{\!|R|\!\}_S$. Details of the semantics are provided below.

[4] The ASM method, as defined in [6], foresees further function kinds such as *shared* or *derived*, which we do not deal with for the time being.

as these never change during a run. The union of all carrier sets $\mathcal{T}$ is called the *superuniverse* $\mathcal{U}$ and any element $x \in \mathcal{U}$ is called a *value*. As a notational convention, we always use the letter x to refer to values.

Locations If $f : T_1, \ldots, T_n \to T$ is a function name, a pair $l = (f, \overline{x})$ with $\overline{x} = (x_1, \ldots, x_n) \in \mathcal{T}_1 \times \ldots \times \mathcal{T}_n$ is called a *location* (then, the *type* of l is T and the *value* of l in a state S is the element $x \in \mathcal{T}$ given by $x = \mathbf{f}_S(\overline{x})$). As a notational convention, we always use the letter l to refer to locations.

Updates and Transitions A transition transforms a state S into its *sequel state* S' by changing the interpretation of some controlled function names on a finite number of points (i.e., updating the values of a finite number of locations).

More precisely, a transition transforms S into $S' = S + U$ by firing an *update set* U at S. An update set U is a finite set of *updates* of the form $((f, \overline{x}), y)$, where $(f, \overline{x})$ is the location to be updated and y is the new value of the location after the update. The carrier sets remain unchanged in the transition from S to S'. For each controlled function name $f : T_1, \ldots, T_n \to T$ and each $\overline{x} \in \mathcal{T}_1 \times \ldots \times \mathcal{T}_n$, the interpretation of f in $S' = S + U$ is given by

$$\mathbf{f}_{S+U}(\overline{x}) = \begin{cases} y & \text{if } ((f, \overline{x}), y) \in U \\ \mathbf{f}_S(\overline{x}) & \text{otherwise.} \end{cases}$$

The above definition is applicable if U is *consistent*, i.e. it does not contain any updates $((f, \overline{x}), y)$ and $((f, \overline{x}), y')$ with $y \neq y'$ (*conflicting updates*). If U is *inconsistent*, there is no sequel state S'.

2.2 ASM Language

Terms Terms over a signature Σ are recursively defined as follows:

- If $f : T_1, \ldots, T_n \to T$ is an n-ary function name in Σ, $n \geq 0$, and $t_1, \ldots, t_n$ are terms of type $T_1, \ldots, T_n$, respectively, then

  $f(t_1, \ldots, t_n)$

  is a term (of type T).

A term defined as above is called *function application term* (or simply *application term*). If $n = 0$, an application term is usually written f instead of $f()$. The notation $t : T$ is used to indicate that t is a term of type T. Furthermore:

- For all terms $G : \mathit{Boolean}$, $t_1 : T$ and $t_2 : T$,

  `if` G `then` t_1 `else` t_2

  is also a term (of type T), called *conditional term*. G is usually called *guard*.[5]

[5] There is no conditional term in the ASM definition of [6]. In principle, an application term with an appropriately defined ternary function *if-then-else*(G, t_1, t_2) could be used. We prefer to use explicit conditional terms, as they are treated specially in connection with the path condition of symbolic execution (see Sect. 3.4 and Table 2).

The semantics of a term $t : T$ in state S is a value $\{\!\{t\}\!\}_S \in \mathcal{T}$ obtained by:

$$\{\!\{f(t_1, \ldots, t_n)\}\!\}_S = \mathbf{f}_S(\{\!\{t_1\}\!\}_S, \ldots, \{\!\{t_n\}\!\}_S) \quad (n \geq 0)$$

$$\{\!\{\texttt{if } G \texttt{ then } t_1 \texttt{ else } t_2\}\!\}_S = \begin{cases} \{\!\{t_1\}\!\}_S & \text{if } \{\!\{G\}\!\}_S = \mathbf{true} \\ \{\!\{t_2\}\!\}_S & \text{if } \{\!\{G\}\!\}_S = \mathbf{false} \end{cases}$$

Formulas There is no distinct syntactic category for logic formulas. Terms of *Boolean* type are used as formulas (and conventionally denoted by φ, ψ, or also by G, short for *guard*, when used as conditions in conditional terms and rules).

Transition rules The syntax of rules is defined by the following grammar:

$R ::=$	$\texttt{skip}$	(empty rule)
$\mid$	$f(t_1, \ldots, t_n) := t$	(update rule)
$\mid$	$\texttt{if } G \texttt{ then } R_1 \texttt{ else } R_2$	(conditional rule)
$\mid$	$R_1 \texttt{ par } R_2$	(parallel composition rule)
$\mid$	$R_1 \texttt{ seq } R_2$	(sequential composition rule)
$\mid$	$\texttt{iterate } R$	(iteration rule)

The semantics of a rule R in a state S is an update set $U = \{\!\{R\}\!\}_S$ determined as shown in Table 1, wherein $U_1 \oplus U_2$ denotes the composition of update sets:

$$U_1 \oplus U_2 = U_2 \cup \{(l, x) \in U_1 \mid \text{there is no } x' \text{ with } (l, x') \in U_2\}$$

Table 1. Evaluation of rules

$$\frac{\{\!\{t_i\}\!\}_S = x_i \text{ for all } i \in \{1, \ldots, n\}}{\{\!\{f(t_1, \ldots, t_n) := t\}\!\}_S = \{((f, (x_1, \ldots, x_n)), \{\!\{t\}\!\}_S)\}} \qquad \frac{}{\{\!\{\texttt{skip}\}\!\}_S = \emptyset}$$

$$\frac{\{\!\{G\}\!\}_S = \mathbf{true}}{\{\!\{\texttt{if } G \texttt{ then } R_1 \texttt{ else } R_2\}\!\}_S = \{\!\{R_1\}\!\}_S} \qquad \frac{\{\!\{G\}\!\}_S = \mathbf{false}}{\{\!\{\texttt{if } G \texttt{ then } R_1 \texttt{ else } R_2\}\!\}_S = \{\!\{R_2\}\!\}_S}$$

$$\frac{\{\!\{R_1\}\!\}_S = U_1 \qquad \{\!\{R_2\}\!\}_S = U_2}{\{\!\{R_1 \texttt{ par } R_2\}\!\}_S = U_1 \cup U_2}$$

