A Relational Encoding for a Clash-Free Subset of ASMs

Gerhard Schellhorn, Gidon Ernst, Jörg Pfähler, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg, Germany
{schellhorn,ernst,pfaehler,reif}@isse.de

Abstract. This paper defines a static check for clash-freedom of ASM rules, including sequential and parallel composition, nondeterministic choice, and recursion. The check computes a formula that, if provable, makes a relational encoding of ASM rules possible, which is an important prerequisite for efficient deduction. The check is general enough to cover all sequential rules as well as many typical uses of parallel composition.

Keywords: Abstract State Machines, Synchronous Parallelism, Clashes

1 Introduction

ASM rules are very expressive. Compared to other state-based formalisms they do not just give a transition relation as a formula $\varphi(\underline{x}, \underline{x'})$ in terms of the prestate $\underline{x}$ and the post state $\underline{x'}$ (like e.g. Z, TLA or Event-B do). The additional concepts like function updates, parallel and sequential composition, nondeterministic choice, and defined rules with recursion give ASMs a lot of additional expressiveness that allows refinement from very abstract models down to ASMs which can easily be seen to be equivalent to real programs. For formalisms based on transition relations translating to real programs is hard, typically only the reverse is done: encoding programs to transition relations with the help of program counters. On the flip side a relational encoding for ASM rules is difficult. As a consequence, we are not aware of any deduction approach with tool support for arbitrary ASM rules. Most verification tools, such as e.g. KIV [3], have allowed the purely sequential fragment only with parallel assignments restricted to different function symbols. Others have avoided sequential composition and recursion, and used assignment for functions with arity zero only. With these restrictions however we are in essence back to transition systems.

As soon as parallel rules R are allowed, it becomes hard to define a relation $\mathsf{rel}(R)(\underline{f}, \underline{f'})$, which characterizes the effect of R in terms of the dynamic functions $\underline{f}$ it assigns. Consider the simple parallel rule $f(t_1) := u_1$ **par** $f(t_2) := u_2$. If we define $\mathsf{rel}(f(t_i) := u_i)(f, f') \equiv f'(t_i) = u_i$ and use conjunction for **par** then the relation will not ensure that f is unchanged for arguments other than t_1 and t_2. The formula will also miss the clash for the case $t_1 = t_2$ but $u_1 \neq u_2$, which results in undefined behavior. Clashes are the main obstacle for a relational encoding and in most applications rules with clashes are undesirable anyway.

The contribution of this paper is a predicate $\mathsf{con}(R)$ that statically computes a first-order formula from ASM rules R. If provable, then all executions of rule R are guaranteed to be clash-free[1] and a relational encoding is possible.

The predicate $\mathsf{con}(R)$ is related to the one used in the logic for ASMs defined by Stärk and Nanchen [7] (also given in [2]). While we use the syntax and semantics of ASMs given there, our predicate $\mathsf{con}(R)$ differs in several aspects. It does not use modal constructs ($[R]\ \varphi$) but statically computes a formula even for recursive rules, where the definitions in [7] would lead to an infinite computation (note that the completeness theorem that permits to eliminate modal constructs is for *hierarchical* ASMs only, where recursion is forbidden). Our computation stops at calls and therefore allows one to check each (sub-)rule separately. Different from [7] our $\mathsf{con}(R)$ does not imply that executing R terminates (via $\mathsf{def}(R)$) — termination must be shown using well-founded orders otherwise. We support nondeterministic choice, replaced by choice functions in [7] (making rules and verification conditions at least harder to read).

The new approach in [4] extends [7] to nondeterminism, but does not consider recursion at all. The rules of our relational encoding rel have some similarity to the ones for $\mathrm{upd}(R, X)$ in [4], in particular higher-order functions are used in both. However, our consistency check is purely first-order.

The price we pay for having a computable $\mathsf{con}(R)$ for all rules is that our predicate only approximates clash-freedom. There are clash-free rules which our predicate rejects. The scheme is however strong enough to trivially return true for all rules of the sequential fragment, as well as for some typical parallel rules. In general a theorem prover (or a decision procedure, when the data structures used by the rule are decidable) is needed to prove the computed $\mathsf{con}(R)$, and an SMT solver should suffice for many practical cases to establish clash-freedom.

