A security logic for abstract state machines

Report**Author(s):**

Nanchen, Stanislas; Stärk, Robert F.

Publication date:

2003

Permanent link:

<https://doi.org/10.3929/ethz-a-006714738>

Rights / license:

In Copyright - Non-Commercial Use Permitted

Originally published in:

Technical Report / ETH Zurich, Department of Computer Science 423

A Security Logic for Abstract State Machines

Stanislas Nanchen and Robert F. Stärk

Computer Science Department, ETH Zürich, CH-8092 Zürich, Switzerland
nanchen@inf.ethz.ch and staerk@inf.ethz.ch

Abstract. We extend the logic for Abstract State Machines by a read predicate that allows to make precise statements about the accesses of locations of an ASM. The logic can be used to prove security properties of ASMs like that the machine does not read locations containing critical information or that all accesses of the machine are in a well-defined region of the abstract memory. The new read predicate is also useful for proving refinements of parallel ASMs to sequential C-like programs. The logic is complete for hierarchical ASMs and still sound for turbo ASMs. It is integrated in the ASMKeY theorem prover.

1 Introduction

The states of an Abstract State Machine (ASM) are algebraic first-order structures. Algebraic structures are widely used in mathematics but are understood here as a kind of generalized memory. A mathematical term like $f(x + f(y))$ has a similar meaning like an expression `mem[x + mem[y]]` in computer science. The function f is dynamic and can change its values at specific arguments. A pair $\langle f, a \rangle$ is called a *location* for the function f and the value $f(a)$ is then considered to be the *content* of the location, like `mem[a]` is the content of the memory cell with address a . Updating a function means putting a new content into a location of the function. Formally, an update is a pair $\langle l, v \rangle$ where l is a location and v is a value. Dynamic functions are more general than ordinary arrays. They can be of arity greater than one and the arguments can be of any type not just non-negative integers.

When a transition rule of an ASM is evaluated in a given state, it produces a set of updates. If the updates do not conflict, then they are applied to the state and yield the next state in the computation. Hence the change from one state to the next state is fully described by the set of updates. In order to produce the updates, however, a transition rule has also to *read* some locations of the state. The reading of locations is not observable from outside when we look just at the sequence of states in the computation of an ASM.

In the world of ASMs the states are global states of a system. It has always been assumed that a transition rule can read every location of the state. Therefore the main focus has always been on function updates and the dynamic change of states. The reading of locations has been considered as not so important. The only exception we know is the definition of micro-steps of an ASM in [2].

When it comes to applications, however, the reading of locations is as important as the updating. It is very important to know that certain processes cannot read certain locations, for example a private key stored somewhere in the memory of a computer or a smart card. Therefore it is very important to have tools and formal systems which allow to prove that programs do not access certain locations or that all accesses of a program are in a certain range of the memory.

In this paper we extend the logic for ASMs introduced in [14]. The new logic allows to express and prove properties like ‘the program P does not read the location $f(x)$ ’ or ‘whenever program P reads location $f(x)$, then $0 \leq x < 10$ ’. Thus the logic has a read/access predicate in addition to the update predicate that has already been there from the beginning. The meaning of the new predicate $\text{acc}(P, f, x)$ is that the program P accesses (reads) the function f at the argument x , whereas the old predicate $\text{upd}(P, f, x, y)$ still means that the program P updates the function f at the argument x to the new value y .

The logic is designed in a modular way such that a local analysis of a program is sufficient to assess possible vulnerabilities as opposed to a global analysis of all uses of a program. Possible applications of the logic include: applet isolation on smart cards, memory safeness of high-performance code like network filters, elimination of array bound checks, sequentialization of parallel programmes.

The logic is implemented as a sequent calculus in the ASMKeY interactive prover. ASMKeY is a derivative of the interactive prover of the KeY Project [1]. The KeY project aims to create an integrated tool for modeling, specifying and verifying object-oriented programmes, in particular JavaCard programmes.

1.1 Related work

In [5] dynamic logic (DL), in particular Java Card DL from the KeY Project [1], is used to express the *Secure Information Flow* problem in a generic program logic. The problem is standardly defined as follows ([4], [9]): given a program P with variables partitioned into two classes, low-security variables L and high-security variables H , it is impossible to deduce anything about the initial values of the high-security variables H from the observations of the initial and final values of the low-security variables L . A typical DL formula in [5] for expressing the secure information flow with a program P , one low-security variable l and one high-security variable h is the following:

$$\forall l \exists r \forall h \langle P \rangle r = l \quad (1)$$

It asserts that for every initial value of l , there exists a final value r such that after termination of P the variable l has the value r independently of the initial value of h . Hence, no information flows from h to l . As we can see, the main concern of the Secure Information Flow problem is rather *leak of information* than *memory access violation*. Therefore, the authors of [5] have no read predicate or equivalent. In Sect. 3.3 we show how the problem of secure information flow can be formalized in the logic for ASMs.

Proof-carrying code [11,12] is based on the following principles: in order to assure code safety, a code-consumer publishes a security policy that is required to be respected by code-producers. The security policy is given in particular by a Verification Condition Generator (VCGen) for programs with an abstract “machine” of the environment in which the programs run (e.g. a micro-processor). After coding the program, a code-producer uses the VCGen to produce a first-order formula asserting that the program respects the security policy. The code-producer then has to prove that formula and delivers a binary program together with the proof of its correctness. The code-consumer can check the proof and gain trust that the code is secure.

The proof-carrying code approach is a general method applicable from assembler programmes to high-level programming languages. However it has been used and implemented mainly for low-level applications, for instance, the safety of kernel extensions [11]. In such setting, the safety policy is defined by a *safe* interpreter for an assembly language in which the kernel extensions are written. Safe means that the interpreter stops if the programmes attempt to issue an instruction that would violate safety.

Such safe interpreter is typically implemented by having pre- and post-conditions on execution for each instruction. An important example for us are the preconditions for reading and writing to memory which are expressed with the help of two predicates **saferd** and **safewr**. These predicates assert that a read, resp. write, in memory are safe. The VCGen for this security policy is then derived from the safe interpreter.