$$\frac{\{\!\{R_1\}\!\}_S = U_1 \qquad U_1 \text{ is inconsistent}}{\{\!\{R_1 \texttt{ seq } R_2\}\!\}_S = U_1}$$

$$\frac{\{\!\{R_1\}\!\}_S = U_1 \qquad U_1 \text{ is consistent} \qquad \{\!\{R_2\}\!\}_{S+U_1} = U_2}{\{\!\{R_1 \texttt{ seq } R_2\}\!\}_S = U_1 \oplus U_2}$$

$$\frac{\{\!\{R\}\!\}_S = \emptyset}{\{\!\{\texttt{iterate } R\}\!\}_S = \emptyset} \qquad \frac{\{\!\{R\}\!\}_S \neq \emptyset}{\{\!\{\texttt{iterate } R\}\!\}_S = \{\!\{R \texttt{ seq } (\texttt{iterate } R)\}\!\}_S}$$

The binary rule composition operators `par` and `seq` are associative. Therefore, "R_1 `par` R_2 `par` $R_3 \ldots$" or "R_1 `seq` R_2 `seq` $R_3 \ldots$" do not need parentheses. It is nevertheless assumed that `par` and `seq` are left-associative when parsed. The following syntactic abbrevations can be used:

$$\begin{aligned}
&\texttt{if } G \texttt{ then } R &&\equiv \texttt{ if } G \texttt{ then } R \texttt{ else skip}\\
&\{R_1, R_2, \ldots\} &&\equiv R_1 \texttt{ par } R_2 \texttt{ par } \ldots\\
&[\,R_1; R_2; \ldots] &&\equiv R_1 \texttt{ seq } R_2 \texttt{ seq } \ldots\\
&\texttt{while } (G)\ R &&\equiv \texttt{ iterate } (\texttt{if } G \texttt{ then } R)
\end{aligned}$$

3 Symbolic Execution Framework

In this section, we introduce a symbolic execution infrastructure consisting of various entities mirroring the respective ASM entities. This "symbolic ASM" has the same signature Σ as the original ASM: the types and the function names, with their types and kinds (i.e. static, controlled, or monitored), are the same.

3.1 Uninterpreted Locations and Functions

The starting point for symbolic execution is an incompletely defined ASM state S where the value of some or all locations $(f, \overline{x})$ of one or more non-static function names f is *unknown*, also written "$\mathbf{f}_S(\overline{x})$ is unknown".[6] We call such locations *uninterpreted locations*.

If all possible locations $(f, \overline{x})$ of a given function name f are uninterpreted in S, we say that f is *uninterpreted* in S.

Note that a location $(f, \overline{x})$ that is uninterpreted in S becomes interpreted in a sequel state $S' = S + U$ if there is an update of $(f, \overline{x})$ in U.

We assume that only controlled and monitored function names (representing the mutable state of a system being modelled) may be uninterpreted or have uninterpreted locations. Static function names (defining the operations on the data types forming the background of an ASM, such as *Boolean*, *Integer*, etc.) always have an interpretation in the sense of Sect. 2.1.

3.2 Symbolic States

States as defined in Sect. 2.1 are replaced by *symbolic states*, where (symbolic) values of locations of type T are *partially evaluated terms* from $PE\_TERM(\Sigma, \mathcal{T})$ (see definition in Sect. 3.3 below), rather than values $x \in \mathcal{T}$ of the original ASM.

[6] We deliberately use the term "unknown" instead of "undefined" to avoid any confusion with the *undef* value normally used to model partial functions, which is an element of the ASM superuniverse. A location $(f, \overline{x})$ with $f(\overline{x}) = \mathit{undef}$ is not uninterpreted: its value is known and equals *undef*. The question may arise how the two can be distinguished. For the purpose of transforming turbo ASM into basic ASM, which is the focus of this paper, this is not necessary, as all non-static functions must be uninterpreted (see section 4). For other uses of symbolic execution, a tool may need to provide some syntax to distinguish between *undef* und "unknown value".

More precisely, a *symbolic state* S over a signature Σ consists of

(i) a nonempty set $\mathcal{T}$ for each basic type T in Σ, and
(ii) a function $\mathbf{f}_\mathsf{S} : \mathcal{T}_1 \times \ldots \times \mathcal{T}_n \to \mathit{PE\_TERM}(\Sigma, \mathcal{T})$ for each function name $f : T_1, \ldots, T_n \to T$ in Σ (the *symbolic interpretation* of f in S).

A symbolic state $\mathsf{S} = \mathit{symstate}(S)$, mapping each (interpreted or uninterpreted) location of S to a symbolic value, can initially be obtained from an incompletely defined state S with uninterpreted locations (see Sect. 3.1) as follows:

$$\mathbf{f}_\mathsf{S}(x_1, \ldots, x_n) = \begin{cases} \langle \text{val } y \rangle & \text{if } \mathbf{f}_S(x_1, \ldots, x_n) = y \\ \langle \text{initial } (f, (x_1, \ldots, x_n)) \rangle & \text{if } \mathbf{f}_S(x_1, \ldots, x_n) \text{ is unknown.} \end{cases}$$

3.3 Partially Evaluated Terms

Partially evaluated terms ("pe-terms") are trees with a structure similar to terms, but the base case of their inductive definition includes, in addition to nullary function names, the symbolic values $\langle \text{val } x \rangle$ and $\langle \text{initial } (f, \overline{x}) \rangle$ introduced above:

$$t \ ::= \ f(t_1, \ldots, t_n) \ \mid \ \langle \text{val } x \rangle \ \mid \ \langle \text{initial } (f, \overline{x}) \rangle$$

where x is a value from a subset $X \subseteq \mathcal{U}$ of the superuniverse of the original ASM and $(f, \overline{x})$ is a location (as defined in Sect. 2.1) where $f : T_1, \ldots, T_n \to T$ is a non-static function name in Σ and $\overline{x} \in \mathcal{T}_1 \times \ldots \times \mathcal{T}_n$. Such a set of partially evaluated terms is called $\mathit{PE\_TERM}(\Sigma, X)$.

