# Decomposition of verification of machine programs based on control-state Abstract State Machines.

Werner Gabrisch
Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
06099 Halle/Saale, Germany, gabrisch@informatik.uni-halle.de

## ABSTRACT

We are presenting a method verifying programs based on extended control-state Abstract State Machines (ASM). Programs are special initial states in ASM's. The aim is to prove that every run holds an algebraic specification of functions. The proof of different functions could be made by independent steps.

## 1. INTRODUCTION

Abstract State Machines (ASM) are well-suited to specify the behavior of real machines. They are able to describe the semantic of processors and programming languages [2, 1]. In reality the behavior of processors can be defined by a fixed control-state ASM. Programming language applications define a fixed semantic [6, 11]. The concrete behavior of a running computer is specified on a base on static description (the program). The program is static because it does not change while the processor operates or the compiler translates a fixed source-code.

The aim of a program is to realize a new function, new in a sense that processor or language can't do it elementary. The functions are realized by a sequence of steps of a machine. Functions are only realized on demand. There is a start-condition and there is an other end-condition defined when the function is complete executed. The start-condition and the end-condition are defined as an abstract data type. In higher programing languages we can use Hoare-calculus to prove the correctness of programs. The behavior of processors can't be proved by original Hoare-calculus. In [4] we show an application to use Hoare-triplets to prove the correctness of control-state ASMs, right now we are proving the correctness of a program on a given machine executed as a part of a run. The proof of correctness is showing that this part of each run respects the axioms of the function in the abstract data type.

To guarantee that a machine is realizing a function it requires a program. The program is fixed, never changing while executed. The algebraic definition is a simple set of formulas (normally equations), defining assignments in storage. The aim of a program is to realize a complex function. This function is algebraic defined. In many applications this function is too complex to prove. Complex function are normally based on many more simple functions. Our method is able to make the proof step by step, from simple functions to complex ones.

Based on the method in [4] to show the correctness of the behavior of control-state ASMs realize a function by respecting axiomatic definition of a function. The control-state ASM is separated in two parts. The first one is a graph defining the states in nodes. The second part defines the state transition. The proof is done by checking the labeling of the states by Hoare like triples [7].

## 2. CONTROL-STATE ASMS AND VERIFICATION

An ASM is defined [2] as $(\Sigma, \Phi_{\text{Init}}, R_0)$ where $\Sigma$ is a signature, $\Phi_{\text{Init}}$ is a set of $\Sigma$-formulas (the *initial conditions*), and $R_0$ is a *transition rule*.

In it's special form as control-state ASM the *transition rule* $R_0$ is recursively defined as:

$$R_k ::= \begin{cases} f(t_1, ..., t_n) := t_0 & (1) \\ R_i; R_j & (2) \\ \textbf{if } cs = i \textbf{ then } R_n;\ cs := j \textbf{ endif} & (3) \\ \textbf{if } cs = i\ \wedge \varphi \textbf{ then } cs := j \textbf{ endif} & (4) \end{cases}$$

The machine starts with a state (an algebra) $q_0 \in \Phi_{\text{Init}}$.

The first part (1) is called an update. It's semantic is to change the interpretation of $f(t_1, ..., t_n)$ to $t_0$. $t_i$ are basic terms defining a fixed value. If we do the update we get a new state (algebra). First we are collecting the updates only.

The second part (2) defines a sequence of rules. All updates of $R_i$ and $R_j$ are united to one set of updates.

The third part (3) is an condition for updates ($R_n$). The updates in $R_n$ are only united if $cs = i$ holds in the state $q_i \models cs = i$. The control-state $cs$ has also to change ($cs := j$).

The fourth part (4) has an additional condition $\varphi$ for the collection of the only update $cs := j$.

BCI'13 September 19-21, Thessaloniki, Greece. Copyright © 2013 for the individual papers by the papers' authors.
Copying permitted only for private and academic purposes. This volume is published and copyrighted by its editors.

The *transition rule* $R_0$ is used to generate a set of updates. This set contains all collected updates. If the whole rule $R_0$ is evaluatet than the generatet set is compleat. The changes of the interpretations are done in one step. We get a new interpretation of functions, a new algebra, the next state.