2 Access sets of transition rules

An ASM consists of a vocabulary Σ , an initial state $\mathfrak{A}$ for Σ , a rule declaration for each rule name and a distinguished rule name of arity zero called the main rule of the machine. A rule declaration for a rule name r is an expression $r(x) \triangleq P$, where P is a transition rule in which there are no free occurrences of variables except of x . Rule declarations can be recursive, i.e., the rule name r may appear in the body P . We denote the terms of Σ by s, t and the formulas of Σ by φ, ψ . The function names in Σ are declared either as static or dynamic. A term is static, if it does not contain dynamic functions. The transition rules P, Q are syntactic expressions generated as follows (the function arguments can be read as vectors):

1. *Skip Rule:* **skip**
Meaning: Do nothing.
2. *Update Rule:* $f(s) := t$
Syntactic condition: f is a dynamic function name of Σ
Meaning: In the next state, the value of f at s is updated to t .
3. *Block Rule:* **P par Q**
Meaning: P and Q are executed in parallel.
4. *If Rule:* **if φ then P else Q**
Meaning: If φ is true, then execute P , otherwise execute Q .

5. *Let Rule:* **let** $x = t$ **in** P
Meaning: Assign the value of t to x and execute P .
6. *Forall Rule:* **forall** x **with** φ **do** P
Meaning: Execute P in parallel for each x satisfying φ .
7. *Sequence Rule:* P **seq** Q
Meaning: P and Q are executed sequentially, first P and then Q .
8. *Try Rule:* **try** P **else** Q
Meaning: If P is consistent, execute P , else execute Q .
9. *Call Rule:* $r(t)$
Meaning: Call transition rule r with parameters t .

The guards φ in the if-rule and the forall-rule are boolean combinations of equations between terms (quantifier-free formulas). We allow the strict (logical) connectives $\wedge$ and $\vee$ as well as the conditional (short circuit) connectives $\&\&$ and $||$ (in AsmL 2, ‘**and then**’ and ‘**or else**’, see [6]).

The semantics of transition rules is defined in terms of update sets U , V and access sets A , B . An access set is a set of accesses. The intended meaning of an access $\langle f, a \rangle$ for a state $\mathfrak{A}$ is that the machine reads the interpretation of the dynamic function f at point a . Contrary to the update sets, as we are concerned with read properties only, there is no need of a notion of consistency for access sets.

Definition 1 (Access). *An access for $\mathfrak{A}$ is a location for a dynamic function, i.e. a pair $\langle f, a \rangle$ where f is an n -ary dynamic function name and a in an element of the base set of $\mathfrak{A}$.*

Why the restriction of accesses to dynamic functions? There are two main reasons. First, static functions like ‘+’ correspond to the hard-wired functions of a processor. Hence, an agent is allowed to evaluate them at every possible argument. If the static functions is partial, then the result may be **undef**. In applications where we want to prove, for example, that a program does not generate overflows for arithmetical operations, we have to declare these operations as dynamic functions explicitly. The only consequence will then be that certain principles and rules of the logic are more restricted than they were with static arithmetical functions. Hence, the user has to work more in the proofs.

The second reason is technical one. If we would consider accesses to static functions, then the substitution principle would be violated. Consider the following valid formula:

$$\forall x \forall y (\text{acc}(f(x) < 0, f, y) \rightarrow y = x)$$

If we instantiate the variable x by the term $f(0)$ then we obtain the following formula that is not true in general, since for the evaluation of $f(f(0))$ the function f has to be read at argument 0 as well as at $f(0)$:

$$\forall y (\text{acc}(f(f(0)) < 0, f, y) \rightarrow y = f(0))$$

$$\begin{aligned} \text{AccSet}(x, \mathfrak{A}, \zeta) &= \emptyset \\ \text{AccSet}(f(t), \mathfrak{A}, \zeta) &= \begin{cases} \{\langle f, \llbracket t \rrbracket_{\zeta}^{\mathfrak{A}} \rangle\} \cup \text{AccSet}(t, \mathfrak{A}, \zeta), & \text{if } f \text{ is dynamic;} \\ \text{AccSet}(t, \mathfrak{A}, \zeta), & \text{otherwise.} \end{cases} \end{aligned}$$

Table 1. Access sets of terms.

$$\begin{aligned} \text{AccSet}(s = t, \mathfrak{A}, \zeta) &= \text{AccSet}(s, \mathfrak{A}, \zeta) \cup \text{AccSet}(t, \mathfrak{A}, \zeta) \\ \text{AccSet}(\neg\varphi, \mathfrak{A}, \zeta) &= \text{AccSet}(\varphi, \mathfrak{A}, \zeta) \\ \text{AccSet}(\varphi \star \psi, \mathfrak{A}, \zeta) &= \text{AccSet}(\varphi, \mathfrak{A}, \zeta) \cup \text{AccSet}(\psi, \mathfrak{A}, \zeta) \quad \text{for } \star \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \\ \text{AccSet}(\varphi \&\& \psi, \mathfrak{A}, \zeta) &= \begin{cases} \text{AccSet}(\varphi, \mathfrak{A}, \zeta) \cup \text{AccSet}(\psi, \mathfrak{A}, \zeta), & \text{if } \llbracket \varphi \rrbracket_{\zeta}^{\mathfrak{A}} = \text{true;} \\ \text{AccSet}(\varphi, \mathfrak{A}, \zeta), & \text{otherwise.} \end{cases} \\ \text{AccSet}(\varphi || \psi, \mathfrak{A}, \zeta) &= \begin{cases} \text{AccSet}(\varphi, \mathfrak{A}, \zeta), & \text{if } \llbracket \varphi \rrbracket_{\zeta}^{\mathfrak{A}} = \text{true;} \\ \text{AccSet}(\varphi, \mathfrak{A}, \zeta) \cup \text{AccSet}(\psi, \mathfrak{A}, \zeta), & \text{otherwise.} \end{cases} \end{aligned}$$

Table 2. Access sets of rule guards.