2 Syntax and Semantics of ASM Rules

We assume the reader to be familiar with first order logic and the syntax of ASM rules R and their semantics as given in e.g. [7]. We only repeat a few basic notations. Given an algebra $\mathfrak{A}$ and a valuation ξ, term t is evaluated to $t_\xi^{\mathfrak{A}}$ and formula φ by $\mathfrak{A}, \xi \models \varphi$. An ASM rule R modifies an algebra $\mathfrak{A}$ to $\mathfrak{A}'$. The basic assignment is $f(\underline{t}) := u$ for a *dynamic* function f. The set of assigned functions in a rule is denoted $mod(R)$. The main rule does not use any free variables, but subrules within **choose** x **with** $\varphi(x)$ **in** R or **forall** x **with** $\varphi(x)$ **in** R may use free variables denoted as $free(R)$. Given an algebra $\mathfrak{A}$ executing a rule R computes $\mathfrak{A}'$ in two steps. First, a set of updates U is computed recursively over the structure of R as $[\![R]\!]_\xi^{\mathfrak{A}} \triangleright U$. An update is of the form $(f, \underline{a}, b)$. Applying it on an algebra $\mathfrak{A}$ modifies function $f^{\mathfrak{A}}$ at arguments $\underline{a}$ to be b. A set of updates U is consistent, if it does not contain two updates $(f, \underline{a}, b_1)$ and $(f, \underline{a}, b_2)$ with $b_1 \neq b_2$.

[1] We regard potential clashes that occur only under some specific non-deterministic choices to be even worse than guaranteed clashes in every run. Even simulating runs of the ASM may fail to detect them. We also regard computing the same update twice as undesirable, and our approach will return $\mathsf{con}(R) = \text{false}$ in both cases.

In this case the whole set U can be applied to give $\mathfrak{A}' \coloneqq \mathfrak{A} \oplus U$. If a rule always computes consistent sets of updates, it is called clash-free. A rule is *defined* if it computes an update set at all as recursive rules may fail to terminate. In the following we assume that all dynamic functions are unary, or have no arguments ("program variables" typically named z). Declarations of a subrule named ρ have the form $\rho(x; z).R$ where variable x is one value parameter and z is a program variable. None of these restrictions is essential, they just allow us to save notation for sequences of arguments. The body R of ρ is restricted to have $mod(R) = z$, all updated locations must be explicitly given. A call of ρ is of the form $\rho(t; f(u))$ where u may contain static function symbols only (to avoid problems with lazy evaluation), semantically $[\![\rho(t; f(u))]\!]^{\mathfrak{A}}_{\xi} \triangleright U$ iff $[\![R^{f(u)}_z]\!]^{\mathfrak{A}}_{\xi\{x \mapsto t^{\mathfrak{A}}_{\xi}\}} \triangleright U$ where $R^{f(u)}_z$ replaces all occurrences of z with $f(u)$, so $f(u)$ is read and updated instead of z.

3 Syntactic Consistency

Syntactic consistency uses the function $\mathsf{asg}(R, f)$ that computes a formula over a dedicated variable f_{arg} and $free(R)$. The values $\xi(f_{\text{arg}})$ for which it hold give an overapproximation of the arguments where f is assigned.

$$\mathsf{asg}(g(u) \coloneqq t, f) \equiv \mathsf{asg}(\rho(t; g(u)), f) \equiv \begin{cases} f_{\text{arg}} = u, & f = g \\ \text{false}, & \text{otherwise} \end{cases}$$

$$\mathsf{asg}(R_1 \textbf{ seq } R_2, f) \equiv \begin{cases} \mathsf{asg}(R_1, f) \vee \mathsf{asg}(R_2, f), & mod(R_1) \cap dep(R_2, f) = \emptyset \\ \text{true}, & f \in mod(R_1 \textbf{ seq } R_2) \\ \text{false}, & \text{otherwise} \end{cases} \qquad (\star)$$

