Symbolic Execution for a Clash-Free Subset of ASMs ⋆

Gerhard Schellhorn[a], Gidon Ernst[a], Jörg Pfähler[a], Stefan Bodenmüller[a], Wolfgang Reif[a]

[a] *Institute for Software & Systems Engineering*
University of Augsburg, Germany

Abstract

Providing efficient theorem proving support for general ASM rules that update proper functions, use sequential and parallel composition, nondeterministic choice and recursion is difficult, since it is not easy to find a predicate logic formula that describes the transition relation of an ASM rule. One important obstacle to achieving this goal is that executing rules may result in a clash, that aborts the ASM run. This paper contributes three results towards this goal.

First, it shows that it is possible to compute a first-order formula for each rule that implies clash-freedom when provable. The derived formula is not a precise characterization, but is provable for many ASMs that are used in practice.

Second, we give axioms that describe the transition relation for clash-free ASM rules as formulas of predicate logic that can be used to verify pre/post-condition assertions using automated theorem provers.

Third, we show that the relational encoding can be used to justify a calculus for clash-free ASM rules based on symbolic execution. Such a calculus is useful for interactive theorem provers such as our tool KIV.

Keywords: Abstract State Machines, Symbolic Execution, 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 pre state $\underline{x}$ and the post state $\underline{x'}$ (like 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. They allow refinement from very abstract models down to ASMs

Email addresses: schellhorn@isse.de (Gerhard Schellhorn), ernst@isse.de (Gidon Ernst), pfaehler@isse.de (Jörg Pfähler), bodenmueller@isse.de (Stefan Bodenmüller), reif@isse.de (Wolfgang Reif)

⋆ This work is partly sponsored by the Deutsche Forschungsgemeinschaft (DFG), grant number RE828/13-1 ("Verifikation von Flash-Dateisystemen").

which can easily be seen to be equivalent to real programs which is the focus of our work, see e.g. [1]. For formalisms based on transition relations translating to real programs is hard, typically only the reverse is done. Using program counters a transition relation can be derived from a program.

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, since an explicit transition relation is the basis for model checking as well as for abstracting calls of auxiliary rules using contracts. Most verification tools, such as KIV [2], have allowed the purely sequential fragment without any parallel rules or allowed only parallel assignments 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 is possible that the rule R produces clashes. A *clash* occurs if R produces two different, parallel updates for the same location. With clashes it becomes hard to define a transition relation $\mathsf{rel}(R)(\underline{f}, \underline{f'}, \underline{x})$ which characterizes the effect of R in terms of the dynamic functions $\underline{f}$ it assigns and the variables $\underline{x}$ it reads. Consider the simple parallel rule $f(t_1) := u_1$ **par** $f(t_2) := u_2$. If we define $\mathsf{rel}(f(t_i) := u_i)(f, f', \underline{x}) \equiv f'(t_i) = u_i$ and use conjunction for **par**, the relation will not ensure that f is unchanged for arguments other than t_1 and t_2. The formula will also mask 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. However, in most applications *clash-free* rules which avoid such conflicting updates are desirable.

This paper contributes a clash-freedom check $\mathsf{cfc}(R)$ that statically computes a first-order formula for an ASM rule R. If provable, all executions of rule R are guaranteed to be clash-free in the strongest possible sense, i.e., its evaluation will never result in updates to the same location, not even with the same value. We discuss weaker notions in related work.

Our emphasis is on a modular definition. Mutually recursive rules $R_1, \ldots R_n$ give formulas $\mathsf{cfc}(R_i)$ for each R_i that can be verified independently. Such mutually recursive rules are critical for a refinement based approach that seeks to develop efficient software. Top-level specifications have very simple and abstract rules, but the final implementation typically involves iteration and recursion. The price we pay is that the predicate is necessarily an overapproximation of clash-freedom, since $\mathsf{cfc}(R_i)$ cannot depend on the semantics of other called rules R_j, but on their call-interface only. We add reference parameters to make (potentially) assigned locations explicit in calls and to facilitate syntactic substitution of dynamic functions.

We then contribute a relational encoding and a symbolic execution calculus for deduction for clash-free ASM rules. This paper extends the short paper [3] with an improved dependency analysis (Sec. 4), and a calculus for deduction (Sec. 7). The relational encoding (Sec. 6) is now proven to be sound for all clash-free rules, not just for those where checking $\mathsf{cfc}(R)$ is successful. We have also added more examples throughout.

The paper is structured as follows. Sec. 2 recaps the syntax and seman-

tics of ASMs. Sec. 3 states the goals behind the various predicates and their interaction. Sec. 4 gives a predicate for dependency analysis and one for the assigned locations. The clash-freedom check is presented in Sec. 5 and builds on these predicates. Sec. 6 and Sec. 7 give a relational encoding of and a calculus for ASMs based on symbolic execution, under the assumption of clash-freedom. Sec. 8 summarizes related work and Sec. 9 concludes.

2. Syntax and Semantics of ASM Rules

We assume the reader to be familiar with first order logic and repeat only some basic notations. A signature Σ consists of function symbols f and predicate symbols p, all equipped with some arity. Given a signature and a set of variables $x, y, z \in X$, terms t, u and formulas φ can be built up as usual. Terms and formulas are evaluated over a (Σ-)Algebra $\mathfrak{A}$ (with carrier set $|\mathfrak{A}|$ and functions $f^{\mathfrak{A}}$) and a valuation ξ as $t^{\mathfrak{A}}_{\xi}$ and by $\mathfrak{A}, \xi \models \varphi$. The set of variables in a term t or a formula φ are denoted $\mathit{vars}(t)$ and $\mathit{vars}(\varphi)$, the free variables of φ as $\mathit{free}(\varphi)$. In the following tuples are written underlined, i.e., $\underline{t} = t_1, \ldots, t_n$ is a tuple of terms, and we assume functions to be extended to tuples in the natural way, i.e., $\underline{t}^{\mathfrak{A}}_{\xi}$ is a tuple of semantic values, $\mathit{vars}(\underline{t})$ are all variables occurring in any t_i.

For the relational encoding we will also use second order formulas that additionally allow function variables $\hat{f}$. These are used like function symbols in terms. Like variables they can be quantified and $\xi(\hat{f})$ returns a function.

2.1. Syntax

For use in ASMs, the first order signature is assumed to be partitioned into a static signature, which is axiomatized using algebraic specifications, and a dynamic signature, which is allowed to be modified by the rules.

Given a finite, possibly empty set of rule identifiers P with typical element ρ, an ASM rule R follows the grammar

$$\begin{aligned} R ::= \; & \textbf{skip} \;\mid\; f(\underline{t}) := u \;\mid\; R_1 \textbf{ par } R_2 \;\mid R_1 \textbf{ seq } R_2 \;\mid \\ & \textbf{if } \varphi \textbf{ then } R_1 \textbf{ else } R_2 \;\mid\; \textbf{choose } x \textbf{ with } \varphi \textbf{ in } R \;\mid \\ & \textbf{forall } x \textbf{ do } R \;\mid\; \rho(\underline{t}; \underline{h(u)}) \end{aligned}$$

Rule **skip** does nothing, the update $f(\underline{t}) := u$ modifies the dynamic function f to be u at arguments $\underline{t}$. Constructs **par** and **seq** are parallel and sequential composition. The **if** executes R_1 if φ holds and R_2 otherwise. The **choose** nondeterministically binds some element that satisfies φ to local variable x and executes R with this binding. The **forall** construct executes R for all possible values of x in parallel. A call has by-name parameters $\underline{t}$ and reference parameters $\underline{h(u)} \equiv h_1(\underline{u}_1), \ldots, h_n(\underline{u}_n)$, where h_i are pairwise different dynamic functions. The terms $\underline{t}$ as well as $\underline{u}_i$ are required to be static, i.e. to not not contain any dynamic functions. It is allowed to pass an entire dynamic function h_i with $u_i \equiv \langle\rangle$, where $\langle\rangle$ stands for an empty list of arguments. This allows the callee to assign arbitrary locations of h_i.

$$\frac{}{[\![f(\underline{t}) := u]\!]^{\mathfrak{A}}_{\xi} \triangleright \{(f, \underline{t}^{\mathfrak{A}}_{\xi}, u^{\mathfrak{A}}_{\xi})\}} \quad \text{(sem-asg)}$$

$$\frac{[\![R_1]\!]^{\mathfrak{A}}_{\xi} \triangleright U_1 \quad [\![R_2]\!]^{\mathfrak{A}}_{\xi} \triangleright U_2}{[\![R_1 \textbf{ par } R_2]\!]^{\mathfrak{A}}_{\xi} \triangleright U_1 \cup U_2} \quad \text{(sem-par)}$$

$$\frac{[\![R_1]\!]^{\mathfrak{A}}_{\xi} \triangleright U_1 \quad \mathsf{con}(U_1) \quad [\![R_2]\!]^{\mathfrak{A}+U_1}_{\xi} \triangleright U_2}{[\![R_1 \textbf{ seq } R_2]\!]^{\mathfrak{A}}_{\xi} \triangleright U_1 \oplus U_2} \quad \text{(sem-seq-cons)}$$

$$\frac{[\![R_1]\!]^{\mathfrak{A}}_{\xi} \triangleright U_1 \quad \neg\, \mathsf{con}(U_1)}{[\![R_1 \textbf{ seq } R_2]\!]^{\mathfrak{A}}_{\xi} \triangleright U_1} \quad \text{(sem-seq-incons)}$$

$$\frac{[\![R_1]\!]^{\mathfrak{A}}_{\xi} \triangleright U \quad \mathfrak{A}, \xi \models \varphi}{[\![\textbf{if } \varphi \textbf{ then } R_1 \textbf{ else } R_2]\!]^{\mathfrak{A}}_{\xi} \triangleright U} \quad \text{(sem-if-pos)}$$

$$\frac{[\![R_2]\!]^{\mathfrak{A}}_{\xi} \triangleright U \quad \mathfrak{A}, \xi \not\models \varphi}{[\![\textbf{if } \varphi \textbf{ then } R_1 \textbf{ else } R_2]\!]^{\mathfrak{A}}_{\xi} \triangleright U} \quad \text{(sem-if-neg)}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\xi\{x \mapsto a\}} \triangleright U \quad \mathfrak{A}, \xi\{x \mapsto a\} \models \varphi}{[\![\textbf{choose } x \textbf{ with } \varphi \textbf{ in } R]\!]^{\mathfrak{A}}_{\xi} \triangleright U} \quad \text{(sem-choose)}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\xi\{x \mapsto a\}} \triangleright U_a \quad \text{for all } a \in |\mathfrak{A}|}{[\![\textbf{forall } x \textbf{ do } R]\!]^{\mathfrak{A}}_{\xi} \triangleright \bigcup_a U_a} \quad \text{(sem-forall)}$$

$$\frac{[\![R^{\underline{t}}_{\underline{y}\ \underline{g}}]\!]^{\mathfrak{A}}_{\xi} \triangleright U}{[\![\rho(\underline{t}; \underline{h(u)})]\!]^{\mathfrak{A}}_{\xi} \triangleright U} \text{ declaration } \rho(\underline{y}; \underline{g}).R \quad \text{(sem-call)}$$

Figure 1: Derivation rules for update sets

The dynamic functions used in R and transitively in all rules called by R are denoted $\mathsf{dyn}(R)$. The subset of those functions f modified in updates $f(\underline{t}) := u$ of R or in any of its subrules are denoted $\mathsf{mod}(R)$. The free variables $\mathit{free}(R)$ of a rule are all variables used without the bound occurrences of x in **choose** x and **forall** x.