We have to remember here that the quantifier axioms and the substitution principle of the logic for ASMs in [14] have to be restricted to static terms anyway. For example, the axiom $\forall x \varphi \rightarrow \varphi_{\frac{t}{x}}$ has to be restricted to static terms t . Consider the following instance of a valid formula $\forall x \varphi$:

$$\forall x (x = 0 \rightarrow [f(0) := 1]x = 0)$$

To derive the formula $\varphi_{\frac{f(0)}{x}}$ using the non-static term $f(0)$ is not allowed, since this would yield a non-valid formula:

$$f(0) = 0 \rightarrow [f(0) := 1]f(0) = 0$$

Hence, we see that axioms and rules of the logic are already restricted (compared to classical first-order logic). Since we do not want to restrict them further, we do not consider accesses to locations of static functions in Def. 1.

In each step of its computation, an ASM accesses locations of $\mathfrak{A}$ for producing update sets. To compute these update sets, it has to evaluate terms and formulas that occur in the transition rules. For a term t , we denote by $\text{AccSet}(t, \mathfrak{A}, \zeta)$ the access set that is needed to compute the value $\llbracket t \rrbracket_{\zeta}^{\mathfrak{A}}$ in state $\mathfrak{A}$ with respect to the environment ζ that assigns elements of $\mathfrak{A}$ to the variables of t . The exact definition can be found in Table 1. Note, that if t is a static term, then its access set is always empty.

Table 2 contains the definition of the access sets that are needed to compute the truth-values of rule guards φ . The connectives $\&\&$ and $||$ are conditional (short-circuit) operators that do not evaluate the second argument if the result

can already be determined by the value of the first argument. Example:

$$\begin{aligned}\text{AccSet}(x < 2 \ \&\& \ f(x) < 2, \mathfrak{A}, \{x \mapsto 1\}) &= \{\langle f, 1 \rangle\} \\ \text{AccSet}(x < 2 \ \&\& \ f(x) < 2, \mathfrak{A}, \{x \mapsto 2\}) &= \emptyset\end{aligned}$$

In the second case the operand $f(2) < 2$ is not evaluated, hence the access set is empty.

The semantics of transition rules is inductively defined in Table 3 using a predicate $\text{yields}(P, \mathfrak{A}, \zeta, U, A)$ with the meaning that the rule P yields in state $\mathfrak{A}$ with respect to the environment ζ the update set U and the access set A . Note that, since cyclic rule declarations are allowed, there may be no such sets at all. The update and access set are unique, since we consider deterministic transition rules only. Non-determinism has to come from outside via monitored functions that are updated by the environment. What is new compared to the definition of the ‘yields’ predicate in [3] is the addition of the access sets A, B . The ‘yields’ predicate has five arguments here whereas it has only four arguments in [3].

By $\mathfrak{A} + U$ we denote in Table 3 the state obtained from $\mathfrak{A}$ by firing the updates in U ; $U \oplus V$ denotes the sequential composition of update sets where the updates of V override possible updates in U . Note that the access sets of the guard φ in a forall-rule have to be computed for every element $a \in |\mathfrak{A}|$ and not just for the range I . The range of a formula in the forall-rule is defined as follows:

$$\text{range}(x, \varphi, \mathfrak{A}, \zeta) = \{v \in |\mathfrak{A}| : \llbracket \varphi \rrbracket_{\zeta[x \mapsto v]}^{\mathfrak{A}} = \text{true}\}$$

Example: the rule

$$\text{forall } x \text{ with } 0 \leq x \ \&\& \ x < 10 \ \&\& \ 0 < f(x) \text{ do } f(x) := 0$$

yields the access set $\{\langle f, 0 \rangle, \langle f, 1 \rangle, \dots, \langle f, 9 \rangle\}$ although the range I may be a proper subset of $\{0, 1, \dots, 9\}$.

3 Extending the logic for ASMs by an access predicate

We extend the basic logic for ASMs introduced in [14] by a new access predicate. The logic is an extension of first-order predicate logic by a modal operator and several atomic predicates. The formulas of the logic are generated by the following grammar:

$$\begin{aligned}\varphi, \psi ::= & s = t \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \forall x \varphi \mid \exists x \varphi \mid [P]\varphi \mid \text{upd}(P, f, s, t) \\ & \mid \text{def}(P) \mid \text{Con}(P) \mid \text{inv}(P, f, s) \mid \text{acc}(t, f, s) \mid \text{acc}(\varphi, f, s) \mid \text{acc}(P, f, s)\end{aligned}$$

The formula $[P]\varphi$ means that, if P is defined and consistent, then φ is true in the next state after executing P ; $\text{upd}(P, f, s, t)$ means that P is defined and updates the dynamic function f at the argument s to the new value t ; $\text{def}(P)$ means that P is defined and yields a consistent or inconsistent update set; $\text{Con}(P)$ asserts that P is defined and yields a consistent update set (strong consistency); $\text{inv}(P, f, s)$ means that P is defined and does not update f at argument s . The