The ASM starts with a concrete algebra as state $q_0 \in \Phi_{\text{Init}}$. Based on this state the formulas in the if-clausels holds or hold not. So we get a set of updates in this state. Next we use the updates to change the interpretations of functions. We get a new state and so on.

If the set of updates is empty the machine stops.

Especially control state ASM's are isomorph to a graph, where all constants used in $cs = i$ on the right side are the nodes. All updates of $cs := j$ are the edges from the old interpretation of $cs$ was $i$ to node $j$. Based of ruletype (3) the edges from $i$ to $j$ can be labeled by a fixed set of updates $U(i, j)$ because the conditions $(cs = i)$ defines node $i$. Based on ruletype (4) we label the edges by $\varphi$.

In a last step we have to label the nodes by formulas $\psi_i$ by using Hoare in ruletype (3) for all edges from $i$ to $j$ the nodes and updates has to be a valid Hoare-triplet.

$$\psi_i \qquad U(i,j) \qquad \psi_j \tag{5}$$

For all edges based on ruletyp (4) the defined condition $\varphi$ is additionally correct in the aim state $j$.

$$\psi_i \qquad \varphi \qquad \psi_j = \phi_i \wedge \varphi \tag{6}$$

The labels holdng in all reachable states (algebras) $q_i$ if the abpropeate control state $cs = i$ holds in node $i$. If all triples are correct in (5) and (6) than the whole labeling holds for all algebras of every run of the machine. The prove was shown in [4].

## 3. EXTENDED CONTROL-STATE ASMS

To check programs we have to extend control-state ASMs definition by a classification of states. A class of states can be defined by an algebraic equation. This equation defines a homomorph picture of the states. Using a constant-operation $bef_k$ as an ambassador of the class. defined by $ct(cs_i) = ct(cs_j)$. We are using a term depending on $cs$ to update the control-state. Everything else is the same. All proves shown in [4] are correct. The rules form is now:

$$R_k ::= \begin{cases} f(t_1, ..., t_n) := t_0 & (7) \\ R_i; R_j & (8) \\ \textbf{if } ct(cs) = bef_x \textbf{ then } R_n;\ cs := t_j(cs) \textbf{ endif} & (9) \\ \textbf{if } ct(cs) = bef_x\ \wedge\ \varphi\textbf{then } cs := t_i(cs) \textbf{ endif} & (10) \end{cases}$$

The state graphs nodes are defined by $cs$ and the edges can be labeled like before.

In reality labeling large systems is very difficult, therefore we try to build it step by step.

## 4. SEPARATING CONTROL-STATE ASMS

To describe a part of a run we add a formula $\sigma(cs)$ defining a subset of control-states. The aim is to terminate all states in a run where $\sigma(cs) = true$. The formula should define an abstraction of the given machine. For all initial and final states formula $\sigma(cs)$ should be false.

If the ASM is defined as:

$$\mathfrak{A} = (\Sigma, \Phi_{\text{Init}}, R_0) \tag{11}$$

The ASM separated by $\sigma(cs)$ is:

$\mathfrak{A} = (\Sigma, \Phi_{\text{Init}}, R_\sigma; R_{\neg\sigma})$

where $\quad R_\sigma ::= if\ \sigma(cs)\ then\ R_0;$ (12)

and $\quad R_{\neg\sigma} ::= if\ \neg\sigma(cs)\ then\ R_0;$ (13)

This machine generates the same runs because all generated updates are the same. All runs starts based on updates generated from (13) because of the definition of $\sigma(cs)$ (false in initial states). As long as $\sigma(cs)$ is false only the rule $R_{\neg\sigma(cs)}$ controls the states transitions until $\sigma(cs)$ evaluates to true. Now $R_{\sigma(cs)}$ controls the transitions until $\sigma(cs)$ evaluates to false. etc.

The aim is to terminate all states where $\sigma(cs)$ evaluates to true. If $\sigma(cs)$ evaluates to true in rulepart (13) there must be an update $cs := k_{\neg\sigma}$. The same happens if $\sigma(cs)$ is evaluated to false in rulepart (12) there has to be an $cs := k_\sigma$. We assume that there is exactly one state $k_\sigma$ for one $k_{\neg\sigma}$.