In an ASM all rules should have a proper declaration.

Definition 1 (Proper Declaration)**.** *A* proper *declaration $\rho(\underline{y}, \underline{g}) = R$ for a rule identifier ρ has pairwise different variables $\underline{y}$ as formal call-by-name parameters and pairwise different dynamic functions $\underline{g}$ as formal reference parameters. The* body *R of ρ is required to satisfy $\mathsf{dyn}(R) = \underline{g}$ and $\mathit{free}(R) = \underline{y}$.*

Definition 2 (Call Fits to Declaration)**.** *A call $\rho(\underline{t}; \underline{h(u)})$* fits *to a declaration $\rho(\underline{y}, \underline{g}) = R$ if the numbers of parameters agree. A parameter h_i with $u_i \neq \langle\rangle$ is allowed when g_i is a dynamic function without arguments, otherwise the arity of g_i and h_i must be the same.*

A call will execute the body R where $\underline{y}$ has been substituted with $\underline{t}$ and all g_i have been substituted with $h_i(\underline{u_i})$ (resp. with h_i if $u_i = \langle\rangle$). The substituted body is written $R^{\underline{t}\ \underline{h(u)}}_{\underline{y}\ \underline{g}}$. Substitution has to rename variables x bound by **choose** x or **forall** x when they conflict with variables used in $\underline{t}$ or $\underline{u}$.

Definition 3 (ASM)**.** *An ASM consists of proper declarations for all rule identifiers P, where all calls fit to the declarations. An ASM has one designated main rule, which does not have any call-by-name parameters $\underline{y}$ (so the rule has no free variables), and has all $\underline{u_i} = \langle\rangle$.*

Note that Def. 3 allows for mutually recursive calls. The toplevel rule has no "real" parameters, it reads and/or modifies some dynamic functions $\underline{g}$ from the signature.

We have defined the core syntax as minimal as possible to save unnecessary cases in definitions and proofs. To get the full syntax of [4], the following abbreviations for the standard **let**-binding and conditional **forall** suffice.

let $x = t$ **in** R	$\equiv$	**choose** x_0 **with** $x_0 = t$ **in** $R^{x_0}_x$
if φ **then** R	$\equiv$	**if** φ **then** R **else skip**
forall x **with** φ **do** R	$\equiv$	**forall** x **do if** φ **then** R

The abbreviation for **let** renames all occurrences of x to a fresh variable x_0 in R. Freshness is required to avoid a naming conflict when x is free in t since the scope of x in **let** excludes the term t while the scope of x_0 in **choose** includes the equation $x_0 = t$.

Our calling convention differs in the following aspects from the standard convention of ASMs as given in [5].

- We restrict call-by-name parameters to be static terms, which effectively makes them call-by-value. Otherwise it is not possible to determine a uniform characterization of clash-freedom (see Example 1 below).

- We require the dynamic functions used in ASM rules to be explicitly mentioned as reference-parameters in calls and declarations, while usually they are considered as globally available. To translate an ordinary ASM to our setting one has to compute the used functions $\mathsf{dyn}(R)$ of a rule as the transitive closure of the ones directly used by R together with the ones used in bodies of called rules. Both formal and actual reference-parameters $\underline{g}$ and $\underline{h(u)}$ then have to be set to $\mathsf{dyn}(R)$ and substitution of $\underline{g}$ by $\underline{h}$ in R then has no effect. Having the used functions explicit in rule declarations is not required for most results of this paper. It is however mandatory for defining a calculus based on symbolic execution, see Sec. 7.

- We add the possibility of using reference parameters $h_i(\underline{u_i})$ with $\underline{u_i} \neq \langle\rangle$. These increase the precision of our clash-freedom check since they ensure that all assignments to h_i will be specifically with arguments $\underline{u_i}$. Such parameters are also useful when parameterizing ASMs with processes. A typical example is when lifting a rule $R(;\underline{g})$ that uses registers $\underline{g}$ of a single thread to a rule parameterized with a thread t as is done in the Java ASM [6]. With the extension defined here this can be done by just using a call $R(;\underline{g}(t))$ in the main rule and by assuming that the ASM has registers $\underline{g}(t)$ for every thread t. Ambient ASMs [7] offer a concept of environments that goes beyond the simple extension we use here.

Definition 4 (Sequential Fragment)**.** *An ASM rule is in the sequential fragment if it does neither use the* ***par*** *nor the* ***forall*** *construct. An ASM is in the sequential fragment, if all declared rules are.*

2.2. Semantics

Given an algebra $\mathfrak{A}$ executing a rule R computes a new algebra $\mathfrak{A}'$ in two steps. First, a set U of updates is computed (nondeterministically) as the least fixpoint of the rules given in Fig. 1. In other words, a closed derivation tree with conclusion $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$ must exist. Each update in U is of the form $(f, \underline{a}, b)$, where f is a dynamic function symbol and $\underline{a}, b$ are elements of the carrier set of the algebra. An *f-update* is an update with f as function symbol and we call $f(\underline{a})$ the updated *location*.

Definition 5 (Consistency of Update Sets)**.** *A set of updates U is consistent, written $\mathsf{con}(U)$, if it does not contain two updates for the same location $f(\underline{a})$, i.e., there are no $(f, \underline{a}, b_1)$ and $(f, \underline{a}, b_2)$ in U with $b_1 \neq b_2$.*

If consistent, the set U can be applied in the second step to give the new algebra as $\mathfrak{A}' := \mathfrak{A} + U$. The algebra $\mathfrak{A} + U$ leaves the carrier set of $\mathfrak{A}$ unchanged but modifies each $f^{\mathfrak{A}}$ at any argument $\underline{a}$ to be b where $(f, \underline{a}, b) \in U$.

Definition 6 (Clash-Freedom)**.** *A rule R is called clash-free, written $\mathsf{con}(R)$, if it never computes an inconsistent set of updates, i.e., $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$ for any $\mathfrak{A}$ and ξ and U implies $\mathsf{con}(U)$.*

The set $U_1 \oplus U_2$ computed by the rule *sem-seq-cons* contains all updates of U_2 and all updates $(f, \underline{a}, b)$ from U_1 for which no update $(f, \underline{a}, c)$ is in U_2. The definition is such that $\mathfrak{A} + (U_1 \oplus U_2) = (\mathfrak{A} + U_1) + U_2$ holds, i.e., the effect is that of sequentially applying the update sets. The rule *sem-seq-incons* makes sure that R_1 **seq** R_2 produces an inconsistent update set, if R_1 does.

The computation of the update set for **choose** (*sem-choose*) is nondeterministic. Possible update sets may be computed by binding x to any value a that satisfies φ in its premise.

The union in rule *sem-forall* is over all elements a of the carrier set $|\mathfrak{A}|$ of $\mathfrak{A}$, so the rule usually has infinitely many premises. A closed derivation tree using *sem-forall* therefore has infinite width. The least fixpoint of the rule system still exists by Knaster-Tarski's Theorem [8]. Restricted versions of *sem-forall*, which allow only finite choice, and therefore admit a simpler characterization of the least fixpoint by Kleene's theorem, are possible, but we do not need such a restriction in this work, since rule induction (induction, assuming the property for all premises to prove it for the conclusion) is still admissible and sufficient for all proofs. Many proofs only require induction over the structure of a rule.

Clashes are only produced by the *sem-forall* and *sem-par* rules. For example the rule $f(0) \coloneqq 1$ **par** $f(0) \coloneqq 2$ contains a clash since $f(0) \coloneqq 1$ and $f(0) \coloneqq 2$ yield the update set $\{(f, 0, 1)\}$ and $\{(f, 0, 2)\}$, respectively. Combining those two update sets as done by the *sem-par* rule produces an inconsistent update set and therefore the rule has a clash.

Rule *sem-call* assumes that the declaration of ρ is again $\rho(\underline{y}; \underline{g}).R$. As described before, calling R replaces formal with actual parameters, denoted $R^{\underline{t}}_{\underline{y}\,\underline{g}}{}^{\underline{h(u)}}$.

In general a rule may compute infinitely many different update sets, but it may also fail to compute an update set at all, when the computation goes into an infinite recursion (even nondeterministically) or when there is no choice in rule *sem-choose* (when $\varphi = \mathit{false}$). We do not consider such a rule to have a clash in this paper, even though possible nontermination when computing an update set can be viewed as a similarly erroneous behavior than producing an inconsistent one. The reason is simply that proving guaranteed termination requires well-foundedness arguments, which are quite different from checks for clash-freedom.

We remark that a definition of guaranteed termination is of course possible, based on another set of rules similar to the ones given e.g. in [9] for guaranteed termination of while programs. This would rule out the possibility of infinite recursion and would allow to define a weakest precondition operator. A full definition is however outside the scope of this paper.

Lemma 1. *Given that R yields update set U, i.e., $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$, and $f \notin \mathsf{mod}(R)$ then $(f, \underline{a}, b) \notin U$ for all $\underline{a}$, b.*

Proof. By rule induction over R. □

The restriction to allow only static terms $\underline{u}$ and $\underline{t}$ in calls $\rho(\underline{t}; \underline{h(u)})$ (Def. 3) is motivated by our intent, to analyze subrules modularly, with a separate clash-

free analysis of the body. To do this we must ensure that a reference parameter $h(u)$ forces that h can be assigned at u only. Then Theorem 2 holds, which will be used to justify a modular analysis. Otherwise, the Examples 1 and 2 show that the theorem does not hold.