$$\mathsf{asg}(R_1 \textbf{ par } R_2, f) \equiv \mathsf{asg}(R_1, f) \vee \mathsf{asg}(R_2, f)$$

$$\mathsf{asg}(\textbf{if } \varphi \textbf{ then } R_1 \textbf{ else } R_2, f) \equiv (\varphi \wedge \mathsf{asg}(R_1, f)) \vee (\neg\, \varphi \wedge \mathsf{asg}(R_2, f))$$

$$\mathsf{asg}(\textbf{choose } x \textbf{ with } \varphi(x) \textbf{ in } R, f) \equiv \mathsf{asg}(\textbf{forall} \ldots, f) \equiv \exists\, x.\ \varphi(x) \wedge \mathsf{asg}(R, f)$$

For assignments to $f(u)$, we keep $f_{\text{arg}} = u$. The case for sequential composition considers whether R_1 assigns to some g that controls the argument of f as $f(g(u))$ in R_2 ($g \in dep(R_2, f)$). In this case, possible values for f_{arg} are unconstrained if f is modified at all. Conditionals strengthen the check of the branches by the assumption from the test. In a **forall** or **choose** rule, f could be affected by any execution of the body for an x that satisfies the condition φ. Note that we do not impose any constraint on φ, infinitely many choices for x are possible.

The set of dynamic function symbols $dep(R, f)$ that the final value of f after execution of R depends on, can be defined recursively as follows:

$$dep(g(u) \coloneqq t, f) = dep(\rho(t; g(u)), f) \coloneqq \begin{cases} \{h : h \text{ occurs in } t \text{ or in } u\}, & f = g \\ \emptyset, & \text{otherwise} \end{cases}$$

$$dep(R_1 \textbf{ seq } R_2, f) \coloneqq dep(R_1, f) \cup dep(R_2, f) \cup \textstyle\bigcup_{g \in dep(R_2, f)} dep(R_1, g)$$

$$dep(R_1 \textbf{ par } R_2, f) \coloneqq dep(R_1, f) \cup dep(R_2, f)$$

$$dep(\textbf{if } \varphi \textbf{ then } R_1 \textbf{ else } R_2, f) \coloneqq \{h : h \text{ occurs in } \varphi\} \cup dep(R_1, f) \cup dep(R_2, f)$$

$$dep(\textbf{choose } x \textbf{ with } \varphi(x) \textbf{ in } R, f)$$
$$= dep(\textbf{forall } x \textbf{ with } \varphi(x) \textbf{ in } R, f) \coloneqq \{h : h \text{ occurs in } \varphi(x)\} \cup dep(R, f)$$

For assignments, dependencies come from the argument terms and the right hand sides. Sequential composition chains the dependencies transitively. For **if**, **choose**, and **forall**, the dynamic functions h occurring in the respective test φ potentially have an influence on the final value of f as well.

Syntactic consistency $\mathsf{con}(R)$ of a rule R is defined over the structure of rules:

$$\mathsf{con}(g(u) \coloneqq t) \equiv \mathsf{con}(\rho(t; f(u))) \equiv \text{true}$$

$$\mathsf{con}(R_1 \textbf{ seq } R_2) \equiv \mathsf{con}(R_1) \wedge \mathsf{con}\Big(R_2{}^{\underline{f'}}_{mod(R_1)}\Big) \quad \text{where the } \underline{f'} \text{ are globally fresh}$$

$$\mathsf{con}(R_1 \textbf{ par } R_2) \equiv \mathsf{con}(R_1) \wedge \mathsf{con}(R_2) \wedge \textstyle\bigwedge_f \neg\, \exists\, y.\ \mathsf{asg}(R_1, f)(y) \wedge \mathsf{asg}(R_2, f)(y)$$

$$\mathsf{con}(\textbf{if } \varphi \textbf{ then } R_1 \textbf{ else } R_2) \equiv (\varphi \wedge \mathsf{con}(R_1)) \vee (\neg\, \varphi \wedge \mathsf{con}(R_2))$$