The idea is to change the transition rule where $cs := k_{\neg\sigma}$ is updated by an update of $cs := k_\sigma$ and an additional update $y := newfnc(x)$ in form (3). The same can be done in form (4) by adding a new state.

The change is a correct abstraction of (11) iff the control-state ASM

$$\mathfrak{A}_\sigma = (\Sigma, \Psi_{k_\sigma}, R_\sigma) \tag{14}$$

stopps in state $k_{\neg\sigma}$ and the machine

$\mathfrak{A}^*_\sigma = (\Sigma, \Psi_{k_\sigma}, if\ cs = k_\sigma\ then\ y := newfnk(x); cs := k_{\neg\sigma})$
(15)

is an abstraction of $\mathfrak{A}_\sigma$ for all assignments of $x$ allowed in $\Psi_{k_\sigma}$ and

$$\Psi_{k_\sigma} \qquad y := newfnk(x) \qquad \Psi_{k_{\neg\sigma}} \tag{16}$$

is a correct Hoare triple in $\mathfrak{A}_\sigma$.

To show this we use the method discribed in [4] and in a short form in section 2. $\mathfrak{A}$ starts in a state labeled $\Psi_{k_\sigma}$ and if we are able to label $\Psi_{k_{\neg\sigma}}$ as a substitution of $\Psi_{k_\sigma}[newfnc(y)/x] = \Psi_{k_{\neg\sigma}}$ the prove of abstraction is done.

If there exists only one state $k_{\neg\sigma}$ the rulepart (12) never generate updates. It can be eleminated from the transition rules. If there is an other state $k_{\neg\sigma}$ the same can be done to eleminate the connection to state in $\sigma(cs)$. If all are terminated no state in $\sigma(cs)$ is reachable.

The ASM

$\mathfrak{A} = (\Sigma, \Phi_{\text{Init}}, if\ \ cs = k_\sigma\ \ then\ y := newfnk(x);\ \ cs := k_{\neg\sigma}\ \ else\ R^*_0\ )$
(17)

is an abstraction of

$$\mathfrak{A} = (\Sigma, \Phi_{\text{Init}}, R_\sigma; xxxxR_{\neg\sigma}) \tag{18}$$

## 5. EXAMPLE

We will verify a simple part of a program. We use the same example like in [4], the computation of the greatest common divider (gcd) of two numbers. Next we will show that a program running on a fixed machine is correct. We need a machine, a program and an algebraic definition of gcd.

A simple basic machine:

**sorts** $OPC, INT, BOOL$ (defines the sorts)
**ops** $\ldots, -1, 0, 1, \ldots :$ $\rightarrow INT$
$+, - :$ $INT \times INT$ $\rightarrow INT$
$true, false$ $\rightarrow BOOL$
$>, ==:$ $INT \times INT$ $\rightarrow BOOL$
$ld, st, sub, br, brlt, call, ret :$ $\rightarrow OPC$
$bef :$ $OPC \times INT$ $\rightarrow INT$
$adr :$ $INT$ $\rightarrow INT$
$mem :$ $INT$ $\rightarrow INT$
$pc :$ $\rightarrow INT$
**trans** (defines the transition rules)
**if** $op(mem(pc)) = ld$ **then**
$ac := mem(stk + adr(mem(pc))); pc := pc + 1$
**if** $op(mem(pc)) = st$ **then**
$mem(stk + adr(mem(pc)))) := ac; pc := pc + 1$
**if** $op(mem(pc)) = sub$ **then**
$ac := ac - mem(stk + adr(mem(pc))));$
$zero := ac == mem(stk + adr(mem(pc))));$
$lt := ac < mem(stk + adr(mem(pc)))); pc := pc + 1$
**if** $op(mem(pc)) = br$ **then**
$pc := adr(mem(pc)))$
**if** $op(mem(pc)) = breq \wedge zero = true$ **then**
$pc : adr(mem(pc))$ **else** $pc := pc + 1$
**if** $op(mem(pc)) = brlt \wedge lt = true$ **then**
$pc : adr(mem(pc))$ **else** $pc := pc + 1$
**if** $op(mem(pc)) = call$ **then**
$pc := adr(mem(pc)); stk := stk - 1; mem(stk) = pc + 1$
**if** $op(mem(pc)) = ret$ **then**
$pc := mem(stk + 1); stk := stk + 1$