Lemma 2 (Substitution in Calls)**.** *Let $\rho(\underline{t}; h(u))$ be a call in an ASM to a rule declared as $\rho(\underline{y}; \underline{g})\{R_0\}$ where $h_i = g_i$ for all i with $\underline{u}_i \neq \langle\rangle$. Let R be any subrule of the body R_0. Then $[\![R^{\underline{t}}_{\underline{y}\,\underline{h(u)}}]\!]^{\mathfrak{A}}_{\xi} \triangleright U$ implies that all updates to any h_i with $\underline{u}_i \neq \langle\rangle$ are of the form $(h_i, \underline{u}_i{}^{\mathfrak{A}}_{\xi}, b)$. Derivability of U is equivalent to $[\![R]\!]^{\mathfrak{A}'}_{\xi\{\underline{x}\mapsto\underline{t}\}} \triangleright U'$, where U' replaces such updates in U with (g_i, b) and where $\mathfrak{A}'$ adds $\underline{h(u)}^{\mathfrak{A}}_{\xi}$ as the interpretation of the formal parameters $\underline{g}$.*

Proof. By rule induction. The crucial point is that static terms $\underline{u}_i$ and $\underline{t}$ will always evaluate to the same value. In particular, an assignment $h(u_i) := t_j$ that results from syntactic substitution in $g_i := x_j$ will always produce the update $(h_i, u_i{}^{\mathfrak{A}}_{\xi}, t_j{}^{\mathfrak{A}}_{\xi})$, even if it is evaluated in an algebra $\mathfrak{A} + U$ instead of $\mathfrak{A}$. □

The Examples 1 and 2 show that Lemma 2 is wrong, when non-static $\underline{u}$ and $\underline{t}$ are used in calls.

Example 1. Consider a declaration $\rho(; g, c) = \{\, c := c + 1 \;\mathbf{seq}\; g := 2 \,\}$. A call $\rho(; h(c), c)$ which uses non-static reference parameter $h(c)$ assigns h at $c+1$, instead of $h(c)$. □

Example 2. Consider the declaration

$$\rho(x; g, f) = \{\, g := 1 \;\mathbf{seq}\; \{\, f(x) := 1 \;\mathbf{par}\; f(1) := 2 \,\}\,\}.$$

Then $\mathsf{con}(\rho(x; g, f)) \equiv x \neq 1$, but if we pass a non-static parameter g we have $\mathsf{con}(\rho(g; g, f)) \equiv$ false. The formula $\mathsf{con}(\rho(x; g, f))$ then is not $g \neq 1$, the result of substituting x with g in $x \neq 1$. □

The lemma implies that properties of calls like $\rho(\underline{t}; \underline{h(u)})$ can be reduced to a property of its body R. As an example $\mathsf{con}(\rho(\underline{t}; \underline{h(u)}))$ then is the same as $\mathsf{con}(R)^{\underline{t},\underline{h(u)}}_{\underline{y},\underline{g}}$. The same will hold for the relational encoding. Without the restriction, the analysis of calls (clash-freedom as well as determining a relational encoding) must be done individually for each call the ASM will do while executing the main rule, which may be an unbounded number when recursion is present.

In the following sections we assume that all dynamic functions used in rule declarations are nullary or unary. This restriction is not essential, it just allows us to save notation for sequences of arguments which would get very messy. The restriction can also be justified theoretically by encoding n-ary functions as unary-functions on n-tuples, using static functions to construct and destruct the tuples.

3. Goals of the Approach

The goal of Sec. 4 and Sec. 5 will be the definition of a first-order predicate-logic formula $\mathsf{cfc}(R)$ that implies that rule R is clash-free, i.e., never produces inconsistent updates. Formally we want to have

Goal 1 (Consistency)**.** *If* $\mathfrak{A}, \xi \models \mathsf{cfc}(R)$ *and* $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$*, then* U *is consistent.*

The formula $\mathsf{cfc}(R)$ should be computable for the setting of Sec. 2 which allows recursive, nondeterministic rules with parallel and sequential composition. When a set of (mutually recursive) rules is defined, successfully proving $\mathsf{cfc}(R)$ for the body R of each rule should guarantee that runs of the ASM will never produce a clash. Thus, the syntactic check $\mathsf{cfc}(R)$ implies semantic clash-freedom $\mathsf{con}(R)$.

The computation of $\mathsf{cfc}(R)$ can not open up calls $\rho(\underline{t}; \underline{h(u)})$ in the body of a declaration, like the least fixpoint semantics does, since then the computation of $\mathsf{cfc}(R)$ may not terminate, defeating the purpose of generating a proof obligation statically. Instead we have to approximate the effect of a recursive call by assuming that the locations $\underline{h(u)}$ are updated with some unknown value, even though some particular runs of the body may not update it at all.

The main source of clashes are parallel updates R_1 **par** R_2. We will compute $\mathsf{cfc}(R)$ as false for a rule $R = f(t) := t'$ **par** $f(t) := t''$, even though in a concrete run t' and t'' might evaluate to the same value. Since the practically relevant cases we are aware of do not use parallel updates to the same location with the same value, we consider R_1 **par** R_2 to be harmful already when the assigned locations computed by the rules overlap.

Our approach therefore focuses on *sets of potentially modified locations* instead of additionally considering the possible values assigned by updates too.

Goal 2 (Assigned Locations)**.** *We want to define a formula* $\mathsf{asg}(R, f, z)$ *for a rule* R *that approximates its assigned locations.* $\mathfrak{A}, \xi \models \neg\, \mathsf{asg}(R, f, z)$ *and* $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$ *should imply that* U *does not contain an update* $(f, \xi(z), b)$ *for any* b*.*

It is intuitive to view the predicate $\mathsf{asg}(R, f, z)$ as specifying a set of locations $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$. The set $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$ is defined by

$$(f, a) \in \mathsf{Asg}(R)^{\mathfrak{A}}_{\xi} \text{ iff } \mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{asg}(R, f, z).$$

The locations in $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$ are those which are potentially assigned by R when R is executed in $\mathfrak{A}, \xi$. Note that for nondeterministic rules, this set already is an approximation of the locations updated in U. Update sets for

$$R = \textbf{choose } x \textbf{ with } \mathit{true} \textbf{ in } f(x) := 0$$

will assign a single (f, a), but $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$ will be the full set of all (f, a).

Formula $\mathsf{asg}(R, f, z)$ allows to define $\mathsf{cfc}(R_1$ **par** $R_2)$ for a parallel rule. The requirement is that $\exists\, z.\ \mathsf{asg}(R_1, f, z) \wedge \mathsf{asg}(R_2, f, z)$ is false for all functions f assigned in R_1 or R_2, or equivalently that $\mathsf{Asg}(R_1)^{\mathfrak{A}}_{\xi}$ and $\mathsf{Asg}(R_2)^{\mathfrak{A}}_{\xi}$ are disjoint.

Computing the set of assigned locations is difficult for sequential composition.

Example 3. As an example consider

$$R_1 \textbf{ seq } R_2 = \{\, g(y) \coloneqq t \textbf{ seq } f(g(x)) \coloneqq u \,\}$$

Surely g is updated at y, but what about f in R_2? If x and y store different values, f is updated at $g(x)$, but otherwise the updated location depends on the value of t. When the values of assignments are ignored in the computation, the best we can therefore get is

$$\mathsf{asg}(R_1 \textbf{ seq } R_2, f, z) \leftrightarrow z = g(x) \vee x = y$$

When $x = y$ holds, f is potentially assigned at any argument. This formula is a disjunct of $\mathsf{asg}(R_1, f, z)$ (which is false), $\mathsf{asg}(R_2, f, z)$ (which is $z = g(x)$), and the condition that $\mathsf{asg}(R_2, f, z)$ is correct, even when R_2 is not run in the initial state but in some state $\mathfrak{A} + U_1, \xi$, where some update set U_1 computed by R_1 has been applied. □

To compute this formula in general, we need to know which locations (g, b) influence the computation of the assigned locations of f in R_2. In the example it is the single location $(g, \xi(x))$. If none of these locations is assigned in R_1, $\mathsf{asg}(R_2, f, z)$ correctly computes the assigned locations in R_2 even if the rule is not run in the initial state $\mathfrak{A}, \xi$, but in the state after applying the updates of R_1. We therefore have Goal 3.

Goal 3 (Dependent Locations)**.** *Define a predicate* $\mathsf{dep}(R, g, z, f)$ *that determines a set of locations* $\mathsf{Dep}(R, f)^{\mathfrak{A}}_{\xi}$ *by*

$$(g, a) \in \mathsf{Dep}(R, f)^{\mathfrak{A}}_{\xi} \quad \textit{iff} \quad \mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{dep}(R, g, z, f)$$

such that if $\mathfrak{A}'$ *agrees with* $\mathfrak{A}$ *on these locations, the* f*-locations in* $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$ *and in* $\mathsf{Asg}(R)^{\mathfrak{A}'}_{\xi}$ *are the same.*

If $\mathsf{dep}(R, g, z, f)$ holds then the resulting function f after execution of rule R *might* (since it is an overapproximation) depend on the initial value of the location $g(z)$. Conversely, if $\mathsf{dep}(R, g, z, f)$ does not hold, different initial values of $g(z)$ do not lead to different values for f after execution of R.

For rule R_2 from Example 3 we will have

$$\mathsf{dep}(R_2, g, z, f) \leftrightarrow z = x$$

and $\mathsf{dep}(R_2, g', z, f')$ for other functions f', g' will be false, implying that only $(g, \xi(y))$ is relevant to determine the assigned f-locations of R_2. Since this location is assigned in R_1 only when $z = y$, the condition for f being potentially modified at other locations than $(f, \xi(x))$ becomes $\exists\, z.\ z = x \wedge z = y$, which simplifies to $x = y$ as intended.