$$\mathsf{con}(\textbf{choose } x \textbf{ with } \varphi(x) \textbf{ in } R) \equiv \forall\, x.\ \varphi(x) \rightarrow \mathsf{con}(R)$$

$$\mathsf{con}(\textbf{forall } x \textbf{ with } \varphi(x) \textbf{ in } R) \equiv \forall\, x.\ (\varphi(x) \rightarrow \mathsf{con}(R)) \wedge$$
$$\textstyle\bigwedge_f \neg\, \exists\, x_1, x_2, y.\ x_1 \neq x_2 \wedge \varphi(x_1) \wedge \mathsf{asg}(R^{x_1}_x, f)(y) \wedge \varphi(x_2) \wedge \mathsf{asg}(R^{x_2}_x, f)(y)$$

Assignments and calls do not impose any additional constraints. as they do not provoke clashes when viewed in isolation (provided that the body of procedures ρ is checked separately). In a sequential composition, consistency of R_2 must be checked for possibly modified values of dynamic functions, expressed by fresh symbols $\underline{f'}$ that are unconstrained. An example, where we lose precision is

$$R^* \quad \equiv \quad g(u_1) \coloneqq u_2 \textbf{ seq } f(g(u_3)) \coloneqq u_4, \qquad \text{where the } u_i \text{ are static terms.}$$

For the second assignment, we get $dep(f(g(u_3)) \coloneqq u_4, f) = \{g\}$. Therefore case $(\star)$ applies and $\mathsf{asg}(R^*, f) = \text{true}$. Informally, we do not know statically whether the first assignment affects the argument of f, i.e., whether $g(u_1)$ aliases $g(u_3)$. Note that R^* is still clash-free ($\mathsf{con}(R^*) = \text{true}$).

Parallel execution of R_1 and R_2 conservatively excludes assignments to the same location, where $\mathsf{asg}(R, f)(y)$ renames f_{arg} to a fresh variable y in $\mathsf{asg}(R, f)$. To continue the example, putting R^* in parallel with any assignment to f will make con false for the combined rule. Note that this combination of sequential and parallel composition, and the fact that we assume that the argument $f(u)$ is *always* assigned in a recursive call are the only two sources for imprecision (when assigning the same value to a location twice is regarderd as a clash).

Nondeterministic choice hides the bound variable x and adds the assumption $\varphi(x)$ about the choice for that x to the consistency check of the body. For **forall** we additionally exclude conflicts between two pairwise parallel executions of the body where two fresh distinct representants x_1 and x_2 of the index x both cause assignments to $f(y)$ for the same y that replaces f_{arg} locally in the body.

Lemma 1. *Given that R yields update set U, i.e., $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$:*
- *If $f \notin mod(R)$ then $(f, a, b) \notin U$ for all a, b.*
- *If $g^{\mathfrak{A}} = g^{\mathfrak{A} \oplus U}$ for all $g \in dep(R, f)$ then $\mathfrak{A}, \xi \models \mathsf{asg}(R, f)$ iff $\mathfrak{A} \oplus U, \xi \models \mathsf{asg}(R, f)$.*
- *If $(f, a, b) \in U$ for some a, b then $\mathfrak{A}, \xi\{f_{\text{arg}} \mapsto a\} \models \mathsf{asg}(R, f)$.*
- *If $\mathfrak{A}, \xi \models \mathsf{con}(R)$ then U is consistent.*

The lemma states that *mod*, *dep*, asg, resp. con are correct. The second bullet lifts (not) asg over the first rule in a sequential composition with updates U.