The idea behind the above definition is that, in the presence of uninterpreted functions or locations, it is not always possible to fully evaluate a term, but this may be possible for some of its subterms. If a (sub-)term t' of type T can be fully evaluated to a value $x = \{\!\{t'\}\!\} \in \mathcal{T}$ ($\mathcal{T} \subseteq \mathcal{U}$), a corresponding pe-term can be replaced by $\langle \text{val } x \rangle$.

Furthermore, an application term $f(t_1, \ldots, t_n)$ for which all subterms t_i can be fully evaluated in a state S to respective values x_i, but where $\mathbf{f}_S(x_1, \ldots, x_n)$ is unknown, cannot be fully evaluated. However, such a term unambiguously identifies a location $l = (f, \overline{x}) = (f, (x_1, \ldots, x_n))$. If f is a controlled function name and the value of l is unknown in state S, then it must be the case that the value of l was unknown in the initial state and has not been updated by any transition occurring between the initial state and state S. Therefore, the location $(f, \overline{x})$ still has the same (unknown) value that it had in the initial state. The pe-term $\langle \text{initial } (f, \overline{x}) \rangle$ is used to denote this unknown initial value.[7] If f is a monitored function name, instead, its interpretation can change during an ASM run and is determined by the environment, so that it may seem inappropriate to use $\langle \text{initial } (f, \overline{x}) \rangle$ in this case. However, from the point of view of a turbo ASM whose subcomputations have to be transformed into a single basic ASM step (as described in Sect. 4), the initial state is any state S_i of a sequential ASM run and the semantics of turbo ASM are such that the values of monitored functions are guaranteed not to change until the next state S_{i+1} (see [6], Sect. 4.1).

[7] For a given location $l = (f, \overline{x})$, a pe-term $\langle \text{initial } l \rangle$ corresponds to what is usually called "symbol" or "symbolic constant" in the literature on symbolic execution, i.e. a placeholder standing for the unknown initial value of a (program) variable.

3.4 Path Condition

The "path condition" is a standard notion in symbolic execution, characterised in [13] as the "accumulation of conditions which determines a unique control flow path through the program". In the present context, the path condition for a given ASM subterm (or subrule) is the condition that holds in the branch of nested conditional terms (or rules) enclosing that subterm (or subrule). As this formula is a conjuction of the respective guards (for `then` branches) or negated guards (for `else` branches), we prefer to represent it as a set of formulas.

Therefore, the result of a symbolic evaluation does not only depend on a symbolic state S, but also on a set of formulas C (the *path condition*).

3.5 Symbolic Evaluation of Partially Evaluated Terms

The *symbolic evaluation* $[\![t]\!]_{\mathsf{S},\mathsf{C}}$ of a pe-term t in a symbolic state S under path condition C is defined in Table 2. The operator $\mathsf{simplify\_formula}[\,\mathsf{C},\varphi\,]$ used in the evaluation of Boolean terms (i.e. formulas) reduces, if possible, a formula φ to $\langle \text{val } \mathbf{true}\rangle$ or $\langle \text{val } \mathbf{false}\rangle$ in view of the path condition C:

$$\mathsf{simplify\_formula}[\,\mathsf{C},\varphi\,] = \begin{cases} \langle \text{val } \mathbf{true}\rangle & \text{if it can be proved that } \bigwedge_{\psi\in\mathsf{C}} \psi \Rightarrow \varphi \\ \langle \text{val } \mathbf{false}\rangle & \text{if it can be proved that } \bigwedge_{\psi\in\mathsf{C}} \psi \Rightarrow \neg\varphi \\ \varphi & \text{otherwise} \end{cases}$$

How $\mathsf{simplify\_formula}$ tries to prove the formulas is a matter of implementation.[8]

Note that, for some t and S, $[\![t]\!]_{\mathsf{S},\mathsf{C}}$ is undefined. In particular, the symbolic evaluation of a pe-term $t = f(t_1,\ldots,t_n)$ fails if f is not a static function name and not all subterms t_i can be fully evaluated to a $\langle \text{val } x_i\rangle$ in S. The reason to disallow this is to make sure that locations are identified unambiguously, i.e., to avoid the *aliasing* phenomenon.[9]

Note also that, by construction, when the symbolic evaluation of a pe-term t succeeds, its result $t' = [\![t]\!]_{\mathsf{S},\mathsf{C}}$, seen as a tree, has:

(i) application terms $f(\overline{t})$, where f is a static function name, as inner nodes;
(ii) symbolic values of the form $\langle \text{val } x\rangle$ or $\langle \text{initial } (f,\overline{x})\rangle$ as leaves.

This implies that the value of the pe-term t' is not state-dependent.

[8] In our prototype implementation, described in Sect. 5.1 below, we use a combination of: directly checking if $\varphi \in \mathsf{C}$, $\neg\varphi \in \mathsf{C}$, applying Boolean algebra rewrite rules and, most importantly, invoking the SMT solver if the previous measures do not succeed.

[9] For example, consider an ASM with controlled function names f (unary) and i, j (nullary), uninterpreted in a state S. If the symbolic evaluation of $f(i)$ and $f(j)$ were not undefined, it would produce $f(\langle \text{initial } i\rangle)$ and $f(\langle \text{initial } j\rangle)$, i.e. two pe-terms that do not unambiguously identify a location (they may or may not refer to the same location depending on whether $i = j$ holds in S, which cannot be determined). In order to avoid this issue, the symbolic evaluation of such terms is not defined, i.e. fails. This is the main limitation of the symbolic execution approach presented in this paper. There are various ways to address this issue, but working out a good solution in detail requires further work.

Finally, it is observed that, in each state S without uninterpreted locations, i.e. in each ASM state as defined in Sect. 2.1, the symbolic evaluation $[\![\cdot]\!]$ and the non-symbolic evaluation $\{\!\{\cdot\}\!\}$ coincide in the sense that, for each term t,

$$[\![\,\mathsf{to\text{-}pe\text{-}term}[t]\,]\!]_{symstate(S),\emptyset} = \langle \text{val } \{\!\{t\}\!\}_S \rangle$$

(where to-pe-term is the trivial conversion of a term to a pe-term).