Sections 4 and 5 will give formal definitions of the three predicates dep, asg, and clf bottom-up, proving the goals stated above as theorems.

$$\mathsf{dept}(g, z, t) \equiv \bigvee_{g(u) \in t} z = u$$

$$\mathsf{depf}(g, z, \forall\, x.\varphi) \equiv \mathsf{depf}(g, z, \exists\, x.\varphi) \equiv \exists\, x.\ \mathsf{depf}(g, z, \varphi)$$

$$\mathsf{depf}(g, z, \varphi \otimes \psi) \equiv \mathsf{depf}(g, z, \varphi) \vee \mathsf{depf}(g, z, \psi) \text{ where } \otimes \in \{\ \rightarrow\ ,\ \leftrightarrow\ ,\ \wedge\ ,\ \vee\ \}$$

$$\mathsf{depf}(g, z, \neg\, \varphi) \equiv \mathsf{depf}(g, z, \varphi)$$

$$\mathsf{depf}(g, z, p(t_1, \ldots, t_n)) \equiv \textstyle\bigvee_{i=1\ldots n} \mathsf{dept}(g, z, t_i)$$

$$\mathsf{depf}(g, z, t_1 = t_2) \equiv \mathsf{dept}(g, z, t_1) \vee \mathsf{dept}(g, z, t_2)$$

Figure 2: Dependency of a formula and term on a location $g(z)$

$$\mathsf{dep}(\mathbf{skip}, g, z, f) \equiv \text{false} \qquad \text{(dep-skip)}$$

$$\mathsf{dep}(\rho(\underline{t}; \underline{h(u)}), g, z, f) \equiv \text{false} \qquad \text{(dep-call)}$$

$$\mathsf{dep}(h(u) := t, g, z, f) \equiv \begin{cases} \mathsf{dept}(g, z, u), & f = h \\ \text{false}, & \text{otherwise} \end{cases} \qquad \text{(dep-asg)}$$

$$\mathsf{dep}(R_1\ \mathbf{seq}\ R_2, g, z, f) \equiv \qquad \text{(dep-seq)}$$
$$\mathsf{dep}(R_1, g, z, f) \vee \mathsf{dep}(R_2, g, z, f)$$
$$\vee \bigvee_{h \in \mathsf{mod}(R_1)} \mathsf{dep}(R_1, g, z, h) \wedge \exists\, z_0.\ \mathsf{dep}(R_2, h, z_0, f)$$

$$\mathsf{dep}(R_1\ \mathbf{par}\ R_2, g, z, f) \equiv \mathsf{dep}(R_1, g, z, f) \vee \mathsf{dep}(R_2, g, z, f) \qquad \text{(dep-par)}$$

$$\mathsf{dep}(\mathbf{if}\, \varphi\, \mathbf{then}\ R_1\, \mathbf{else}\ R_2, g, z, f) \equiv \qquad \text{(dep-if)}$$
$$\mathsf{depf}(g, z, \varphi) \wedge f \in \mathsf{mod}(R_1\ \mathbf{seq}\ R_2)$$
$$\vee\, (\varphi \supset \mathsf{dep}(R_1, g, z, f);\ \mathsf{dep}(R_2, g, z, f))$$

$$\mathsf{dep}(\mathbf{choose}\ x\ \mathbf{with}\ \varphi(x)\ \mathbf{in}\ R, g, z, f) \equiv \qquad \text{(dep-choose)}$$
$$\exists\, x.\ \ \varphi(x) \wedge \mathsf{dep}(R, g, z, f) \vee \mathsf{depf}(g, z, \varphi) \wedge f \in \mathsf{mod}(R)$$

$$\mathsf{dep}(\mathbf{forall}\ x\ \mathbf{do}\ R, g, z, f) \equiv \exists\, x.\ \mathsf{dep}(R, g, z, f) \qquad \text{(dep-forall)}$$

Figure 3: Dependency of f on locations $g(z)$

4. Assigned Locations

This sections gives formal definitions for $\mathsf{dep}(R, g, f, z)$ and $\mathsf{asg}(R, f, z)$ and provides several examples.

All definitions assume that variable z is auxiliary and therefore globally fresh. The definitions abbreviate $(\varphi \to t = u) \wedge (\neg\, \varphi \to t = v)$ as $t = (\varphi \supset u;\ v)$.

The auxiliary definitions of $\mathsf{dept}(g, z, t)$ and $\mathsf{depf}(g, z, \varphi)$ in Fig. 2 express that the (truth) value of term t (formula φ) depends on location $(g, \xi(z))$. The definitions are straightforward.

Fig. 3 defines the dependency predicate $\mathsf{dep}(R, g, z, f)$ that collects locations (g, a) with $a = \xi(z)$ that possibly influence the set of assigned f-locations.

The assigned locations of a call *dep-call* do not depend on any dynamic function, since u must be a static term.

For assignments of the form $f(u) := t$, the assigned f-locations depend on (g, a) if evaluating u depends on the location (g, a) according to *dep-asg*. This is the case if u has a subterm $g(u')$ with $a = u'^{\mathfrak{A}}_{\xi}$. This is expressed by the formula $\mathsf{dept}(g, z, u)$, which is computed as $z = u'$ if $g(u')$ is the only subterm of u involving g.

Example 4. An example is the assignment $f(g(h(x))) := t$, which gives

$$\begin{aligned} &\mathsf{dep}(f(g(h(x))) := t, h, z_0, f) \ \equiv\ z_0 = x \qquad \text{and} \\ &\mathsf{dep}(f(g(h(x))) := t, g, z_1, f) \ \equiv\ z_1 = h(x). \end{aligned}$$ □

The case *dep-seq* in Fig. 3 defines dependency for sequential composition by chaining dependencies of the rule R_1 and R_2 transitively, using a globally fresh variable z_0 for the intermediate location $h(z_0)$.

Example 5. Consider the rule

$$R = R_1 \ \mathbf{seq}\ R_2 \ = \{\, h(g(t_1)) := t \ \mathbf{seq}\ f(h(t_2)) := t' \,\}.$$

Rule R_1 and R_2 have $\mathsf{dep}(R_1, g, z, h) \equiv z = t_1$ and $\mathsf{dep}(R_2, h, z_0, f) \equiv z_0 = t_2$. By the first line of the definition we get the same dependencies for R in place of R_1 and R_2. By transitivity we get $\mathsf{dep}(R, g, z, f)$ when $z = t_1 \wedge \exists\ z_0.\ z_0 = t_2$, which simplifies to $z = t_1$. Indeed modifying h in the first assignment may change the argument of f in the second, thus affecting which f-locations are modified in R. □

Our first intuition was to strengthen the second line of *dep-seq* to

$$\bigvee_{h \in \mathsf{mod}(R_1)} \big(\ \mathsf{dep}(R_1, g, z, h) \wedge \exists\ z_0.\ \mathsf{asg}(R_1, h, z_0) \wedge \mathsf{dep}(R_2, h, z_0, f)\ \big) \qquad \text{(wrong)}$$

This additionally demands that the location $h(z_0)$ that f depends on is actually modified in the first part R_1 of the sequential composition. However, this is not correct, although it is not easy to find a counterexample. We encourage the reader to try to find one before reading on.

For Example 5 this would give the more precise result $\mathsf{dep}(R, g, z, f) \equiv z = x \wedge g(t_1) = t_2$. However, since the dependency is computed in the initial state, this would incorrectly compare $g(t_1)$ and t_2 in the *initial* algebra, while t_2 is evaluated in the state *after* executing R_1. In this algebra functions used in t_2 may have been altered, making the check incorrect, as shown by Example 6.

Example 6. Consider the rule

$$R = R_1 \textbf{ seq } R_2 = \{\, \{\, h(g(c)) := t_1 \textbf{ par } h'(c) := d \,\} \textbf{ seq } f(h(h'(c))) := t_2 \,\}$$

with constants c and d being executed in a context where $g(c) \neq h'(c)$. Using the incorrect definition above would give

$$\mathsf{dep}(R, g, z, f) \equiv z = c \wedge \exists\, z_0.\ z_0 = g(c) \wedge z_0 = h'(c) \qquad \text{(wrong)}$$

which is equivalent to false from the context. The $z_0 = g(c)$ conjunct stems from the additional asg predicate and is wrong. However, if initially $g(c) = d$, the assigned location in R_2 changes from $(f, h(h'(c))^{\mathfrak{A}}_{\xi})$ when executing R_2 alone to $(f, h(d)^{\mathfrak{A}}_{\xi})$ after executing R_1, so there is a dependency. □

According to the definition *dep-if* and *dep-choose* (in Fig. 3) of dependency for **if** and **choose** the locations $h(z)$ occurring in the respective test φ potentially have an influence on the final value of f as well, but only if f is assigned at all. This is illustrated by Example 7.

Example 7.

$$\begin{aligned} & \mathsf{dep}(\textbf{choose } x \textbf{ with } x > g(x) \textbf{ in } f(x) := t, g, z, f) \\ \equiv\ & \exists\, x.\ x > g(x) \wedge \mathsf{dep}(f(x) := t, g, z, t) \vee \mathsf{depf}(g, z, x > g(x)) \\ \equiv\ & \exists\, x.\ x > g(x) \wedge \mathsf{dep}(f(x) := t, g, z, t) \vee z = x \quad \equiv \quad \text{true} \end{aligned}$$

Here $\mathsf{depf}(g, z, x > g(x))$ evaluates to $z = x$ and the existential quantifier therefore evaluates to true. Thus, f depends on the entire dynamic function g. Note that this result is precise and not an overapproximation. □

We remark, that in Example 7 the dependency of f on g could be removed when the f-assignments do not depend on the chosen x at all, e.g., if the assignment would be $f(c) := t$ instead of $f(x) := t$. We have not verified this extension, but adding a definition $\mathsf{dep}(R, x, f)$ which treats x like a nullary function would be possible. The second line of the dep-definition for **choose** and similarly for **forall** could then be strengthened to $\mathsf{dep}(R, x, f) \wedge \mathsf{depf}(g, z, \varphi)$.

The f-locations assigned by a rule R are given by the predicate $\mathsf{asg}(R, f, z)$, i.e., $\mathsf{asg}(R, f, z)$ holds if the location $f(z)$ can potentially be assigned by rule R. Fig. 4 shows the definition of this predicate.

For assignments to $f(u)$ we keep $z = u$ according to case *asg-asg*.