4 Relational Encoding

The relational encoding $\mathsf{rel}(R)(\underline{f}, \underline{f}')$ is a predicate over $\underline{f} \equiv \mathit{mod}(R)$ and primed versions $\underline{f}'$ of these functions. Its free variables are at most the free variables of R. Variable y as well as function variables f_1, f_2, F used below are fresh. We abbreviate $(\varphi \to t = u) \wedge (\neg\, \varphi \to t = v)$ with $t = (\varphi \supset u; v)$.

$$\mathsf{rel}(g(t) := u)(\underline{f}, \underline{f}') \equiv \forall\, y.\ g'(y) = (y = t \supset u; g(y)) \wedge \textstyle\bigwedge_{f \in \underline{f}, f \neq g} f' = f$$

$$\mathsf{rel}(\rho(t; g(u)))(\underline{f}, \underline{f}') \equiv \exists\, y.\ y = t \wedge \mathsf{rel}(R^{g(u),y}_{z,x})(g, g') \wedge \textstyle\bigwedge_{f \in \underline{f}, f \neq g} f' = f$$

$$\mathsf{rel}(R_1 \ \mathbf{seq}\ R_2)(\underline{f}, \underline{f}') \equiv \exists\, \underline{f}_1.\ \mathsf{rel}(R_1)(\underline{f}, \underline{f}_1) \wedge \mathsf{rel}(R_2)(\underline{f}_1, \underline{f}')$$

$$\mathsf{rel}(R_1 \ \mathbf{par}\ R_2)(\underline{f}, \underline{f}') \equiv \exists\, \underline{f}_1, \underline{f}_2.\ \mathsf{rel}(R_1)(\underline{f}, \underline{f}_1) \wedge \mathsf{rel}(R_2)(\underline{f}, \underline{f}_2) \wedge \mathit{merge}(\underline{f}_1, \underline{f}_2, \underline{f}')$$

where $\mathit{merge}(\underline{f}_1, \underline{f}_2, \underline{f}') := \bigwedge_{f, f_1, f_2 \in \underline{f}, \underline{f}_1, \underline{f}_2} \forall\, y.\ f'(y) = (\mathsf{asg}(R_1, f)(y) \supset f_1(y); f_2(y))$

$$\mathsf{rel}(\mathbf{if}\ \varphi\ \mathbf{then}\ R_1\ \mathbf{else}\ R_2)(\underline{f}, \underline{f}') \equiv (\varphi \supset \mathsf{rel}(R_1)(\underline{f}, \underline{f}'); \mathsf{rel}(R_2)(\underline{f}, \underline{f}'))$$

$$\mathsf{rel}(\mathbf{choose}\ x\ \mathbf{with}\ \varphi(x)\ \mathbf{in}\ R)(\underline{f}, \underline{f}') \equiv \exists\, x.\ \varphi(x) \wedge \mathsf{rel}(R)(\underline{f}, \underline{f}')$$

$$\begin{aligned}\mathsf{rel}(\mathbf{forall}\ x\ \mathbf{with}\ \varphi(x)\ \mathbf{in}\ R)(\underline{f}, \underline{f}') \equiv\ & \exists\, \underline{F}.\ (\forall\, x.\ \varphi(x) \to \mathsf{rel}(R)(\underline{f}, \underline{F_x})) \wedge \\ & \textstyle\bigwedge_{f, F \in \underline{f}, \underline{F}} \forall\, y. \quad \big(\forall\, x.\ \varphi(x) \wedge \mathsf{asg}(R, f)(y) \to f'(y) = F_x(y)\big) \\ & \qquad \wedge \big((\forall\, x.\ \varphi(x) \to \neg\, \mathsf{asg}(R, f)(y)) \to f'(y) = f(y)\big)\end{aligned}$$

The rule for call renames variable x to y to avoid a conflict when x occurs in t. The definition solves the problem of parallel rules R_1 **par** R_2 from the introduction by first computing two individual results $\underline{f}_1$ and $\underline{f}_2$ for the two rules. Since we know from the fact that $\mathsf{con}(R)$ holds that each location (f, y) is assigned by at most one of the rules (so at most one of the predicates $\mathsf{asg}(R_i, f)(y)$ holds), the definition chooses the first rule if it assigned the location and otherwise the second one. The definition for **forall** generalizes from two results $\underline{f}_1, \underline{f}_2$ to a result F_x for every x satisfying φ. Note that since $x \in \mathit{free}(R)$ the result F_x may be different for every argument. In the presence of a clash, the formula is equivalent to false.