Table 2. Symbolic evaluation of pe-terms

$$\frac{}{[\![\langle \text{val } x\rangle]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{val } x\rangle} \qquad \frac{}{[\![\langle \text{initial } (f,\overline{x})\rangle]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{initial } (f,\overline{x})\rangle}$$

$$\frac{[\![t_i]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{val } x_i\rangle \text{ for all } i \in \{1,\ldots,n\}}{[\![f(t_1,\ldots,t_n)]\!]_{\mathsf{S},\mathsf{C}} = \mathbf{f}_{\mathsf{S}}(x_1,\ldots,x_n)}$$

$$\frac{f : T_1,\ldots,T_n \to T \quad T \neq \mathit{Boolean} \quad f \text{ is static} \qquad [\![t_i]\!]_{\mathsf{S},\mathsf{C}} = t'_i \text{ for } i = 1,\ldots,n \quad t'_i \neq \langle \text{val } x_i\rangle \text{ for some } i \in \{1,\ldots,n\}}{[\![f(t_1,\ldots,t_n)]\!]_{\mathsf{S},\mathsf{C}} = f(t'_1,\ldots,t'_n)}$$

$$\frac{f : T_1,\ldots,T_n \to \mathit{Boolean} \quad f \text{ is static} \qquad [\![t_i]\!]_{\mathsf{S},\mathsf{C}} = t'_i \text{ for } i = 1,\ldots,n \quad t'_i \neq \langle \text{val } x_i\rangle \text{ for some } i \in \{1,\ldots,n\}}{[\![f(t_1,\ldots,t_n)]\!]_{\mathsf{S},\mathsf{C}} = \mathsf{simplify\_formula}[\,\mathsf{C}, f(t'_1,\ldots,t'_n)\,]}$$

$$\frac{[\![t_i]\!]_{\mathsf{S},\mathsf{C}} \neq \langle \text{val } x_i\rangle \text{ for some } i \in \{1,\ldots,n\} \qquad f \text{ is not static}}{[\![f(t_1,\ldots,t_n)]\!]_{\mathsf{S},\mathsf{C}} \text{ is not defined}}$$

$$\frac{[\![G]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{val } \mathbf{true}\rangle \text{ or } [\![t_1]\!]_{\mathsf{S},\mathsf{C}} = [\![t_2]\!]_{\mathsf{S},\mathsf{C}}}{[\![\mathtt{if}\ G\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2]\!]_{\mathsf{S},\mathsf{C}} = [\![t_1]\!]_{\mathsf{S},\mathsf{C}}} \qquad \frac{[\![G]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{val } \mathbf{false}\rangle}{[\![\mathtt{if}\ G\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2]\!]_{\mathsf{S},\mathsf{C}} = [\![t_2]\!]_{\mathsf{S},\mathsf{C}}}$$

$$\frac{[\![G]\!]_{\mathsf{S},\mathsf{C}} = G' \quad G' \notin \{\langle \text{val } \mathbf{true}\rangle, \langle \text{val } \mathbf{false}\rangle\} \quad [\![t_1]\!]_{\mathsf{S},\mathsf{C}} \neq [\![t_2]\!]_{\mathsf{S},\mathsf{C}}}{[\![\mathtt{if}\ G\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2]\!]_{\mathsf{S},\mathsf{C}} = \mathtt{if}\ G'\ \mathtt{then}\ [\![t_1]\!]_{\mathsf{S},\mathsf{C}\cup\{G'\}}\ \mathtt{else}\ [\![t_2]\!]_{\mathsf{S},\mathsf{C}\cup\{\neg G'\}}}$$

3.6 Symbolic Update Sets

For symbolic execution, in the same manner as states have been replaced by symbolic states, update sets have to be replaced by *symbolic update sets*.

A *symbolic update set* U is defined as a finite set of *symbolic updates* of the form $((f,\overline{x}),t)$, where $(f,\overline{x})$ is the location to be updated and t is a pe-term to be stored in that location.

State transitions are defined accordingly. In particular, in the *sequel symbolic state* $\mathsf{S}' = \mathsf{S} + \mathsf{U}$ obtained by firing a symbolic update set U in symbolic state S, the symbolic interpretation of each controlled function name $f : T_1, \ldots, T_n \to T$ is, for each $\overline{x} \in \mathcal{T}_1 \times \ldots \times \mathcal{T}_n$,

$$\mathbf{f}_{\mathsf{S}+\mathsf{U}}(\overline{x}) = \begin{cases} t & \text{if } ((f, \overline{x}), t) \in \mathsf{U} \\ \mathbf{f}_{\mathsf{S}}(\overline{x}) & \text{otherwise.} \end{cases}$$

This is clearly a straightforward adaptation of the corresponding definition based on regular update sets (see Sect. 2.1).

The question of consistency is a bit more problematic, as it is generally not possible to decide, by looking at a symbolic update set U, whether it is consistent or not. If U contains two updates (l, t) and (l, t'), its consistency depends on whether $t = t'$ holds in a given state. One way to deal with this problem could be to generate a "consistency condition" consisting of equations, which must be verified in order to prove consistency. However, generating such a formula is relatively complex and the formula can easily become very large, while updates of the same location are usually (in practice) mistakes. Therefore, this approach does not seem very reasonable. This issue is thoroughly analysed in [20] and we follow the authors' conclusion that it is appropriate to adopt a notion of strong consistency ("clash-freedom").

Accordingly, we define a symbolic update set U to be *consistent* if it does not contain two symbolic updates (l, t) and (l, t') of the same location l.

3.7 Partially Evaluated Rules

Partially evaluated rules ("pe-rules") are trees with a structure similar to rules, with an additional pe-rule $\langle \text{upd } \mathsf{U} \rangle$ encapsulating a symbolic update set U:

$$\begin{array}{lll} R ::= & \ldots & \text{(all previously defined rules, see Sect. 2.2 and Table 2.2)} \\ \mid & \langle \text{upd } \mathsf{U} \rangle & \text{(symbolic update set)} \end{array}$$

The idea behind this definition is that, due mainly to the presence of conditionals, it is not always possible to fully evaluate a pe-rule to a symbolic update set. However, if some of its subrules can be fully evaluated, they can be replaced by the special pe-rule $\langle \text{upd } \mathsf{U} \rangle$, which encapsulates the computed update set.