Figure 1: Abstract state machine

Specification of the greatest common divider:

**ops** $gcd :$ $INT \times INT$ $\rightarrow INT$
**axiom** **gcd ( n , n ) = n**
**n>m → gcd ( n-m , m ) = gcd ( n , m )**
**n<m → gcd ( n , m-n ) = gcd ( n , m )**

Figure 2: Algebraic specification of the static function gcd.

**ops** $gcd :$ $INT \times INT$ $\rightarrow INT$
**axiom** **mem(gcd+0) = bef ( ld , 2 )**
**mem(gcd+1) = bef ( sub , 3 )**
**mem(gcd+2) = bef ( breq , gcd+9 )**
**mem(gcd+3) = bef ( brlt , gcd+6 )**
**mem(gcd+4) = bef ( st , 2 )**
**mem(gcd+5) = bef ( br , gcd )**
**mem(gcd+6) = bef ( neg , 0 )**
**mem(gcd+7) = bef ( st , 3 )**
**mem(gcd+8) = bef ( br , gcd )**
**mem(gcd+9) = bef ( ret , 0 )**

Figure 3: A program to verify its bebavior on the machine figure 1

**pre** $top(stack) = n, top(pop(stack)) = m, n > 0, m > 0$
**call gcd**
**post** $top(stack) = gcd(n, m)$

Figure 4: The behavior to prove (dashed line in figure 5).