In the case *asg-call* for calls to a procedure with declaration $\rho(\underline{x}; \underline{g})\{R\}$, the location $f(z)$ is assigned if it matches one of the reference parameters $h_i(u_i)$ or if the entire function f is passed as a parameter, and the corresponding formal

$$\mathsf{asg}(\mathbf{skip}, f, z) \equiv \text{false} \tag{asg-skip}$$

$$\mathsf{asg}(g(u) \coloneqq t, f, z) \equiv \begin{cases} z = u, & f = g \\ \text{false}, & \text{otherwise} \end{cases} \tag{asg-asg}$$

$$\mathsf{asg}(\rho(\underline{t}; \underline{h(u)}), f, z) \equiv \begin{cases} \text{true} & f = h_i \text{ and } u_i = \langle\rangle \text{ for some } i \\ & \text{and } g_i \in \mathsf{mod}(R) \\ z = u_i, & \text{otherwise, and } f = h_i \text{ for some } i \\ & \text{and } g_i \in \mathsf{mod}(R) \\ \text{false}, & \text{otherwise} \end{cases} \tag{asg-call}$$

$$\begin{aligned} \mathsf{asg}(R_1 \mathbf{\ seq\ } R_2, f, z) \equiv \quad & \mathsf{asg}(R_1, f, z) \vee \mathsf{asg}(R_2, f, z) \\ \vee \quad & \bigvee_{g \in \mathsf{mod}(R_1)} \exists\, z_0.\ \mathsf{dep}(R_2, g, z_0, f) \wedge \mathsf{asg}(R_1, g, z_0) \end{aligned} \tag{asg-seq}$$

$$\mathsf{asg}(R_1 \mathbf{\ par\ } R_2, f, z) \equiv \mathsf{asg}(R_1, f, z) \vee \mathsf{asg}(R_2, f, z) \tag{asg-par}$$

$$\mathsf{asg}(\mathbf{if}\ \varphi\ \mathbf{then}\ R_1\ \mathbf{else}\ R_2, f, z) \equiv (\varphi \supset \mathsf{asg}(R_1, f, z);\ \mathsf{asg}(R_2, f, z)) \tag{asg-if}$$

$$\mathsf{asg}(\mathbf{choose}\ x\ \mathbf{with}\ \varphi(x)\ \mathbf{in}\ R, f, z) \equiv \exists\, x.\ \varphi(x) \wedge \mathsf{asg}(R, f, z) \tag{asg-choose}$$

$$\mathsf{asg}(\mathbf{forall}\ x\ \mathbf{do}\ R, f, z) \equiv \exists\, x.\ \mathsf{asg}(R, f, z) \tag{asg-forall}$$

Figure 4: Assigned f-locations

parameter g_i is modified in the body R. The restriction that the terms u_i of the reference parameters $\underline{h(u)}$ are static is crucial for the correctness of the asg predicate as shown by Example 8.

Example 8. Given the rules R and R'.

$$R = R'(; f, h(f))$$
$$R'(; f, g) = \{\, f \coloneqq 1\ \mathbf{seq}\ g \coloneqq 2 \,\}$$

According to the definition of asg only the locations f and $h(f)$, where f is evaluated in the initial state, are assigned by R. However, in this context R actually assigns f and $h(1)$. □

The case *asg-seq* for sequential composition considers whether R_1 assigns to some $g(z_0)$ that controls the argument of f as $f(g(u))$ in R_2. In this case, possible values for z are unconstrained if f is modified at all.

Conditionals (*asg-if*) strengthen the check of the branches by the assumption from the test. In a **choose** rule (*asg-choose*), f could be affected by any execution of the body for an x that satisfies the condition φ.

Example 9. Example 3 works out as expected with these definitions.

$$\mathsf{asg}(g(y) \coloneqq t\ \mathbf{seq}\ f(g(x)) \coloneqq u, f, z) \leftrightarrow z = g(x) \vee x = y$$

Note that this is obviously an overapproximation and the formula $z = (x = y \supset t; g(x))$ would be a more precise characterization of the assigned locations. □

The correctness of dep and asg is ensured by Thms. 1 and 2, thus Goal 2 and 3 are achieved. The proofs are delayed until the end of the section until several lemmas have been proven.

Theorem 1 (Assigned Locations Depend on dep only)**.** *For all rules R, functions f, algebras $\mathfrak{A}$, and $\mathfrak{A}'$: If $\mathfrak{A}'$ agrees with $\mathfrak{A}$ on the locations $\mathsf{Dep}(R, f)^{\mathfrak{A}}_{\xi}$, then the f-locations in $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$ and in $\mathsf{Asg}(R)^{\mathfrak{A}'}_{\xi}$ are the same.*

Note that Thm. 1 is about locations $(f, a) \in \mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$, i.e., for locations, where the asg-formula holds. A theorem similar to Thm. 1 does not hold in general when $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$ is replaced with update sets U such that $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$. The reason is that the *values* in an update set might depend on more locations than $\bigcup_f \mathsf{Dep}(R, f)^{\mathfrak{A}}_{\xi}$ as shown by Example 10.

Example 10. Consider the rule $R = f(0) \coloneqq g(5)$, which has $\mathsf{Dep}(R, h)^{\mathfrak{A}}_{\xi} = \emptyset$ for all functions h. However, the update sets produced by R depend on the value of $g(5)$. Therefore, there certainly exist two algebras $\mathfrak{A}$ and $\mathfrak{A}'$, which coincide on $\mathsf{Dep}(R, h)^{\mathfrak{A}}_{\xi}$ for all h, and two update sets U and U' with $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$ and $[\![R]\!]^{\mathfrak{A}'}_{\xi} \triangleright U'$ with $U \neq U'$. □

Theorem 2 (Correctness of asg)**.** *Given an update set U of R, i.e., $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$, and a location (f, a) with $\mathfrak{A}, \xi\{z \mapsto a\} \models \neg\, \mathsf{asg}(R, f, z)$, then $(f, a, b) \notin U$ for any b.*

Note that the theorem is stated for *all* update sets, it does not depend on U being a consistent update set.

For the proofs of Thms. 1 and 2 several lemmas are needed, which are proved first.

Lemma 3 (Coincidence of dep and asg)**.** *Given two auxiliary variables z and z_0 with $z, z_0 \notin t, \varphi, R$ and arbitrary value a then*

$$\mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{dept}(g, z, t) \text{ iff } \mathfrak{A}, \xi\{z_0 \mapsto a\} \models \mathsf{dept}(g, z_0, t)$$
$$\mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{depf}(g, z, \varphi) \text{ iff } \mathfrak{A}, \xi\{z_0 \mapsto a\} \models \mathsf{depf}(g, z_0, \varphi)$$
$$\mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{dep}(R, g, z, f) \text{ iff } \mathfrak{A}, \xi\{z_0 \mapsto a\} \models \mathsf{dep}(R, g, z_0, f)$$
$$\mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{asg}(R, f, z) \text{ iff } \mathfrak{A}, \xi\{z_0 \mapsto a\} \models \mathsf{asg}(R, f, z_0)$$

Proof. By structural induction over term t, formula φ and rule R. □

Lemma 3 states that the truth value of the predicates is independent of the concrete variable name z. Renaming z to z_0 does not change the result. This

implies that the equations

$$(g,a) \in \mathsf{Dept}(t)^{\mathfrak{A}}_{\xi} \text{ iff } \mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{dept}(g,z,t)$$
$$(g,a) \in \mathsf{Depf}(\varphi)^{\mathfrak{A}}_{\xi} \text{ iff } \mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{depf}(g,z,\varphi)$$
$$(g,a) \in \mathsf{Dep}(R,f)^{\mathfrak{A}}_{\xi} \text{ iff } \mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{dep}(R,g,z,f)$$
$$(f,a) \in \mathsf{Asg}(R)^{\mathfrak{A}}_{\xi} \text{ iff } \mathfrak{A}, \xi\{z \mapsto a\} \models \mathsf{asg}(R,f,z)$$

are proper definitions of the sets of dependent and assigned locations. In the Lemmas 4-7 and the proofs of Theorems 1 and 2, we therefore can reason about these sets of locations.

Lemma 4 (Modified Functions)**.** *If* $(g,a) \in \mathsf{Dep}(R,f)^{\mathfrak{A}}_{\xi}$ *or* $(f,a) \in \mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$ *then* $f \in mod(R)$.

Proof. By rule induction. □

Lemma 4 shows that the set of assigned f-locations is empty in all states $\mathfrak{A}, \xi$ (or equivalently: formula asg is always false) for all functions f that are not modified in assignments and not used as reference parameters. These functions also have no dependent locations, on which the (always empty) set of assigned locations could depend.

Lemma 5 (Dependency for Terms)**.** *If two algebras* $\mathfrak{A}$ *and* $\mathfrak{A}'$ *agree on all locations* $(g,a) \in \mathsf{Dept}(t)^{\mathfrak{A}}_{\xi}$*, then* $t^{\mathfrak{A}}_{\xi} = t^{\mathfrak{A}'}_{\xi}$ *and* $\mathsf{Dept}(t)^{\mathfrak{A}'}_{\xi} = \mathsf{Dept}(t)^{\mathfrak{A}}_{\xi}$.

Proof. By structural induction over t. □

Lemma 6 (Dependency for Formulas)**.** *If two algebras* $\mathfrak{A}$ *and* $\mathfrak{A}'$ *agree on all locations* $(g,a) \in \mathsf{Depf}(\varphi)^{\mathfrak{A}}_{\xi}$*, then* $\mathfrak{A}, \xi \models \varphi$ *holds iff* $\mathfrak{A}', \xi \models \varphi$ *holds, and* $\mathsf{Depf}(\varphi)^{\mathfrak{A}'}_{\xi} = \mathsf{Depf}(\varphi)^{\mathfrak{A}}_{\xi}$.

Proof. By structural induction over φ. □

Lemma 7 (Dependency for Rules)**.** *If two algebras* $\mathfrak{A}$ *and* $\mathfrak{A}'$ *agree on all locations from* $\mathsf{Dep}(R,f)^{\mathfrak{A}}_{\xi}$*, then* $\mathsf{Dep}(R,f)^{\mathfrak{A}'}_{\xi} = \mathsf{Dep}(R,f)^{\mathfrak{A}}_{\xi}$.

Proof. By structural induction over R. The case of an assignment $f(t) := u$ uses Lemma 5. The case for **choose** uses Lemma 6 with φ. It also requires Lemma 4 to justify that $\mathsf{depf}(g,z,\varphi)$ must not be considered when $f \notin mod(R)$.