3.8 Symbolic Evaluation of Partially Evaluated Rules

The *symbolic evaluation* $[\![R]\!]_{\mathsf{S},\mathsf{C}}$ of a pe-rule R in a symbolic state S under path condition C is defined in Table 3. It works by traversing the pe-rule tree and produces its result in the same form, without going through an intermediate representation or any other encoding.

For any R, $[\![R]\!]_{\mathsf{S},\mathsf{C}}$ is either a symbolic update set $\langle \text{upd } \mathsf{U} \rangle$ or a conditional. The result $[\![R]\!]_{\mathsf{S},\mathsf{C}}$ of the symbolic pe-rule evaluation is essentially a decision tree representing all possible execution paths, where the inner nodes are conditionals and the leaves are symbolic update sets corresponding to the cumulative state changes resulting from the execution of each path.

Table 3. Symbolic evaluation of pe-rules

$$\frac{}{[\![\texttt{skip}]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \emptyset \rangle} \qquad \frac{}{[\![\langle \text{upd } \mathsf{U} \rangle]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \mathsf{U} \rangle}$$

$$\frac{[\![t_i]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{val } x_i \rangle \text{ for all } i \in \{1,\ldots,n\}}{[\![f(t_1,\ldots,t_n) := t]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \{((f,(x_1,\ldots,x_n)),[\![t]\!]_{\mathsf{S},\mathsf{C}})\} \rangle}$$

$$\frac{[\![t_i]\!]_{\mathsf{S},\mathsf{C}} \neq \langle \text{val } x_i \rangle \text{ for some } i \in \{1,\ldots,n\}}{[\![f(t_1,\ldots,t_n) := t]\!]_{\mathsf{S},\mathsf{C}} \text{ is not defined}}$$

$$\frac{[\![G]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{val } \mathbf{true} \rangle \text{ or } [\![R_1]\!]_{\mathsf{S},\mathsf{C}} = [\![R_2]\!]_{\mathsf{S},\mathsf{C}}}{[\![\texttt{if } G \texttt{ then } R_1 \texttt{ else } R_2]\!]_{\mathsf{S},\mathsf{C}} = [\![R_1]\!]_{\mathsf{S},\mathsf{C}}} \qquad \frac{[\![G]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{val } \mathbf{false} \rangle}{[\![\texttt{if } G \texttt{ then } R_1 \texttt{ else } R_2]\!]_{\mathsf{S},\mathsf{C}} = [\![R_2]\!]_{\mathsf{S},\mathsf{C}}}$$

$$\frac{[\![G]\!]_{\mathsf{S},\mathsf{C}} = G' \qquad G' \notin \{\langle \text{val } \mathbf{true} \rangle, \langle \text{val } \mathbf{false} \rangle\} \qquad [\![R_1]\!]_{\mathsf{S},\mathsf{C}} \neq [\![R_2]\!]_{\mathsf{S},\mathsf{C}}}{[\![\texttt{if } G \texttt{ then } R_1 \texttt{ else } R_2]\!]_{\mathsf{S},\mathsf{C}} = \texttt{if } G' \texttt{ then } [\![R_1]\!]_{\mathsf{S},\mathsf{C}\cup\{G'\}} \texttt{ else } [\![R_2]\!]_{\mathsf{S},\mathsf{C}\cup\{\neg G'\}}}$$

$$\frac{[\![R_1]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \mathsf{U}_1 \rangle \qquad [\![R_2]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \mathsf{U}_2 \rangle}{[\![R_1 \texttt{ par } R_2]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } (\mathsf{U}_1 \cup \mathsf{U}_2) \rangle}$$

$$\frac{[\![R_1]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \mathsf{U}_1 \rangle \qquad [\![R_2]\!]_{\mathsf{S},\mathsf{C}} = \texttt{if } G_2 \texttt{ then } R_{21} \texttt{ else } R_{22}}{[\![R_1 \texttt{ par } R_2]\!]_{\mathsf{S},\mathsf{C}} = [\![\texttt{if } G_2 \texttt{ then } (\langle \text{upd } \mathsf{U}_1 \rangle \texttt{ par } R_{21}) \texttt{ else } (\langle \text{upd } \mathsf{U}_1 \rangle \texttt{ par } R_{22})]\!]_{\mathsf{S},\mathsf{C}}}$$

$$\frac{[\![R_1]\!]_{\mathsf{S},\mathsf{C}} = \texttt{if } G_1 \texttt{ then } R_{11} \texttt{ else } R_{12} \qquad (R_2 \text{ is any rule})}{[\![R_1 \texttt{ par } R_2]\!]_{\mathsf{S},\mathsf{C}} = [\![\texttt{if } G_1 \texttt{ then } (R_{11} \texttt{ par } R_2) \texttt{ else } (R_{12} \texttt{ par } R_2)]\!]_{\mathsf{S},\mathsf{C}}}$$

$$\frac{[\![R_1]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \mathsf{U}_1 \rangle \qquad \mathsf{U}_1 \text{ is inconsistent}}{[\![R_1 \texttt{ seq } R_2]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \mathsf{U}_1 \rangle}$$

$$\frac{[\![R_1]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \mathsf{U}_1 \rangle \qquad \mathsf{U}_1 \text{ is consistent} \qquad [\![R_2]\!]_{\mathsf{S}+\mathsf{U}_1,\mathsf{C}} = \langle \text{upd } \mathsf{U}_2 \rangle}{[\![R_1 \texttt{ seq } R_2]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } (\mathsf{U}_1 \oplus \mathsf{U}_2) \rangle}$$

$$\frac{[\![R_1]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \mathsf{U}_1 \rangle \qquad [\![R_2]\!]_{\mathsf{S}+\mathsf{U}_1,\mathsf{C}} = \texttt{if } G_2 \texttt{ then } R_{21} \texttt{ else } R_{22}}{[\![R_1 \texttt{ seq } R_2]\!]_{\mathsf{S},\mathsf{C}} = [\![\texttt{if } G_2 \texttt{ then } (\langle \text{upd } \mathsf{U}_1 \rangle \texttt{ seq } R_{21}) \texttt{ else } (\langle \text{upd } \mathsf{U}_1 \rangle \texttt{ seq } R_{22})]\!]_{\mathsf{S},\mathsf{C}}}$$