$\text{yields}(\text{skip}, \mathfrak{A}, \zeta, \emptyset, \emptyset)$	
$\text{yields}(f(s_1, \dots, s_n) := t, \mathfrak{A}, \zeta, \{\langle \langle f, \langle \llbracket s_1 \rrbracket_{\zeta}^{\mathfrak{A}}, \dots, \llbracket s_n \rrbracket_{\zeta}^{\mathfrak{A}} \rangle \rangle, \llbracket t \rrbracket_{\zeta}^{\mathfrak{A}} \rangle\}, A)$ where $A = \bigcup_{i=1}^n \text{AccSet}(s_i, \mathfrak{A}, \zeta) \cup \text{AccSet}(t, \mathfrak{A}, \zeta)$	
$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U, A) \quad \text{yields}(Q, \mathfrak{A}, \zeta, V, B)}{\text{yields}(P \text{ par } Q, \mathfrak{A}, \zeta, U \cup V, A \cup B)}$	
$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U, A)}{\text{yields}(\text{if } \varphi \text{ then } P \text{ else } Q, \mathfrak{A}, \zeta, U, A \cup \text{AccSet}(\varphi, \mathfrak{A}, \zeta))}$	if $\llbracket \varphi \rrbracket_{\zeta}^{\mathfrak{A}} = \text{true}$
$\frac{\text{yields}(Q, \mathfrak{A}, \zeta, V, B)}{\text{yields}(\text{if } \varphi \text{ then } P \text{ else } Q, \mathfrak{A}, \zeta, V, B \cup \text{AccSet}(\varphi, \mathfrak{A}, \zeta))}$	if $\llbracket \varphi \rrbracket_{\zeta}^{\mathfrak{A}} = \text{false}$
$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto v], U, A)}{\text{yields}(\text{let } x = t \text{ in } P, \mathfrak{A}, \zeta, U, A \cup \text{AccSet}(t, \mathfrak{A}, \zeta))}$	where $v = \llbracket t \rrbracket_{\zeta}^{\mathfrak{A}}$
$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto v], U_v, A_v) \text{ for each } v \in I}{\text{yields}(\text{forall } x \text{ with } \varphi \text{ do } P, \mathfrak{A}, \zeta, \bigcup_{v \in I} U_v, \bigcup_{v \in I} A_v \cup \Phi)}$ where $I = \text{range}(x, \varphi, \mathfrak{A}, \zeta)$ and $\Phi = \bigcup_{v \in \mathfrak{A} } \text{AccSet}(\varphi, \mathfrak{A}, \zeta[x \mapsto v])$	
$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U, A) \quad \text{yields}(Q, \mathfrak{A} + U, \zeta, V, B)}{\text{yields}(P \text{ seq } Q, \mathfrak{A}, \zeta, U \oplus V, A \cup B)}$	if U is consistent
$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U, A)}{\text{yields}(P \text{ seq } Q, \mathfrak{A}, \zeta, U, A)}$	if U is inconsistent
$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U, A)}{\text{yields}(\text{try } P \text{ else } Q, \mathfrak{A}, \zeta, U, A)}$	if U is consistent
$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U, A) \quad \text{yields}(Q, \mathfrak{A}, \zeta, V, B)}{\text{yields}(\text{try } P \text{ else } Q, \mathfrak{A}, \zeta, V, A \cup B)}$	if U is inconsistent
$\frac{\text{yields}(P \stackrel{t}{x}, \mathfrak{A}, \zeta, U, A)}{\text{yields}(r(t), \mathfrak{A}, \zeta, U, A)}$	where $r(x) \triangleq P$ is a rule declaration of M
$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U, A)}{\text{yields}(P, \mathfrak{A}, \zeta, U)}$	

Table 3. The semantics of transition rules.

new predicates $\text{acc}(t, f, s)$ and $\text{acc}(\varphi, f, s)$ assert that the term t resp. the formula φ accesses the function f at the argument s . The predicate $\text{acc}(P, f, s)$ asserts that P is defined and accesses f at the argument s .

If φ is a rule guard, then by $\hat{\varphi}$ we denote the formula that is obtained from φ by replacing the conditional $\&\&$ by the logical conjunction $\wedge$ and the conditional $||$ by the logical disjunction $\vee$. From the logical point of view the formula $\hat{\varphi}$ is equivalent to φ . From the operational point of view, however, they are different. The formula $\hat{\varphi}$ accesses more locations than φ .

The exact semantics of the modal operator and the basic predicates already present in [14] is given in Table 4. The semantics of the access predicate for

$$\begin{aligned}
\llbracket [P]\varphi \rrbracket_{\zeta}^{\mathfrak{A}} &= \begin{cases} true, & \text{if } \llbracket \varphi \rrbracket_{\zeta}^{\mathfrak{A}+U} = true \text{ for each consistent update set } U \\ & \text{such that } \text{yields}(P, \mathfrak{A}, \zeta, U) \text{ is derivable in Table 3;} \\ false, & \text{otherwise.} \end{cases} \\
\llbracket \text{def}(P) \rrbracket_{\zeta}^{\mathfrak{A}} &= \begin{cases} true, & \text{if there exists an update set } U \text{ such that} \\ & \text{yields}(P, \mathfrak{A}, \zeta, U) \text{ is derivable in Table 3;} \\ false, & \text{otherwise.} \end{cases} \\
\llbracket \text{Con}(P) \rrbracket_{\zeta}^{\mathfrak{A}} &= \begin{cases} true, & \text{if there exists a consistent update set } U \text{ such that} \\ & \text{yields}(P, \mathfrak{A}, \zeta, U) \text{ is derivable in Table 3;} \\ false, & \text{otherwise.} \end{cases} \\
\llbracket \text{upd}(P, f, s, t) \rrbracket_{\zeta}^{\mathfrak{A}} &= \begin{cases} true, & \text{if there exists an update set } U \text{ such that} \\ & \text{yields}(P, \mathfrak{A}, \zeta, U) \text{ and } \langle \langle f, \llbracket s \rrbracket_{\zeta}^{\mathfrak{A}} \rangle, \llbracket t \rrbracket_{\zeta}^{\mathfrak{A}} \rangle \in U; \\ false, & \text{otherwise.} \end{cases} \\
\llbracket \text{inv}(P, f, s) \rrbracket_{\zeta}^{\mathfrak{A}} &= \begin{cases} true, & \text{if there exists an update set } U \text{ such that} \\ & \text{yields}(P, \mathfrak{A}, \zeta, U) \text{ and } \langle \langle f, \llbracket s \rrbracket_{\zeta}^{\mathfrak{A}} \rangle, v \rangle \notin U \text{ for all } v \in |\mathfrak{A}|; \\ false, & \text{otherwise.} \end{cases}
\end{aligned}$$

Table 4. The semantics of the modal operator and the basic predicates.

terms, formulas and transition rules is given in Table 5. Note, that some of the predicates can be defined in terms of others. For example, the strong consistency predicate can be defined as follows (with fresh variables x, y, z):

$$\text{Con}(P) \leftrightarrow \text{def}(P) \wedge \bigwedge_{f \text{ dyn.}} \forall x, y, z (\text{upd}(P, f, x, y) \wedge \text{upd}(P, f, x, z) \rightarrow y = z)$$