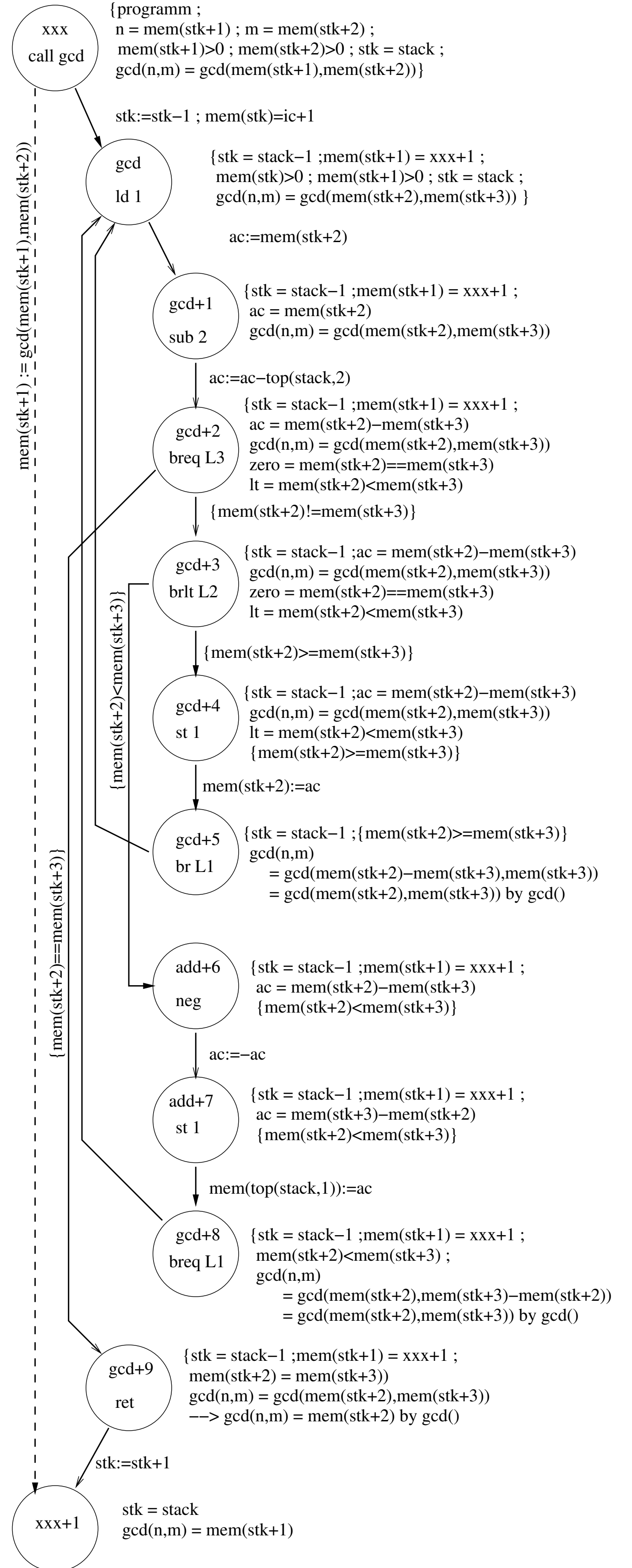


Figure 5: Control-state-graph

The machine in figure 1 defines only the instructions we need in the example. We use a simple one address machine with accumulator *ac*. Each instruction is defined by one code and an address. It is defined by function $bef : OPC \times INT \rightarrow INT$. The function $op : INT \rightarrow OPC$ and $adr : INT \rightarrow INT$ are the inverse to $bef$. The stack is placed in memory and *stk* points to the appropriate address. All load and store instructions are relative to the stack pointer *stk*. We have two flags *lt* and *zero*, set by *sub*-instruction. It is easy to extend this machine to a real processor. All important types of operations are in this example.

The abstract data type is defined in figure 2. Other specifications like NAT are not shown.

The program is shown in figure 3. We have to prove that is it correct. To show this we define an Hoare triple like shown in figure 4. The parameters are on the top of the stack and they are greather than zero. Additionally we have to show the correct handling of the stackpointer and we have to guarantee the existence of the program in memory.

At the end we need the separator $\sigma$ to define the parts of runs we need to terminate. This is the execution of *gcd*,

$$\sigma = pc \geq gcd \wedge pc \leq gcd + 9. \quad (19)$$

We have to start by $bef(CALL, gcd)$ this instruction occurs at any time on any address (named *xxx*). The opcode *op*(*mem*(*ic*)) is CALL and the ASM generates the updates {pc:=adr(mem(pc));stk:=stk-1;mem(stk):=pc+1}.
Next state is adr(mem(xxx)) = gcd and there is an edge from xxx to gcd. The edge is labeled by the updates {stk:=stk-1;mem(stk):=pc+1}. The first $pc := adr(mem(pc))$ is not necessary because it is defined by the graph.

Figure 5 presents the control-graph and the labeling of all edges. They are made in the same simple way like the first call-instruction. If the way of the graph is splitting like in node *gcd*+2 we get the conditions transferred by the machine *flags*.

Now the labeling of the states by Hoare can be done. This is like Hoare-proving in higher label languages. We only need the assignment-rule.

# 6. INDEPENDENCE OF SEPARATORS

The separation of control-states are independent if the set of states $\sigma_i(cs)$ are disjunct for all $i$. There is no order in the updates because they are handeled as a set. Each update in all rules is selected by the **if**-constructs. Only the conditions decide which updates are in the generated set of updates. If there are two separators $\sigma_1(cs)$ and $\sigma_2(cs)$ and the defined control-state-sets are disjunct than the separations produce the same sets of updates. This applies because of the capabilities of the if-condition:

$$if\ \sigma_1(cs)\ then\ R_1\ else\ if\ \sigma_2(cs)\ then\ R_2\ else\ R_3 \quad (20)$$

is the same like

$$if\ \sigma_2(cs)\ then\ R_2\ else\ if\ \sigma_1(cs)\ then\ R_1\ else\ R_3 \quad (21)$$

This behavior can be extended to any finite number of separators.

If there is a recursive use of generated functions and the recursion stops after a limited depth of calls the deepest call can be proven as correct (start of induction). So the correctness of a call can be used as precondition to prove the correctness of higher calls.

In this way the explosion of states in more complex functions can be reduced. The only precondition is that the sets $\sigma_i(cs)$ are all disjunct and the recursition stops. In the example $\sigma(cs)$ are the locations in memory used by the programcode. The only condition is that different programms using different storage.