$$\frac{[\![R_1]\!]_{\mathsf{S},\mathsf{C}} = \texttt{if } G_1 \texttt{ then } R_{11} \texttt{ else } R_{12} \qquad (R_2 \text{ is any rule})}{[\![R_1 \texttt{ seq } R_2]\!]_{\mathsf{S},\mathsf{C}} = [\![\texttt{if } G_1 \texttt{ then } (R_{11} \texttt{ seq } R_2) \texttt{ else } (R_{12} \texttt{ seq } R_2)]\!]_{\mathsf{S},\mathsf{C}}}$$

$$\frac{[\![R]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \emptyset \rangle}{[\![\texttt{iterate } R]\!]_{\mathsf{S},\mathsf{C}} = \langle \text{upd } \emptyset \rangle} \qquad \frac{[\![R]\!]_{\mathsf{S},\mathsf{C}} \neq \langle \text{upd } \emptyset \rangle}{[\![\texttt{iterate } R]\!]_{\mathsf{S},\mathsf{C}} = [\![R \texttt{ seq } (\texttt{iterate } R)]\!]_{\mathsf{S},\mathsf{C}}}$$

Note that the symbolic execution of update rules can fail for the reasons already explained in Sect. 3.5 and footnote 9 with respect to application terms (potential aliasing).

The symbolic evaluation for `skip`, update rules $f(t_1, \ldots, t_n) := t$, conditional rules, `iterate`, as well as `par` and `seq` when both arguments are symbolic update sets $\langle \text{upd } \mathsf{U}_1 \rangle$ and $\langle \text{upd } \mathsf{U}_2 \rangle$, has been derived in a straightforward manner from the non-symbolic rule semantics of Table 1, the main differences being that:

- for update rules, in order to avoid the aliasing situation, there are two cases: a success case derived from the rule semantics (when all arguments can be fully evaluated), and a failure case with undefined result (otherwise);
- for conditional rules, besides the "true" and "false" cases, there is a third case that applies when the guard G cannot be fully evaluated: in this case, the conditional remains in place, but its `then` and `else` branches are evaluated symbolically with an updated path condition; furthermore, the conditional is simplified if its `then` and `else` branches are equal.

The remaining cases are R_1 `par` R_2 and R_1 `seq` R_2 where R_1 and/or R_2 are conditional rules. These cases are resolved by making use of the following semantic equivalences (where *op* stands for `par` or `seq`):

$$(\texttt{if } G_1 \texttt{ then } R_{11} \texttt{ else } R_{12}) \mathit{ op } R_2 \simeq \texttt{if } G_1 \texttt{ then } (R_{11} \mathit{ op } R_2) \texttt{ else } (R_{12} \mathit{ op } R_2)$$
$$R_1 \mathit{ op } (\texttt{if } G_2 \texttt{ then } R_{21} \texttt{ else } R_{22}) \simeq \texttt{if } G_2 \texttt{ then } (R_1 \mathit{ op } R_{21}) \texttt{ else } (R_1 \mathit{ op } R_{22})$$

The symbolic evaluation for these cases rewrites, according to the above equivalences, the `par` or `seq` to a conditional, which is in turn evaluated symbolically.

3.9 Reconversion to ASM Transition Rule

After applying the transformation of Table 3, the generated pe-rule should be converted back to an equivalent, syntactically correct ASM transition rule (as defined in Sect. 2.2), not containing any special $\langle \text{upd } \ldots \rangle$ pe-rules nor $\langle \text{val } \ldots \rangle$ and $\langle \text{initial} \ldots \rangle$ pe-terms.

For this purpose, a to-rule operator is introduced (not defined in detail here, as it is quite obvious). It replaces all the $\langle \text{val } x \rangle$ forms, depending on their type, with terms that always evaluate to x (e.g., integer constants if x is a value of type *Integer*). All $\langle \text{initial } (f, \overline{x}) \rangle$ forms are replaced by corresponding application terms. Any $\langle \text{upd } \mathsf{U} \rangle$ is replaced by an appropriate `par` block of update rules.

4 Transforming Turbo ASM into Basic ASM

Using the symbolic pe-rule evaluation $[\![\cdot]\!]$ defined in Table 3, a turbo ASM rule R with `seq` and/or `iterate` subrules (but no submachine calls) can be transformed into an equivalent basic ASM rule R' with no `seq` nor `iterate` as follows[10]:

$$R' = \text{to-rule}[\, [\![\text{to-pe-rule}[R]]\!]_{symstate(S_0^-),\emptyset} \,]$$

[10] The to-pe-rule operator represents the (trivial) conversion of a rule to a pe-rule.

where S_0 is the initial state of the original ASM and S_0^- is a state in which the static function names have the same interpretation as in S_0 and all non-static (i.e. controlled and monitored) function names are uninterpreted.[11]

The transformation is not limited to the symbolic execution of a fixed number n of sequential steps, but keeps rewriting the turbo ASM pe-rule until all `seq` and `iterate` are eliminated. However, it can fail. In particular, it fails if the symbolic evaluation of a pe-term or pe-rule is not defined due to potential aliasing (see Tables 2 and 3 and footnote 9). In the presence of `iterate`, it can also fail due to non-termination, either because the original `iterate` itself does not terminate or because simplify_formula is unable to fully evaluate the termination conditions. Finally, it may fail due to the stronger consistency condition (see Sect. 3.6).

5 Implementation, Examples and Experimental Results

In this section we briefly describe a prototype symbolic execution tool for ASM and present some examples and experimental results obtained using that tool.

5.1 Tool

The techniques presented in this paper have been implemented in a prototype tool, which is publicly available at

`https://github.com/constructum/asm-symbolic-execution`

The tool is written in the functional programming language F# (a dialect of ML for the .NET framework) and consists at the time of writing of about 2200 lines of source code. It uses the Z3 SMT solver [17] via the .NET interface provided by Z3, and does not depend on any other external tool other than Z3.