The invariance predicate can be defined in terms of other predicates, too:

$$\text{inv}(P, f, s) \leftrightarrow \text{def}(P) \wedge \forall x \neg \text{upd}(P, f, s, x)$$

3.1 Axioms and rules of for the extended logic

What has to be added to the axioms and rules of the basic system of [14] (which can also be found in [3, p. 319]) are the following axioms and principles:

X. Axioms for acc:

18. Axioms AT1–AT3 in Table 6
19. Axioms AF1–AF5 in Table 7
20. Axioms AR1–AR9 in Table 8
21. $\text{acc}(P, f, x) \rightarrow \text{def}(P)$
22. $\text{Con}(P) \wedge \text{def}(Q) \wedge \text{not\_read}(P, Q) \rightarrow \text{same\_after}(P, Q) \wedge [P]\text{def}(Q)$

The predicate ‘not_read’ means that Q does not read locations updated by P :

$$\text{not\_read}(P, Q) \triangleq \bigwedge_{f \text{ dyn.}} \forall x (\text{acc}(Q, f, x) \rightarrow \text{inv}(P, f, x))$$

$$\begin{aligned}
\llbracket \text{acc}(t, f, s) \rrbracket_{\zeta}^{\mathfrak{A}} &= \begin{cases} \text{true}, & \text{if } \langle f, \llbracket s \rrbracket_{\zeta}^{\mathfrak{A}} \rangle \in \text{AccSet}(t, \mathfrak{A}, \zeta); \\ \text{false}, & \text{otherwise.} \end{cases} \\
\llbracket \text{acc}(\varphi, f, s) \rrbracket_{\zeta}^{\mathfrak{A}} &= \begin{cases} \text{true}, & \text{if } \langle f, \llbracket s \rrbracket_{\zeta}^{\mathfrak{A}} \rangle \in \text{AccSet}(\varphi, \mathfrak{A}, \zeta); \\ \text{false}, & \text{otherwise.} \end{cases} \\
\llbracket \text{acc}(P, f, s) \rrbracket_{\zeta}^{\mathfrak{A}} &= \begin{cases} \text{true}, & \text{if there exist } U \text{ and } A \text{ with } \text{yields}(P, \mathfrak{A}, \zeta, U, A) \\ & \text{and } \langle f, \llbracket s \rrbracket_{\zeta}^{\mathfrak{A}} \rangle \in A; \\ \text{false}, & \text{otherwise.} \end{cases}
\end{aligned}$$

Table 5. The semantics of the `acc` predicate.

$$\begin{aligned}
\text{AT1. } & \neg \text{acc}(y, f, x) \\
\text{AT2. } & \text{acc}(f(t), f, x) \leftrightarrow x = t \vee \text{acc}(t, f, x) \\
\text{AT3. } & \text{acc}(f(t), g, x) \leftrightarrow \text{acc}(t, f, x) \quad \text{if } f \neq g
\end{aligned}$$

Table 6. Axioms of the `acc` predicate for terms.

$$\begin{aligned}
\text{AF1. } & \text{acc}(s = t, f, x) \leftrightarrow \text{acc}(s, f, x) \vee \text{acc}(t, f, x) \\
\text{AF2. } & \text{acc}(\neg \varphi, f, x) \leftrightarrow \text{acc}(\varphi, f, x) \\
\text{AF3. } & \text{acc}(\varphi \star \psi, f, x) \leftrightarrow \text{acc}(\varphi, f, x) \vee \text{acc}(\psi, f, x) \quad \text{where } \star \in \{\wedge, \vee, \rightarrow\} \\
\text{AF4. } & \text{acc}(\varphi \&\& \psi, f, x) \leftrightarrow \text{acc}(\varphi, f, x) \vee (\hat{\varphi} \wedge \text{acc}(\psi, f, x)) \\
\text{AF5. } & \text{acc}(\varphi \mid \mid \psi, f, x) \leftrightarrow \text{acc}(\varphi, f, x) \vee (\neg \hat{\varphi} \wedge \text{acc}(\psi, f, x))
\end{aligned}$$

Table 7. Axioms of the `acc` predicate for rule guards.

$$\begin{aligned}
\text{AR1. } & \neg \text{acc}(\text{skip}, f, x) \\
\text{AR2. } & \text{acc}(g(s) := t, f, x) \leftrightarrow \text{acc}(s, f, x) \vee \text{acc}(t, f, x) \\
\text{AR3. } & \text{acc}(P \text{ par } Q, f, x) \leftrightarrow \text{def}(P \text{ par } Q) \wedge (\text{acc}(P, f, x) \vee \text{acc}(Q, f, x)) \\
\text{AR4. } & \text{acc}(\text{if } \varphi \text{ then } P \text{ else } Q, f, x) \leftrightarrow \text{def}(\text{if } \varphi \text{ then } P \text{ else } Q) \wedge \\
& \quad (\text{acc}(\varphi, f, x) \vee (\hat{\varphi} \wedge \text{acc}(P, f, x)) \vee (\neg \hat{\varphi} \wedge \text{acc}(Q, f, x))) \\
\text{AR5. } & \text{acc}(\text{let } y = t \text{ in } P, f, x) \leftrightarrow \quad \text{if } y \notin \text{FV}(t) \\
& \quad \text{def}(\text{let } y = t \text{ in } P) \wedge (\text{acc}(t, f, x) \vee \exists y (y = t \wedge \text{acc}(P, f, x))) \\
\text{AR6. } & \text{acc}(\text{forall } y \text{ with } \varphi \text{ do } P, f, x) \leftrightarrow \\
& \quad \text{def}(\text{forall } y \text{ with } \varphi \text{ do } P) \wedge \exists y (\text{acc}(\varphi, f, x) \vee (\hat{\varphi} \wedge \text{acc}(P, f, x))) \\
\text{AR7. } & \text{acc}(P \text{ seq } Q, f, x) \leftrightarrow \\
& \quad (\text{acc}(P, f, x) \wedge [P] \text{def}(Q)) \vee (\text{Con}(P) \wedge [P] \text{acc}(Q, f, x)) \\
\text{AR8. } & \text{acc}(\text{try } P \text{ else } Q, f, x) \leftrightarrow \\
& \quad (\text{acc}(P, f, x) \wedge (\text{Con}(P) \vee \text{def}(Q))) \vee (\text{def}(P) \wedge \neg \text{Con}(P) \wedge \text{acc}(Q, f, x)) \\
\text{AR9. } & \text{acc}(r(t), f, x) \leftrightarrow \text{acc}(P \frac{t}{x}, f, x) \quad \text{if } r(x) \triangleq P \text{ is a rule declaration of } M
\end{aligned}$$

Table 8. Axioms of the `acc` predicate for transition rules.