If we interpret $\mathfrak{A} \cup \mathfrak{A}'$ as the algebra that evaluates every unprimed function f over $\mathfrak{A}$ and every primed function f' over $\mathfrak{A}'$, we have:

Theorem 1. *Given a rule R with $\mathit{mod}(R) = \underline{f}$ and $\mathfrak{A}, \xi \models \mathsf{con}(R)$, then $\mathfrak{A} \cup \mathfrak{A}' \models \mathsf{rel}(R)(\underline{f}, \underline{f}')$ if and only if there is some consistent U such that $[\![R]\!]^{\xi}_{\mathfrak{A}} \triangleright U$ and $\mathfrak{A}' = \mathfrak{A} \oplus U$.*

5 Conclusion & Outlook

We have defined a clash-freedom check for ASM rules. All sequential rules check trivially. Typical parallel rules with disjoint tests (e.g. used in the WAM [1])

$$\mathbf{if}\ \mathit{instruction} = i_1\ \mathbf{then}\ R_1 \quad \mathbf{par} \quad \mathbf{if}\ \mathit{instruction} = i_2\ \mathbf{then}\ R_2$$

are also allowed. Lifting a rule $R(; lv)$ of one process p with process-local state lv to a parallel rule **forall** p **do** $R(; lv(p))$ for all processes p works, too (e.g. used for the threads of the Java ASM [8]).

We have verified the results in KIV by a predicate logic embedding of ASM rules (except for calls) and their semantics (see the URL [9]), similar to what we have done for the temporal logic RGITL [6] (including calls). This uncovered several mistakes in initial versions of the definitions.

The check presented in this paper could be improved in practice by using invariants of the ASM or preconditions of recursive rules as assumptions, e.g., the rule $f(x) \coloneqq 1$ **par** $f(y) \coloneqq 2$ is clash-free when the invariants imply $x \neq y$.

In parallel to our work, a relational encoding of ASMs to Event-B was developed in [5]. In contrast to ours, the clash-freedom check is exact and tolerates rules, which compute the same update several times. The approach has been evaluated with several examples, while we have only tried mimimal ones. The approach avoids the use of higher-order functions using set theory instead. On the other hand the approach is limited to ASM rules without recursion and sequential composition, so it is not sufficient to support the rules used in KIV.

This work is only the first step towards interactive proofs with a larger set of rules in KIV. Such deduction needs rules for symbolic execution, as the result of simply substituting the relation for the rule would be incomprehensible. For a clash-free rule R_1 **par** R_2 our result shows that it is valid to transform the rule to $\{\underline{f_1}, \underline{f_2} \coloneqq \underline{f}, \underline{f}\}$ **seq** $R_1\frac{f_1}{f}$ **seq** $R_2\frac{f_2}{f}$ and compute the final result with *merge* from the definition of $\mathsf{rel}(R_1$ **par** $R_2)$. When **forall** iterates over a finite set, inductive arguments over its size should be possible.

References

1. E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence 11, pages 20–90. Elsevier, 1995.
2. E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer, 2003.
3. G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV - Overview and VerifyThis Competition. *Software Tools for Techn. Transfer*, 17(6):677–694, 2015.
4. F. Ferrarotti, K.-D. Schewe, L. Tec, and Q. Wang. A logic for non-deterministic parallel Abstract State Machines. In *Proc. of FoIKS*, Springer LNCS 9616, pages 334–354, 2016.
5. M. Leuschel and E. Börger. A compact ecoding of sequential ASMs in Event-B. In *Proc. ABZ of 2016 (this volume)*. Springer LNCS, 2016.
6. G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, and W. Reif. RGITL: A temporal logic framework for compositional reasoning about interleaved programs. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 71:131–174, 2014.
7. R. F. Stärk and S. Nanchen. A complete logic for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 7(11):981–1006, 2001.
8. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
9. A relational encoding for a clash-free subset of ASMs: Formalization and proofs. `https://swt.informatik.uni-augsburg.de/swt/projects/Refinement/ASM-clashfreedom.html`.