The internal representation of the ASM language in the tool is a simplified version of the one used in the ASM Workbench [7], a tool written in Standard ML (SML) [16]. The concrete syntax for ASM is based on (a subset of) UASM [1]. The reason not to build the symbolic execution tool on top of the existing ASM Workbench infrastructure is that neither Z3 nor other major SMT solvers provide SML bindings out of the box. Implementing such bindings requires not only a good amount of low-level programming, but also some knowledge of the SMT solver internals. Therefore, it was decided to use F# with the Z3 bindings for .NET to quickly get the prototype running.

The prototype tool includes a simple non-symbolic interpreter for the ASM subset described in this paper, which is essentially an implementation of Sect. 2, and a symbolic interpreter based on the techniques described in Sect. 3. The tool has been written with the aim of making all elements of the techniques presented here easily recognisable in the source code. Accordingly, the tool is a more or

[11] This is necessary because the transformed rule R' must produce the same update set as R in any state of an ASM run. Therefore, it must not depend on the interpretation of any non-static function name.

less direct translation of the content of Sects. 2 and 3 into a functional program, with no particular effort being made to reduce the execution time.

At the time of writing the only data types supported are *Boolean* and *Integer* with the usual basic operations. This is clearly not enough for any interesting applications, but was sufficient to validate the feasibility of the techniques of this paper, which as such are not type-dependent. Adding further data types should be straightforward, as long as they are supported by the SMT solver of choice. Should that not be the case, the symbolic evaluation of terms and rules defined in Sect. 3 could still be applied (by disabling the invocation of the SMT solver), but would produce a large number of redundant paths that can never be taken, as shown by the experiments below (see Sect. 5.3). Furthermore, the SMT solver can in many cases verify properties involving symbolic values, which would not be possible otherwise (see Sect. 5.2).

5.2 Examples

Considering that only *Boolean* and *Integer* are currently supported, the examples were chosen accordingly: matrix multiplication, sorting networks, bubblesort, two slightly different (non-recursive) versions of quicksort, binary search. All these examples can be found in the GitHub repository.

The bubblesort example will be shown here because, due to its simplicity, it provides an easy demonstration of the symbolic execution method. Table 4 shows the ASM *program* (without function declarations). The only function with a defined initial value is n (representing the array size). All other functions have no interpretation in the initial state. However, i, j and *sorted* receive an interpretation during the run as a result of updates, while a remains symbolic.

Table 4. Turbo ASM rule for bubblesort

```
rule Main = [
   i := 0;
   while (i < n) [
      j := n - 1;
      while (j > i) [
         if a(j-1) > a(j) then { a(j-1) := a(j), a(j) := a(j-1) } endif;
         j := j - 1
      ];
      i := i + 1
   ];
   { sorted := true, i := 0 };
   while (i < n - 1) { sorted := sorted and a(i) <= a (i+1), i := i + 1 }
]
```

The last two program lines are not part of the algorithm, but are used to check that the array elements are in ascending order after executing the algorithm[12] (it is not checked that they are a permutation of the initial array though).

The output of the symbolic execution tool (slightly edited to save space by squeezing the parallel update blocks into single lines) is shown in Table 5. An interesting thing to note is that, besides the fact that (with the help of the SMT solver) the "sorted" property could be verified for all execution paths, the generated rule provides a rather efficient sorting algorithm for a special case, derived from a very poor algorithm.

During the experiments it was also noticed that useful insights can sometimes be obtained by merely inspecting the generated rule. For example, in the first of the two non-recursive quicksorts mentioned before, one could clearly see that the stack was in many cases growing to the same size as the array to be sorted (which is clearly undesirable). This may potentially be a useful method for exhaustive testing, for small input sizes, which does not even require writing test cases.

Table 5. Result of symbolic evaluation of bubblesort ($n = 3$)

```
if (a(1) > a(2)) then
   if (a(0) > a(2)) then
      if (a(0) > a(1)) then
         { a(0) := a(2), a(1) := a(1), a(2) := a(0), i := 2, j := 2, sorted := true }
      else
         { a(0) := a(2), a(1) := a(0), a(2) := a(1), i := 2, j := 2, sorted := true }
      endif
   else
      { a(1) := a(2), a(2) := a(1), i := 2, j := 2, sorted := true }
   endif
else
   if (a(0) > a(1)) then
      if (a(0) > a(2)) then
         { a(0) := a(1), a(1) := a(2), a(2) := a(0), i := 2, j := 2, sorted := true }
      else
         { a(0) := a(1), a(1) := a(0), i := 2, j := 2, sorted := true }
      endif
   else
      { i := 2, j := 2, sorted := true }
   endif
endif
```

[12] The proper way to do this would be, of course, to have language constructs to specify properties such as pre- and postconditions, which is outside the scope of the prototype tool. However, it can be seen how symbolic execution may be used for verification of program properties (for finite and sufficiently small input sizes).

5.3 Experimental Results

In this section some experimental results are presented. The following quantities were measured:

1. number of execution paths (leaves) in the decision tree of the generated rule;
2. generated rule size (in number of AST nodes);
3. CPU time for executing the transformation (in seconds).

The first quantity (number of execution paths) is the most interesting, as it shows how well redundant execution paths are eliminated and can be easily compared with the optimum, which in the given examples is known (the other quantities, i.e. rule size and CPU time, are roughly proportional to the number of paths). In the bubblesort example the SMT solver does an extremely good job at reducing the number of paths, which is in fact exactly $n!$. In some of the sorting network examples, instead, this number is slightly higher. We have not yet investigated the reasons for this behaviour, in particular whether this is due to our symbolic execution method, to the SMT solver or to the sorting algorithm itself.