The only other difficult case is $R \equiv R_1$ **seq** R_2. We assume $\mathfrak{A}$ and $\mathfrak{A}'$ are equal on locations in $\mathsf{Dep}(R,f)^{\mathfrak{A}}_{\xi}$ and prove that $\mathsf{Dep}(R,f)^{\mathfrak{A}}_{\xi} = \mathsf{Dep}(R,f)^{\mathfrak{A}'}_{\xi}$. The elements (g,a) of $\mathsf{Dep}(R,f)^{\mathfrak{A}}_{\xi}$ are either from $\mathsf{Dep}(R_1,f)^{\mathfrak{A}}_{\xi}$, $\mathsf{Dep}(R_2,f)^{\mathfrak{A}}_{\xi}$ (where the induction hypothesis directly gives equality to being in $\mathsf{Dep}(R_1,f)^{\mathfrak{A}'}_{\xi}$ or $\mathsf{Dep}(R_2,f)^{\mathfrak{A}'}_{\xi}$), or there is $(h,b) \in \mathsf{Dep}(R_2,f)^{\mathfrak{A}}_{\xi}$ such that $(g,a) \in \mathsf{Dep}(R_2,h)^{\mathfrak{A}}_{\xi}$. By induction hypothesis for R_2 and f we get equivalence to $(h,b) \in \mathsf{Dep}(R_2,f)^{\mathfrak{A}'}_{\xi}$, so it remains to prove $\mathsf{Dep}(R_2,h)^{\mathfrak{A}}_{\xi} = \mathsf{Dep}(R_2,h)^{\mathfrak{A}'}_{\xi}$ when $(h,b) \in \mathsf{Dep}(R_2,f)^{\mathfrak{A}}_{\xi}$. Using the induction hypothesis with rule R_1 and function h proves the result if it

can be established that $\mathfrak{A}$ and $\mathfrak{A}'$ agree on $\mathsf{Dep}(R_1, h)^{\mathfrak{A}}_{\xi}$. This is indeed the case. When (h, b) is in $\mathsf{Dep}(R_2, f)^{\mathfrak{A}}_{\xi}$, the set $\mathsf{Dep}(R_1, h)^{\mathfrak{A}}_{\xi}$ is a subset of $\mathsf{Dep}(R, f)^{\mathfrak{A}}_{\xi}$ by definition. □

Lemmas 5-7 show that modifying locations outside of the dependent locations, neither modifies the semantics of terms, formulas, or rules nor does it modify the computation of dependent locations itself.

With these lemmas the proofs for Thm. 1 and 2 proceed as follows.

Proof of Thm. 1. By structural induction over R. For assignments $f(t) := t'$, lemma 5 is used to ensure $t^{\mathfrak{A}}_{\xi} = t^{\mathfrak{A}'}_{\xi}$, implying that the assigned location is the same. Similarly, for **choose** lemma 6 is used to ensure that those x for which φ is true are the same. The difficult case is again the one for $R \equiv R_1$ **seq** R_2. A location (f, a) is in $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$ if it is in $\mathsf{Asg}(R_1)^{\mathfrak{A}}_{\xi}$, $\mathsf{Asg}(R_2)^{\mathfrak{A}}_{\xi}$, or if there is a location (g, b) that is in both $\mathsf{Asg}(R_1)^{\mathfrak{A}}_{\xi}$ and in $\mathsf{Dep}(R_2, g)^{\mathfrak{A}}_{\xi}$. The first two cases are trivial by induction hypothesis. For the last case, we get equivalence to (g, b) being in $\mathsf{Asg}(R_1)^{\mathfrak{A}'}_{\xi}$ by the induction hypothesis. Finally, that $(g, b) \in \mathsf{Dep}(R_2, g)^{\mathfrak{A}}_{\xi}$ is equivalent to $(g, b) \in \mathsf{Dep}(R_2, g)^{\mathfrak{A}'}_{\xi}$ is the content of the previous lemma 7. □

Proof of Thm. 2. By structural induction over the rule R. The interesting case is again $R \equiv R_1$ **seq** R_2. There are two cases. The first, where R_1 produces an inconsistent update set, is by induction hypothesis. Otherwise $[\![R_1]\!]^{\mathfrak{A}}_{\xi} \triangleright U_1$ with a consistent set U_1 and $[\![R_2]\!]^{\mathfrak{A}+U_1}_{\xi} \triangleright U_2$ such that $U = U_1 \oplus U_2$. Assume, some (f, a, b) is in U but (f, a) is not in $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$. If (f, a) is in U_1, the induction hypothesis for R_1 would give that it is in $\mathsf{Asg}(R_1)^{\mathfrak{A}}_{\xi}$, contradicting that it is not in the superset $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$.

So (f, a, b) must be in U_2. The induction hypothesis implies that (f, a) must be in $\mathsf{Asg}(R_2)^{\mathfrak{A}+U_1}_{\xi}$. If this set is the same as $\mathsf{Asg}(R_2)^{\mathfrak{A}}_{\xi}$, we get a contradiction like in the case $(f, a) \in U_1$ before. According to Theorem 1 (using $\mathfrak{A}' := \mathfrak{A} + U_1$) the sets can differ in the f-location (f, a) only if some location $(g, a') \in \mathsf{Dep}(R_2, f)^{\mathfrak{A}}_{\xi}$ evaluates differently in $\mathfrak{A}$ and $\mathfrak{A} + U_1$. This is possible only if U_1 contains an update (g, a', b'). By the induction hypothesis, this implies $(g, a') \in \mathsf{Asg}(R_1)^{\mathfrak{A}}_{\xi}$. However, this makes the second line of the definition of asg for the case of sequential composition true. There now exists a function g and a z_0 (semantically, the location (g,a')) which is in both $\mathsf{Dep}(R_2, f)^{\mathfrak{A}}_{\xi}$ and $\mathsf{Asg}(R_1)^{\mathfrak{A}}_{\xi}$, implying that $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$ contains all f-locations, contradicting $(f, a) \notin \mathsf{Asg}(R)^{\mathfrak{A}}_{\xi}$. □

5. Checking Clash-Freedom

This section defines a syntactic check for clash-freedom, expressed by the first-order formula $\mathsf{cfc}(R)$. If the formula is provable, then the rule is clash-free (Thm. 4), i.e., $\mathsf{con}(R)$ holds. The static check $\mathsf{cfc}(R)$ of a rule R is defined over

$$\mathsf{cfc}(\mathbf{skip}) \equiv \text{true} \qquad \text{(cfc-skip)}$$

$$\mathsf{cfc}(g(u) \coloneqq t) \equiv \text{true} \qquad \text{(cfc-asg)}$$

$$\mathsf{cfc}(\rho(\underline{t}; \underline{h(u)})) \equiv \text{true} \qquad \text{(cfc-call)}$$

$$\mathsf{cfc}(R_1 \ \mathbf{seq}\ R_2) \equiv \qquad \text{(cfc-seq)}$$
$$\quad \mathsf{cfc}(R_1)$$
$$\wedge\ \Big(\Big(\textstyle\bigwedge_{f \in mod(R_1)} \forall\, z. \neg\, \mathsf{asg}(R_1, f, z) \rightarrow f(z) = f'(\underline{z})\Big) \rightarrow \mathsf{con}\big(R_2{}^{\underline{f'}}_{\mathsf{mod}(R_1)}\big)\Big)$$

$$\mathsf{cfc}(R_1 \ \mathbf{par}\ R_2) \equiv \qquad \text{(cfc-par)}$$
$$\quad \mathsf{cfc}(R_1) \wedge \mathsf{cfc}(R_2)$$
$$\wedge\ \textstyle\bigwedge_{f \in mod(R_1\ \mathbf{par}\ R_2)} \neg\, \exists\, y.\ \mathsf{asg}(R_1, f, y) \wedge \mathsf{asg}(R_2, f, y)$$

$$\mathsf{cfc}(\mathbf{if}\ \varphi\ \mathbf{then}\ R_1\ \mathbf{else}\ R_2) \equiv (\varphi \supset \mathsf{cfc}(R_1);\ \mathsf{cfc}(R_2)) \qquad \text{(cfc-if)}$$

$$\mathsf{cfc}(\mathbf{choose}\ x\ \mathbf{with}\ \varphi(x)\ \mathbf{in}\ R) \equiv (\forall\, x.\ \varphi(x) \rightarrow \mathsf{cfc}(R)) \qquad \text{(cfc-choose)}$$

$$\mathsf{cfc}(\mathbf{forall}\ x\ \mathbf{do}\ R) \equiv \qquad \text{(cfc-forall)}$$
$$\forall\, x. \quad \mathsf{cfc}(R)$$
$$\wedge\ \textstyle\bigwedge_{f \in mod(R)} \neg\, \exists\, x_1, x_2, y.\ x_1 \neq x_2 \wedge \mathsf{asg}(R^{x_1}_x, f, y) \wedge \mathsf{asg}(R^{x_2}_x, f, y)$$

Figure 5: Syntactic Consistency $\mathsf{cfc}(R)$ of Rule R

the structure of rules. Note that all variables that appear on the right hand side of a definition but not on the left are assumed to be globally fresh.

Fig. 4 shows the definition of the $\mathsf{cfc}(R)$ predicate.

Assignments *cfc-asg* and calls *cfc-call* never provoke clashes. Note that this assumes that the called rule ρ is checked separately for clashes, since this only checks that no additional clashes are introduced.

In a sequential composition *cfc-seq*, consistency of R_2 must be checked for possibly modified values of dynamic functions, expressed by fresh function symbols $\underline{f'}$ that are constrained to be the same as the original ones only for arguments that R_1 certainly does not assign. Note that $\underline{f'}$ can be viewed alternatively as new function variables that are implicitly universally quantified. Since none of the rules uses existential quantifiers over $\mathsf{cfc}(R)$-formulas, the universal quantifier can be moved to the top-level, and replaced with a new uninterpreted function symbol in the signature, thus staying in first-order logic. The resulting formula is then evaluated over an extended algebra that provides some interpretation for all introduced f'. When submitting the formula to a prover, f' is just an uninterpreted function without any additional axioms.

Parallel execution *cfc-par* of R_1 and R_2 conservatively excludes assignments

to the same location, where $\mathsf{asg}(R, f, y)$ renames z to a fresh variable y in $\mathsf{asg}(R, f, z)$.

Example 11. To continue Example 3, putting the rule

$$g(y) := t \textbf{ seq } f(g(x)) := u$$

in parallel with any assignment to an f-location will make con false for the combined rule. □

Note that the combination of sequential and parallel composition as in Example 11 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 if assigning the same value to a location twice is regarded as a clash.

Nondeterministic choice *cfc-choose* 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 general parallel execution *cfc-forall* the second line excludes conflicts between two parallel executions of the body. Two fresh different representants x_1 and x_2 of the index x are used to ensure that there are no two assignments to the same $f(y)$.

All sequential rules check trivially as stated by Theorem 3.