The predicate ‘same_after’ means that the updates and the reads of Q are the same after executing P :

$$\text{same\_after}(P, Q) \triangleq \bigwedge_{f \text{ dyn.}} \left\{ \begin{array}{l} \forall x, y (\text{upd}(Q, f, x, y) \leftrightarrow [P]\text{upd}(Q, f, x, y)) \\ \forall x (\text{acc}(Q, f, x) \leftrightarrow [P]\text{acc}(Q, f, x)) \end{array} \right.$$

The notions of logical consequence and formal derivability are defined as usual. If M is an ASM and Ψ a set of sentences, then $\Psi \models_M \varphi$ means that φ is true in every structure that makes all formulas in Ψ true; $\Psi \vdash_M \varphi$ means that there exists a finite subset $\Theta \subseteq \Psi$ such that the formula $\bigwedge \Theta \rightarrow \varphi$ is derivable using the axioms and rules of the logic.

In order to show that the logic is still sound one has to verify that the new axioms in Tables 6–8 are valid. For Axiom 22 the following lemma is used.

Lemma 1. *If $\text{yields}(P, \mathfrak{A}, \zeta, U, A)$ and W is a consistent update set that does not contain updates for locations in A , then $\text{yields}(P, \mathfrak{A} + W, \zeta, U, A)$.*

Proof. Note first that the values of terms and rule guards depend only on the locations in their access sets. Let W be a consistent update set that does not updates locations in $\text{AccSet}(t, \mathfrak{A}, \zeta)$. Then we have

$$\llbracket t \rrbracket_{\zeta}^{\mathfrak{A}} = \llbracket t \rrbracket_{\zeta}^{\mathfrak{A} + W} \quad \text{and} \quad \text{AccSet}(t, \mathfrak{A}, \zeta) = \text{AccSet}(t, \mathfrak{A} + W, \zeta).$$

A similar property holds for rule guards.

The lemma can then be proved by induction on the definition of the ‘yields’ predicate in Table 3. $\square$

Theorem 1 (Soundness of the logic). *If $\Psi \vdash_M \varphi$, then $\Psi \models_M \varphi$.*

An ASM is called *hierarchical* if it does not contain cycles in the dependency graph of rule declarations. Hierarchical ASMs could also be called *recursion-free* ASMs. The completeness proof in [14] for hierarchical ASMs can then be extended to the acc predicate. It can even be shown that the logic is a definitional extension of FOL.

Theorem 2 (Completeness of the logic). *Let M be a hierarchical ASM. If $\Psi \models_M \varphi$, then $\Psi \vdash_M \varphi$.*

The Axiom 22 is not used in the completeness proof. In case of hierarchical ASMs it can be derived from the other axioms.

3.2 Application: Sequentialization of ASMs

If ASMs are refined to programs of a traditional imperative language, then all parallel compositions have to be sequentialized. The following principle may be useful to prove the correctness of such refinement steps. We first define what it means that ‘ Q does not overwrite updates of P ’:

$$\text{not\_over}(P, Q) \triangleq \bigwedge_{f \text{ dyn.}} \forall x, y (\text{upd}(P, f, x, y) \rightarrow \text{inv}(Q, f, x) \vee \text{upd}(Q, f, x, y))$$

The following sequentialization principle can be derived from Axiom 22:

$$\text{Con}(P) \wedge \text{def}(Q) \wedge \text{not\_read}(P, Q) \wedge \text{not\_over}(P, Q) \rightarrow (P \text{ par } Q) \equiv (P \text{ seq } Q)$$

It says, that under the above conditions, the parallel composition of P and Q is equivalent to the sequential composition, i.e. it yields the same updates and reads the same locations. Other approaches to sequentialization of ASMs can be found in [7, 13].

3.3 Memory access vs. information flow

Let l be a location of a dynamic function in a state $\mathfrak{A}$. We say that a transition rule P does not depend on the location l in $\mathfrak{A}$, if P yields the same update set whatever the content of l is. More precisely, if P yields the update set U in state $\mathfrak{A}$ under ζ , then for every element $v \in |\mathfrak{A}|$, P yields the same set U in the state $\mathfrak{A} + \{\langle l, v \rangle\}$ under ζ .

It is intuitively clear that if a program does not access a location l , then it does not depend on the location l . The converse is not true. A program may read a location l and nevertheless not depend on the location l , for example if it simply ignores the content of l .

That the updates of a program P do not depend on the location $f(x)$ can be expressed in the logic as follows (where a, b, y are fresh variables):

$$\text{not\_dep\_upd}(P, f, x) \triangleq \forall a, b, y (\text{upd}(P, f, a, b) \leftrightarrow [f(x) := y] \text{upd}(P, f, a, b))$$

The following formula is a special case of Axiom 22:

$$\text{def}(P) \wedge \neg \text{acc}(P, f, x) \rightarrow \text{not\_dep\_upd}(P, f, x) \quad (2)$$

Since the logic is complete for hierarchical ASMs the above principle is derivable from the basic axioms and rules in case of hierarchical ASMs.