Table 6. Experimental Results: Bubblesort, Sorting Networks

	Bubblesort						Sorting networks					
	No SMT			SMT			No SMT			SMT		
n	paths	size	time	paths	size	time	paths	size	time	paths	size	time
2	2	59	0.05	2	47	0.07	2	43	0.04	2	31	0.05
3	7	347	0.05	6	201	0.08	7	291	0.05	6	147	0.08
4	38	2565	0.25	24	999	0.47	28	1651	0.07	24	783	0.19
5	300	25287	1.79	120	5847	1.68	336	25839	0.86	130	5033	0.99
6	3300	332343	24.41	720	39879	12.78	2432	225987	5.57	910	41615	6.95
7	49031	2792991	303.03	5040	311679	127.68	23960	2615273	72.05	5512	286523	49.79

6 Conclusions

In this paper, we introduced a transformation method using symbolic execution to transform a subset of turbo ASM with `seq` and `iterate` rules (but without submachines) into basic ASM and presented a prototype tool implementing the transformation.

The most basic work that remains to be done for this method to be useful for practical applications is to add support for further data types and more ASM rules, including the (possibly recursive) submachines of turbo ASM of [5]. We would expect this to be relatively straightforward. Dealing with the aliasing problem, even though various techniques are known from the symbolic execution literature, is likely to be more challenging. Finally, the method seems most useful in combination with other techniques (such as annotation of ASM rules with properties, verification by model checking, compilation) in order to enhance the

capabilities of existing ASM tools. Therefore, the proposed method should be integrated with those tools.

A first application that could be envisioned is the enhancement of the model checking interfaces of existing ASM tools to support turbo ASM. Furthermore, the potential of symbolic execution for checking program properties as an improved form of unit testing (as hinted to in Sect. 5.2) may be worth exploring. Finally, it may be possible to apply the proposed symbolic execution framework, with some modifications, to basic sequential ASM (rather than turbo ASM) in order to implement a simple bounded model checker to verify some properties of ASM models directly, i.e. without translating the models for verification by existing model checkers.

Acknowledgments. Thanks to Egon Börger for various discussions and interactions which renewed my interest in ASM, to Sylvain Lelait for his helpful comments on a preliminary version of this paper and to the anonymous reviewers for their insightful remarks and good suggestions, which I tried to address as much as possible.

References

1. Arcaini, P., Bonfanti, S., Dausend, M., Gargantini, A., Mashkoor, A., Raschke, A., Riccobene, E., Scandurra, P., Stegmaier, M.: Unified syntax for abstract state machines. In: Butler, M., Schewe, K.D., Mashkoor, A., Biro, M. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 231–236. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_14
2. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: A way to link high-level ASM models to low-level NuSMV specifications. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) Abstract State Machines, Alloy, B and Z. pp. 61–74. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_6
3. Baldoni, R., Coppa, E., Cono D'Elia, D., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3) (may 2018). https://doi.org/10.1145/3182657, https://doi.org/10.1145/3182657
4. Börger, E.: The Abstract State Machines method for modular design and analysis of programming languages. Journal of Logic and Computation **27**(2), 417–439 (2017). https://doi.org/10.1093/logcom/exu077
5. Börger, E., Schmid, J.: Composition and submachine concepts for sequential ASMs. In: Clote, P.G., Schwichtenberg, H. (eds.) Computer Science Logic. pp. 41–60. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
6. Börger, E., Stärk, R.: Abstract State Machines. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
7. Del Castillo, G.: The ASM Workbench - A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models. Ph.D. thesis, Universität Paderborn (2000), https://github.com/constructum/the-asm-workbench/blob/main/doc/2000-Del-Castillo-The-ASM-Workbench.pdf
8. Del Castillo, G., Winter, K.: Model checking support for the ASM high-level language. In: Graf, S., Schwartzbach, M. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS 2000). p. 331–346. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_23

9. Dorfmeister, D., Ferrarotti, F., Fischer, B., Haslinger, E., Ramler, R., Zimmermann, M.: An approach for safe and secure software protection supported by symbolic execution. In: Kotsis, G., Tjoa, A.M., Khalil, I., Moser, B., Mashkoor, A., Sametinger, J., Khan, M. (eds.) Database and Expert Systems Applications - DEXA 2023 Workshops. pp. 67–78. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-39689-2_7
10. Fruja, N.G., Stärk, R.F.: The hidden computation steps of turbo abstract state machines. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003, Proceedings. pp. 244–262. Springer (2003). https://doi.org/10.1007/3-540-36498-6_14
11. Glesner, S., Goos, G., Zimmermann, W.: Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and architecture of verifying compilers). it - Information Technology **46**(5), 265–276 (2004). https://doi.org/doi:10.1524/itit.46.5.265.44799
12. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Specification and Validation Methods, pp. 9–36. Oxford University Press (January 1995)
13. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (jul 1976). https://doi.org/10.1145/360248.360252
14. Lezuo, R.: Scalable Translation Validation. Ph.D. thesis, Vienna University of Technology (2014), https://www.complang.tuwien.ac.at/tbfg
15. Ma, G.Z.S.: Model checking support for CoreASM: model checking distributed abstract state machines using Spin. Master's thesis, Simon Fraser University (2007), https://summit.sfu.ca/item/8056
16. Milner, R., Harper, R., MacQueen, D., Tofte, M.: The Definition of Standard ML (Revised Edition). MIT Press (1997)
17. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. Paun, V.A., Monsuez, B., Baufreton, P.: Integration of symbolic execution into a formal abstract state machines based language. IFAC-PapersOnLine **50**(1), 11251–11256 (2017). https://doi.org/10.1016/j.ifacol.2017.08.1610, 20th IFAC World Congress
19. Schellhorn, G., Ahrendt, W.: The WAM case study: verifying compiler correctness for Prolog with KIV. In: Bibel, W., Schmitt, P.H. (eds.) Automated deduction - a basis for applications: volume 3 - applications. Kluwer Academic Publishers (1998)
20. Schellhorn, G., Ernst, G., Pfähler, J., Bodenmüller, S., Reif, W.: Symbolic execution for a clash-free subset of ASMs. Science of Computer Programming **158**, 21–40 (2018). https://doi.org/10.1016/j.scico.2017.08.014
21. Topor, R.W.: Interactive program verification using virtual programs. Ph.D. thesis, University of Edinburgh (1975), https://era.ed.ac.uk/handle/1842/6610
22. Zimmermann, W., Weißbach, M.: A framework for modeling the semantics of synchronous and asynchronous procedures with abstract state machines. Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday pp. 326–352 (2021)