Theorem 3. *Given a rule R from the sequential fragment of ASM rules, $\mathsf{cfc}(R)$ holds trivially.*

Proof. By induction over the structure of the rule R. □

Example 12. Typical parallel rules with disjoint tests (e.g. used in the WAM [10])

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

are recognized as clash-free. Lifting a rule $R(; g)$ of one process p with process-local state g to a parallel rule

$$\textbf{forall } p \textbf{ do } R(; f(p))$$

for all processes p is also permissible by the check. This is for example used for the threads of the Java ASM [6]. □

Theorem 4 states that con is correct and therefore Goal 1 is achieved. Note that $\mathsf{cfc}(R)$ needs to be evaluated over an extended algebra that evaluates the new function symbols $\underline{f}'$ that are introduced in the case for sequential composition *cfc-seq*.

Theorem 4 (Correctness of cfc)**.** *Given a rule R with $\mathfrak{A}', \xi \models \mathsf{cfc}(R)$ for every $\mathfrak{A}'$ that extends $\mathfrak{A}$ with an interpretation of the new function symbols used in $\mathsf{cfc}(R)$, then every update set U with $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$ is consistent.*

Proof. Rule induction is needed here to get an induction hypothesis, which directly closes the case of a call, since we can assume the theorem to hold for the body (with appropriate renaming, as done by the semantic rule).

The case of sequential composition is simple here. Given $\mathsf{cfc}(R_1)$, the induction hypothesis implies that R_1 produces a consistent update set U_1. By Theorem 2 the algebra $\mathfrak{A} + U_1$ differs from $\mathfrak{A}$ in at most the locations of $\mathsf{Asg}(R_1)^{\mathfrak{A}}_{\xi}$. Choosing $\mathfrak{A}'$ such that $\underline{f'}^{\mathfrak{A}'} = \underline{f}^{\mathfrak{A}+U_1}$ satisfies the precondition of the second line. This implies $\mathfrak{A} \cup \mathfrak{A}', \xi \models \mathsf{cfc}(R_2{}^{\underline{f'}}_{\mathsf{mod}(R_1)})$. Since $\underline{f'}^{\mathfrak{A}'} = \underline{f}^{\mathfrak{A}+U_1}$, we can rename $\underline{f'}$ back to $\underline{f}$ and get $(\mathfrak{A} + U_1) \cup \mathfrak{A}', \xi \models \mathsf{cfc}(R_2)$. By induction hypothesis this implies that the update set U_2 of R_2 is consistent too, which is sufficient for consistency of the full update set $U_1 \oplus U_2$.

The complex cases in this proof are the parallel rule R_1 **par** R_2 and the rule for **forall** x **do** R. Since the parallel rule can be viewed as the special case of a **forall** with a binary choice, we give the proof for the latter only. The rule R (which may depend on x) computes an update set U_a as $[\![R]\!]^{\mathfrak{A}}_{\xi\{x\mapsto a\}} \triangleright U_a$ for each a. The sets are all consistent themselves by the induction hypothesis. If the union $\bigcup_a U_a$ were inconsistent, then there would be two elements a_1 and a_2, such that U_{a_1} and U_{a_2} both contain an update for the same location (f, b) (with different value). By Theorem 2 (f, b) would be in $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi\{x\mapsto a_1\}}$ and in $\mathsf{Asg}(R)^{\mathfrak{A}}_{\xi\{x\mapsto a_2\}}$. This however, implies that the second line of the definition of con is false by choosing the quantified variables x_1, x_2 and y to be a_1, a_2 and b, contradicting the assumption that $\mathsf{cfc}($**forall** x **do** $R)$ holds. This requires a standard coincidence lemma, which asserts that $\mathfrak{A}, \xi \models \mathsf{asg}(R, f, x)$ holds iff $\mathfrak{A}, \xi\{x_1 \mapsto \xi(x)\} \models \mathsf{asg}(R^{x_1}_x, f, x_1)$ is satisfied. □

6. Relational Encoding

A relational encoding of a clash-free ASM with main rule R is a formula $\varphi(\underline{f}, \underline{f'})$ that describes a state transition of R from the state $\underline{f}$ before applying the rule to the state $\underline{f'}$ after applying it. The two states (algebras) are expressed here using $\underline{f}$ and new primed function symbols $\underline{f'}$, where $\underline{f}$ is the set of dynamic functions of the algebra. The defining property (see Thm. 5) is that $\varphi(\underline{f}, \underline{f'})$ holds if and only if R can calculate an update set U such that applying it to $\underline{f}$ (as the values of in the initial algebra) yields $\mathfrak{A} \oplus U$ where the values are $\underline{f'}$.

Having an explicit formula in (second-order) predicate logic that expresses the effect of an ASM rule allows the use of predicate logic theorem provers to derive properties of ASMs.[2] As an example a partial correctness assertion with precondition pre and postcondition post then can be verified by proving the

[2] This could involve using a standard transformation of second-order logic to first-order logic first, if the prover can handle first-order logic only. See [11] for a specification of dynamic functions that was used when KIV implemented first-order logic only or [12].

predicate logic goal

$$\mathsf{pre} \wedge \varphi(\underline{f}, \underline{f'}) \rightarrow \mathsf{post}^{\underline{f'}}_{\underline{f}}\,.$$

If all domains are finite then model checking temporal properties of ASM runs becomes possible, since the relation φ then is the transition relation of the relevant Kripke structure. Finally, the relational encoding can also be used to verify symbolic execution rules for ASMs as given in Sec. 7.

We define a relational encoding here by first extending the signature of the ASM with predicates $\mathsf{rel}(R'_j)$ for every subrule R'_j syntactically contained in one of the rules R_j of the ASM. Note that formally the rule R'_j is part of the predicate name,[3] and that the set $\underline{\mathsf{rel}}$ of predicates is finite.

The arguments of the predicate of $\mathsf{rel}(R'_j)$ are two sequences of terms, each of which has the function types given by the types of $\underline{f} = \mathsf{dyn}(R_j)$, together with parameters of the types of the free variables $\underline{x} = \mathit{free}(R'_j)$ of this subrule. A typical application of the predicate will have the form $\mathsf{rel}(R'_j)(\underline{\hat{f}}, \underline{\hat{f}'}, \underline{x})$ with two sequences of *function variables* $\underline{\hat{f}}$ and $\underline{\hat{f}'}$ that have the same types as the sequence $\underline{f}$ of used functions. The arguments could be optimized to drop $\hat{f'}_i$ (but not $\hat{f}_i$) when R'_j just reads, but does not modify f_i (i.e. when $f_i \in \mathsf{dyn}(R) \setminus \mathsf{mod}(R)$) since then obviously $\hat{f'}_i = \hat{f}_i$. We have not done so to simplify the presentation.

Fig. 6 gives the main axioms of the relational encoding for clash-free rules. Variable y as well as function variables $\hat{g}, \underline{\hat{f}_1}, \underline{\hat{f}_2}$ and F are assumed to be different from the other variables used in the formulas.

Axiom *rel-asg* updates the modified location $\hat{g}(t)$ to u and leaves all other functions $f \neq g$ unchanged. We write $\hat{g}(t \mapsto u)$ for the updated function.

Axiom *rel-call* assumes a declaration of ρ with $\rho(\underline{y}; \underline{g})\{R\}$, thus the predicate $\mathsf{rel}(R)$ for the body has formal arguments $\hat{g}, \hat{g'}$ and $\underline{y}$. These get instantiated with actual parameters $\underline{\hat{h}(u)}$, $\underline{\hat{g'}}$ and $\underline{t}$. The second conjunct ensures that the final values $\underline{\hat{g'}}$ are propagated back to the locations $\underline{\hat{h}(u)}$. If some $u_i = \langle\rangle$ then the equation $\hat{h'}_i = \hat{h}_i(u_i \mapsto \hat{g'}_i)$ becomes $\hat{h'}_i = \hat{g'}_i$ and the existentially quantified $\hat{g'}_i$ can be optimized away.

Sequential composition *rel-seq* introduces an intermediate state $\underline{\hat{f}_1}$ between execution of R_1 and R_2. Both *rel-seq* and *rel-if* use $\underline{x_1} = \mathit{free}(R_1)$ and $\underline{x_2} = \mathit{free}(R_2)$, which are both subsets of the free variables $\underline{x}$ of the whole rule.

Axiom *rel-par* uses a static, second-order function $\mathsf{merge}(\hat{f}_1, \hat{f}_2, \hat{f})$, which merges two computed functions, and is defined as

$$\hat{f'} = \mathsf{merge}(\hat{f}_1, \hat{f}_2, \hat{f}) \iff \hat{f'}(y) = \begin{cases} \hat{f}_1(y), & \text{if } \hat{f}(y) \neq \hat{f}_1(y) \\ \hat{f}_2(y), & \text{otherwise} \end{cases}$$

[3] For simplicity we assume that two different rules never have the same subrule, otherwise the index j of the rule has to be added to the predicate name, too.

$$\mathsf{rel}(\mathbf{skip})(\hat{\underline{f}}, \hat{\underline{f}}') \;\leftrightarrow\; \hat{\underline{f}}' = \hat{\underline{f}} \tag{rel-skip}$$

$$\mathsf{rel}(g(t) \coloneqq u)(\hat{\underline{f}}, \hat{\underline{f}}', \underline{x}) \leftrightarrow \hat{g}' = \hat{g}(t \mapsto u) \wedge \bigwedge_{f \in \underline{f}, f \neq g} \hat{f}' = \hat{f} \tag{rel-asg}$$

$$\mathsf{rel}(\rho(\underline{t}; \underline{h(u)}))(\hat{\underline{f}}, \hat{\underline{f}}', \underline{x}) \leftrightarrow \tag{rel-call}$$
$$\left(\bigwedge_{f \in \hat{\underline{f}} \setminus \hat{\underline{h}}} \hat{\underline{f}}' = \hat{\underline{f}}\right) \wedge \exists\, \hat{\underline{g}}'.\ \mathsf{rel}(R)(\underline{\hat{h}(u)}, \hat{\underline{g}}', \underline{t}) \wedge \hat{\underline{h}}' = \underline{\hat{h}(u \mapsto \hat{g}')}$$

$$\mathsf{rel}(R_1 \ \mathbf{seq}\ R_2)(\hat{\underline{f}}, \hat{\underline{f}}', \underline{x}) \leftrightarrow \tag{rel-seq}$$
$$\exists\, \hat{\underline{f}}_1.\ \mathsf{rel}(R_1)(\hat{\underline{f}}, \hat{\underline{f}}_1, \underline{x_1}) \wedge \mathsf{rel}(R_2)(\hat{\underline{f}}_1, \hat{\underline{f}}', \underline{x_2})$$

$$\mathsf{rel}(\mathbf{if}\ \varphi\ \mathbf{then}\ R_1\ \mathbf{else}\ R_2)(\hat{\underline{f}}, \hat{\underline{f}}', \underline{x}) \leftrightarrow \tag{rel-if}$$
$$\Big(\varphi \supset \mathsf{rel}(R_1)(\hat{\underline{f}}, \hat{\underline{f}}', \underline{x_1});\ \mathsf{rel}(R_2)(\hat{\underline{f}}, \hat{\underline{f}}', \underline{x_2})\Big)$$