We have defined that a program does not depend on a location $f(x)$, if it produces the same update sets whatever the content of $f(x)$ is. A weaker notion would be to require that the contents of the locations different from $f(x)$ after executing the program do not depend on the content of $f(x)$. In other words, the result of applying the updates does not depend on $f(x)$. This can be expressed in the logic as follows:

$$\text{not\_dep\_step}(P, f, x) \triangleq \forall a (a \neq x \rightarrow \exists b \forall y [f(x) := y] [P] f(a) = b)$$

The two notions are different, since a program can produce trivial updates that do not change the contents of locations. For example, the updates of the program

$$\text{if } f(0) = 0 \text{ then } f(1) := f(1)$$

depend on the location $f(0)$ but the next state is always identical to the present state. The following implication, however, is valid:

$$\text{not\_dep\_upd}(P, f, x) \rightarrow \text{not\_dep\_step}(P, f, x) \quad (3)$$

The formula can easily be derived already in the basic logic of [14].

The formula $\text{not\_dep\_step}(P, f, x)$ is the translation of the information flow formula 1 of Sect. 1.1 into the logic for ASMs. The location $f(x)$ is a high-security variable whereas the locations $f(a)$ with $a \neq x$ are low-security variables. It is not possible to translate formula 1 directly, since in DL variables can be updated by programs whereas in the logic for ASMs variables are read-only variables that cannot be updated by transition rules.

That a program does not depend on a location l should also include that the termination of the program and the consistency of the computed update sets do not depend on l . Otherwise, an adversary could get information on l based on error messages about conflicting updates. That the termination and consistency of P do not depend on the location $f(x)$ can be expressed as follows:

$$\text{not\_dep\_con}(P, f, x) \triangleq \forall y (\text{Con}(P) \leftrightarrow [f(x) := y] \text{Con}(P))$$

Remember that the predicate $\text{Con}(P)$ includes the termination of P (strong consistency).

Joshi and Leino's semantic definition of secure information flow of [9] can be applied to ASMs as follows:

$$\text{JoshiLeino}(P, f, x) \triangleq \forall y ((P \text{ seq } H) \simeq ((H \text{ seq } P) \text{ seq } H))$$

where y is not free in P and H is the program $f(x) := y$ that updates $f(x)$ to an arbitrary value y . The symbol $\simeq$ means that the transition rules are equivalent. It can be defined as follows (see [14]):

$$P \simeq Q \triangleq (\text{Con}(P) \leftrightarrow \text{Con}(Q)) \wedge \forall a \forall b ([P]f(a) = b \leftrightarrow [Q]f(a) = b)$$

The relationship between Joshi and Leino's definition and the predicates we have introduced so far is as follows:

$$\text{JoshiLeino}(P, f, x) \leftrightarrow \text{not\_dep\_con}(P, f, x) \wedge \text{not\_dep\_step}(P, f, x) \quad (4)$$

This equivalence is valid for arbitrary transition rules P and can be derived in the basic logic.

3.4 Access sets and critical terms

In [8], Gurevich derives the sequential ASM thesis from three postulates about sequential algorithms. The third postulate, the uniformly-bounded-exploration postulate, says that an algorithm A examines only a bounded number of elements in any state. There must exist a finite set T of ground terms — depending on the algorithm only and not on the initial state — such that the next computation step of the algorithm depends only on that part of the state which can be accessed via terms in T . Whenever two states $\mathfrak{A}$ and $\mathfrak{B}$ coincide over T , then $\Delta(A, \mathfrak{A}) = \Delta(A, \mathfrak{B})$, where $\Delta(A, \mathfrak{A})$ is the set of non-trivial updates that have to be applied to state $\mathfrak{A}$ in order to obtain the next state in the computation of the algorithm A . The terms in T are called *critical terms* of A .

What can be said on the relationship between critical terms and read/access of locations? In general only sequential ASMs satisfy the uniformly-bounded-exploration postulate. In our setup these are hierarchical ASMs that do not contain the forall-rule. Only for this class of ASMs the existence of a finite set of critical terms is guaranteed.

A first but obviously wrong conjecture could be that the set of locations accessed by a sequential ASM is contained in T . More precisely, that if the location $\langle f, a \rangle$ is read by the ASM, then there exists a critical term $f(s)$ where a is the value of s . This is not true as the following example shows:

if $f(0) = 0$ **then** $f(1) := f(1)$ **else** $f(1) := f(1)$

In this case, the empty set is a possible candidate for the set of critical terms, but the locations $f(0)$, 0, 1 that are accessed by the program are not in this set. The problem is that the program reads the locations $f(0)$, 0, 1 but does not depend on them.

The second conjecture, that the locations a program depends on are contained in the set of critical terms, is also wrong. A possible set of critical terms for the program

if $f(f(0)) = 0$ **then** $f(0) := 1$

is $T = \{0, 1, f(f(0))\}$. Obviously the program depends on $f(0)$ but the term $f(0)$ is not in T (it is however a subterm of a term in T).

3.5 Maps in AsmL

In AsmL 2, a map f is a partial function and the application $f(x)$ throws an exception if the argument x is not in the domain of f . In the following example, an empty function **f** is declared. When **f** is applied to the argument 3, an `IndexOutOfBoundsException` is thrown:

```
var f as Map of Integer to Integer = {->}
Main()
  WriteLine(f(3))
```

On the theory side it is usually assumed that the value of $f(x)$ is *undef* if x is not in the domain of f . Hence, if the dynamic functions of a program P are encoded in AsmL as elements of the predefined type **Map** and the program should run in AsmL without throwing exceptions, then one has to prove the following formula:

$$\bigwedge_{f \text{ dyn.}} \forall x (\text{acc}(P, f, x) \rightarrow f(x) \neq \text{undef})$$

The formula asserts that P only accesses defined locations of dynamic functions. Hence no `IndexOutOfBoundsException` will be thrown when P is evaluated.