# 7. RELATED WORK

Winter uses a model-checking approach for verifying ASMs [13, 3, 15, 16, 5]. The main idea is to abstract ASMs to finite state machines that can be checked automatically by a model-checker such as e.g. SMV. The specification language for the model-checkers is similar to the ASM transition rules. This work only allows dynamic constants of a finite domain and therefore may restrict the expressiveness of the assertions. E.g. checking the greatest common divisor example will become difficult if the pre- and postconditions consider all natural numbers and not just a finite subset of it.

Stärk develops a verification calculus for general ASMs[12, 2] based on temporal logic. Therefore it is possible to verify general temporal behaviour. With his approach, it must be proven that control-state invariants are always satisfied. The approach in this paper enables to prove control-state invariants by Hoare-like prove rules individually for each edge in the control-state graph – independent of the proof for the other edges. This seems to be more simple than to prove control-state invariants using temporal logic calculi.

Schellhorn provides a refinement-based approach for verification of ASMs [10, 8, 9]. His approach is better suited for a top-development of ASMs but it cannot be applied to bottom-up or middle-out approaches that are also frequently used in Software Engineering.

All of these calculues use concrete algebras. They require the concrete interpretation of functions. The method presented here does not need any algebra. It shows the correct implementation of an abstract data type on a given machine. It is not the aim to show the correctnes of an ASM; the aim is to prove the behavior of a machine in special initial states (programs).

# 8. CONCLUSIONS

We have shown that a Hoare-style verification of programs running on control state ASMs is possible. Verifying programs by Hoare is not limited to higher programming languages. Hoare is extended to all types of programs running on a control-state ASM. So it is possible to use Hoare on low level programs like assembler too.

At last we describe a method to transform a static definition of storage assigments to functions. In a given basic machine we define storage as fixed (the program), this part of storage is unavailable for other usage. Based on this storage we can show that the machine extends its behavior by new functions.

The approach is surprisingly simple: find control-state invariants for each control-state *cs*, i.e., assertions that are satisfied whenever the ASM is in control-state *s*, and verify

individually the correctness of the state transitions for each edge in the control-state graph. The method is used by an algebraic specification of the ASM. No algebra is used to make the proof. In this way we will extend the declared method to check abstractions of ASM's by defining new static operations by signature and axioms and prove that behavior of the ASM respect the given axioms.

## 9. REFERENCES


[1] E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 353–404. Springer, 1998.

[2] E. Börger and R. Stärk. *Abstract State Machines*. Springer-Verlag, 2003.

[3] G. del Castillo and K. Winter. Model checking support for the asm high-level language. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, TACAS '00, London, UK, 2000. Springer-Verlag.

[4] Werner Gabrisch and Wolf Zimmermann. A hoare-style verification calculus for control state asms. In *Proceedings of the Fifth Balkan Conference in Informatics*, BCI '12, pages 205–210, New York, NY, USA, 2012. ACM.

[5] A. Gawanmeh, S. Tahar, and K. Winter. Formal verification of asm designs using the mdg tool. In *Software Engineering and Formal Methods, 2003. Proceedings. First International Conference on*, pages 210–219. IEEE, 2003.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[8] G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Fakultät für Informatik, 1999.

[9] G. Schellhorn. Verification of asm refinements using general forward simulation. *Journal of Universal Computer Science*, 7(11):952–979, 2001.

[10] G. Schellhorn and W. Ahrendt. Reasoning about abstract state machines: The wam case study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[11] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer, 2001.

[12] R. Stärk and S. Nanchen. A logic for abstract state machines. In *Computer Science Logic*, pages 217–231. Springer, 2001.

[13] K. Winter. Model checking for abstract state machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

[14] K. Winter. *Model checking abstract state machines*. PhD thesis, Technische Universität Berlin, 2001.

[15] K. Winter. Model checking with abstract types. *Electronic Notes in Theoretical Computer Science*, 55(3):382–393, 2001.

[16] K. Winter and R. Duke. Model checking object-z using asm. In *Integrated Formal Methods*, pages 165–184. Springer, 2002.