In AsmL 2 it is possible to test whether an element x belongs to the domain of a map f using the expression ' x in f '. The question now is whether ' x in f ' should be considered as an access of the location $f(x)$ or not. If ' x in f ' is

translated as $f(x) \neq \text{undef}$, then it is definitely an access according to our definitions in Tables 1 and 2. If ‘ x in f ’ is not considered to be an access of $f(x)$, then one has to extend the language of the logic and allow also atomic formulas ‘ t in f ’, where $\text{AccSet}(t \text{ in } f, \mathfrak{A}, \zeta) = \text{AccSet}(t, \mathfrak{A}, \zeta)$.

3.6 Access and definedness

It is not possible to make statements in our logic about the accesses of non-terminating transition rules. For example, if the rule

$$r(x) \triangleq \text{if } 0 < f(x) \text{ then } r(x + 1)$$

is called with $r(0)$, it might not terminate and access an infinite number of locations of f . We think that this is no problem, since the single transitions of ASMs are always terminating. Otherwise, there is something wrong with the modeling of the system. Every proof of an ASM should therefore start with an invariant φ that implies the termination and consistency of the machine: $\varphi \rightarrow \text{Con}(M) \wedge [M]\varphi$.

4 Implementation in ASMKeY

ASMKeY [10] is a theorem prover for ASMs derived from the KeY Project [1] theorem prover. The ASMKeY prover is based on the logic for ASMs of [14]. The underlying sequent calculus has been extended by the new `acc` predicates. The new available properties are now applied in case studies, in particular the memory access properties (violation and non-violation) and the sequentialization properties.

To illustrate these two kinds of properties, we will use the MERGESORT algorithm in Fig. 4. It uses the short-circuit operators `&&` and `||`. As opposed to MERGESORT, we will refer by MERGESORTV (for violation) to the variant with the usual $\wedge$ and $\vee$.

4.1 Memory access

Although we had formally proved using ASMKeY that MERGESORTV is correct, at run-time, in AsmL, it generated an `IndexOutOfBoundsException`: the dynamic function g was being accessed at $r + 1$. Actually, the violating access was made during the computation of the second condition of the MERGECOPYV rule in the test $g(i) \leq g(j)$.

It is easy to see that, if there exists an index i in the first half of the array, where $f(i)$ is greater than $f(j)$ for every j in the second half of the array, then a memory violation will occur. In fact, one can see that, in such a case, the merging of the two sorted subarrays will consume at first the second subarray

```

MERGESORT( $l, r$ ) =
  if  $l < r$  then
    let  $m = \lfloor (l + r)/2 \rfloor$  in
      (MERGESORT( $l, m$ ) par MERGESORT( $m + 1, r$ )) seq
      MERGE( $l, m, r$ )

MERGE( $l, m, r$ ) =
  (forall  $i$  with  $l \leq i \leq r$  do  $g(i) := f(i)$ ) seq
  MERGECOPY( $l, m, m + 1, r, l$ )

MERGECOPY( $i, m, j, r, k$ ) =
  if  $k \leq r$  then
    if ( $i \leq m \ \&\& \ j \leq r \ \&\& \ g(i) \leq g(j)$ ) || ( $r < j$ ) then
       $f(k) := g(i)$  par MERGECOPY( $i + 1, m, j, r, k + 1$ )
    else
       $f(k) := g(j)$  par MERGECOPY( $i, m, j + 1, r, k + 1$ )

```

Fig. 1. The MERGESORT algorithm.

and will cause the formula $g(i) \leq g(j)$ to be evaluated with $j = r + 1$. One can even formally prove the following formula:

$$\begin{aligned}
& \forall l, r, m \ (l < r \wedge m = (l + r)/2 \\
& \quad \wedge \exists i \ (l \leq i \leq m \wedge \forall j \ (m + 1 \leq j \leq r \rightarrow f(j) < f(i))) \\
& \quad \rightarrow \text{acc}(\text{MERGESORT}(l, r), g, r + 1))
\end{aligned}$$

If we use MERGESORT instead of MERGESORTV, then we can prove that there are no accesses of the dynamic function g outside the interval $[l, r]$:

$$\forall l, r, x \ (\text{acc}(\text{MERGESORT}(l, r), g, x) \rightarrow l \leq x \wedge x \leq r)$$

For the dynamic function f , it is easy to see that, in MERGESORT as well as in MERGESORTV, we have never access violation, since f is only accessed during the copy of f into g .

4.2 Sequentialization

Using the principle of section 3.2, we can prove in ASMK_eY that MERGESORT and its sequentialization SEQMERGESORT are equivalent:

$$\forall l, r \ (\text{MERGESORT}(l, r) \simeq \text{SEQMERGESORT}(l, r))$$

The SEQMERGESORT is obtained from MERGESORT as a refinement by replacing every **par** rule construct by **seq**. It is basically a C version of MERGESORT.

5 Conclusion

We have presented an extension of the basic logic for ASMs of [14] that allows to prove also access and read properties of ASMs. One open problem that remains

is to extend the logic to ASMs with the choose-rule and still keep it complete for a large class of ASMs.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool.
2. A. Blass and Y. Gurevich. The linear time hierarchy theorems for Abstract State Machines. *J. Universal Computer Science*, 3(4):247–278, 1997.
3. E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
4. E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
5. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. Technical report, Department of Computing Science, Chalmers University of Technology & Göteborg University, 2003.
6. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at <http://research.microsoft.com/foundations/AsmL/>, 2001.
7. A. Gargantini and E. Riccobene. Encoding abstract state machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, pages 303–322. Springer-Verlag, Lecture Notes in Computer Science 1912, 2000.
8. Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, 2000.
9. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
10. S. Nanchen, H. Schmid, P. Schmitt, and R. F. Stärk. The ASMKY prover. Technical report, Department of Computer Science, ETH Zürich and Institute for Logic, Complexity and Deduction Systems, Universität Karlsruhe, 2003.
11. G. C. Necula. *Compiling with Proofs*. PhD thesis, Computer Science Department, Carnegie Mellon University, CMU-CS-98-154, Pittsburgh PA, 1998.
12. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices*, 33(5):333–344, 1998.
13. G. Schellhorn. *Verification of Abstract State Machines*. PhD thesis, Universität Ulm, 1999.
14. R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. of Universal Computer Science*, 7(11):981–1006, 2001.