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

$$\mathsf{rel}(R_1\ \mathbf{par}\ R_2)(\hat{\underline{f}}, \hat{\underline{f}}', \underline{x}) \leftrightarrow \tag{rel-par}$$
$$\exists\, \hat{\underline{f}}_1, \hat{\underline{f}}_2.\ \mathsf{rel}(R_1)(\hat{\underline{f}}, \hat{\underline{f}}_1, \underline{x_1}) \wedge \mathsf{rel}(R_2)(\hat{\underline{f}}, \hat{\underline{f}}_2, \underline{x_2}) \wedge \hat{\underline{f}}' = \mathsf{merge}(\hat{\underline{f}}_1, \hat{\underline{f}}_2, \hat{\underline{f}})$$

$$\mathsf{rel}(\mathbf{forall}\ x\ \mathbf{do}\ R)(\hat{\underline{f}}, \hat{\underline{f}}', \underline{x}) \leftrightarrow \tag{rel-forall}$$
$$\exists\, \underline{F}.\ \big(\forall\, x.\ \mathsf{rel}(R)(\underline{f}, \underline{F_x}, \underline{x}, x)\big) \wedge \hat{\underline{f}}' = \mathsf{merge}(\underline{F}, \hat{\underline{f}})$$

Figure 6: Relational encoding for clash-free ASM rules

The equation $\hat{\underline{f}}' = \mathsf{merge}(\hat{\underline{f}}_1, \hat{\underline{f}}_2, \hat{\underline{f}})$ in the rule has to be read as the conjunction of all individual equations for the elements of the four tuples involved. The definition solves the problem of parallel rules R_1 **par** R_2 from the introduction by first computing two individual results $\hat{\underline{f}}_1$ and $\hat{\underline{f}}_2$ for the two rules. Since we know from $\mathsf{con}(R)$ that each location (f, y) is assigned by at most one of the two rules, we can merge both computations to get the final result $\hat{\underline{f}}'$.

The axiom *rel-forall* generalizes from merging two results $\hat{f}_1, \hat{f}_2$ to a set F_x of results, with one result F_x for every x. Formally, function variable F is assumed to have two arguments, and we write $F_x(y)$ instead of $F(x, y)$. This allows to view F_x as a unary (curried) function, that maps every y to $F_x(y)$. For different x, F_x is the result of running rule R with input x. Note that since $x \in \mathit{free}(R)$, all functions F_x may be different. To avoid inconsistency, merging is defined only, when

$$\forall\, x_1, x_2, y.\ F_{x_1}(y) \neq \hat{f}(y) \wedge F_{x_2}(y) \neq \hat{f}(y) \rightarrow x_1 = x_2$$

holds. Then we have

$$\hat{f}' = \mathsf{merge}(F, \hat{f}) \iff \hat{f}'(y) = \begin{cases} F_x(y), & \text{if } F_x(y) \neq \hat{f}(y) \text{ for some (unique) } x \\ \hat{f}(y), & \text{otherwise} \end{cases}$$

Again, $\underline{\hat{f}}' = \mathsf{merge}(\underline{F}, \underline{\hat{f}})$ has to be read as a conjunction. That the precondition is valid, depends on $\mathsf{con}(\textbf{forall } x \textbf{ do } R)$ having been proved, e.g. by checking $\mathsf{cfc}(\textbf{forall } x \textbf{ do } R)$, which ensures that at most one x causes an update of $f(y)$.

Finally, the relational encoding $\varphi(\underline{f}, \underline{f}')$ we are looking for is the conjunction of the formula $\mathsf{rel}(R)(\underline{f}, \underline{f}')$ for the main rule R, which has no free variables $\underline{x}$, with all the axioms. Note that $\mathsf{rel}(R)(\underline{f}, \underline{f}')$ uses two sequences of *dynamic functions* as arguments, that describe the state before and after the rule, instead of *function variables*.

If the ASM is hierarchical, i.e., has no recursive rules, then the axioms of Fig. 6 are nonrecursive (hierarchical) definitions of all predicates, so they trivially have a unique interpretation. Given any algebra $\mathfrak{A}$ for the signature of the ASM, there is always a unique algebra $\mathfrak{A}^+$ that extends $\mathfrak{A}$ with interpretations of the predicates and of the two variants of merge such that the axioms are valid (conservative extension).

If recursive rules are present, then the axioms form a conservative extension, provided an axiom is added that characterizes the predicates to have the unique least fixpoint interpretation according to Knaster-Tarski's fixpoint theorem [8]. The least fixpoint of recursive definitions defines the extension $\mathsf{rel}(R'_j)^{\mathfrak{A}^+}$ for each predicate $\mathsf{rel}(R'_j)$ as the smallest one possible. In other words, each predicate is true in $\mathfrak{A}^+$ for those arguments only, where this is implied by the axioms. According to Knaster-Tarski's theorem the least fixpoint can be uniquely characterized by the higher-order axiom

$$\forall\, \underline{\mathsf{rel}'}.\ Ax^{\underline{\mathsf{rel}'}}_{\underline{\mathsf{rel}}} \to \bigwedge_{\mathsf{rel}_i \in \underline{\mathsf{rel}}} \forall\, \underline{\hat{f}}, \underline{\hat{f}}', \underline{x}.\ \mathsf{rel}'_i(\underline{\hat{f}}, \underline{\hat{f}}', \underline{x}) \to \mathsf{rel}_i(\underline{\hat{f}}, \underline{\hat{f}}', \underline{x})$$

that characterizes the intersection of all possible interpretations. Here $\underline{\mathsf{rel}}$ is the sequence of all predicates defined, and $\mathrm{Ax}(\underline{\mathsf{rel}})$ is the conjunction of the (universally quantified) axioms of Fig. 6. New predicate variables $\underline{\mathsf{rel}'}$ are used to quantify over all possible interpretations that satisfy the axioms. This characterization still admits rule induction, i.e., proving a property for $\underline{\mathsf{rel}}$ can be done by proving it for $\underline{\mathsf{rel}}$ on the left hand side of the recursion, assuming it holds for all calls on the right hand side.[4]

A simpler characterization via Kleene's fixpoint theorem is possible, if the recursion is continuous, which is the case if all **forall** x constructs used in the ASM iterate over a finite domain. All practically relevant ASMs we know of satisfy this constraint and some papers, e.g. [14], explicitly enforce it. The use

[4] See e.g. [13], chapter 3.5 on the monotone μ-calculus, which can be used to formally justify mutual recursive definitions, and to derive the principle of rule induction.

of the higher-order axiom above with predicate variables $\underline{\mathsf{rel}'}$ is then replaced with structural induction over recursion depth. Formally, auxiliary predicates $\mathsf{relh}(R)$ with an extra argument n for the recursion depth are needed. The axiom *rel-seq* of Fig. 6 is replaced with

$$\mathsf{relh}(R_1 \,\mathbf{seq}\, R_2)(\underline{\hat{f}}, \underline{\hat{f}'}, \underline{x}, n) \leftrightarrow \exists\, \underline{\hat{f}_1}.\ \mathsf{rel}(R_1)(\underline{\hat{f}}, \underline{\hat{f}_1}, \underline{x_1}, n) \wedge \mathsf{rel}(R_2)(\underline{\hat{f}_1}, \underline{\hat{f}'}, \underline{x_2}, n)$$

All other axioms except for *rel-call* are changed similarly. In *rel-call* the recursion depth is decremented and we get no result (nontermination), if it is already zero.

$$\begin{aligned}\mathsf{relh}(\rho(\underline{t}; \underline{h(u)}))(\underline{\hat{f}}, \underline{\hat{f}'}, \underline{x}, n) \leftrightarrow n \neq 0 \wedge \bigwedge_{f \in \underline{f} \setminus \underline{h}} \hat{f}' = \hat{f} \\ \wedge\ \exists\, \underline{\hat{g}'}.\ \mathsf{rel}(R)(\underline{\hat{h}(u)}, \underline{\hat{g}'}, \underline{t}, n-1) \wedge \underline{\hat{h}'} = \underline{\hat{h}(u \mapsto \hat{g}')}\end{aligned}$$

Finally, for each R'_j axioms of the form

$$\mathsf{rel}(R'_j)(\underline{\hat{f}}, \underline{\hat{f}'}, \underline{x}) \leftrightarrow \exists\, n.\ \mathsf{relh}(R'_j)(\underline{\hat{f}}, \underline{\hat{f}'}, \underline{x}, n)$$

are needed. Together with standard axioms for natural numbers (including the induction scheme) they express that a rule changes the state from $\underline{\hat{f}}$ to $\underline{\hat{f}'}$ if it does so with some finite recursion depth. Note that with infinite iteration in **forall** a characterization with recursion depth is no longer possible as demonstrated by Example 13.

Example 13. Consider an ASM with main rule R and auxiliary rule R_1.

$$\begin{aligned}R &= \mathbf{forall}\ m\ \mathbf{do}\ R_1(m) \\ R_1(m) &= \mathbf{if}\ m > 0\ \mathbf{then}\ R_1(m-1)\ \mathbf{else\ skip}\end{aligned}$$

The rule terminates with an empty update set, so $\mathsf{rel}(R)()$ (which has no arguments) should be equivalent to true. However, since there is no finite recursion depth n such that $\mathsf{relh}(R)(n)$ is valid (each bound will be violated by some call to R_1), the incorrect characterization would evaluate $\mathsf{rel}(R)()$ as false. □

To formulate the correctness of rel as a theorem, we need signatures and algebras with renamed function symbols. Let Σ' be the signature that is obtained by replacing all dynamic functions f with primed versions f', assuming Σ does not itself have function symbols with primes. Then $\mathfrak{A}'$ is defined to be the algebra with signature Σ' that defines $f'^{\mathfrak{A}'}$ to be $f^{\mathfrak{A}}$. We write $\mathfrak{A}^+ \cup \mathfrak{B}'$ for the (disjoint) union of the algebras $\mathfrak{A}^+$ (that for the given ASM uniquely extends $\mathfrak{A}$ with predicate symbols $\underline{\mathsf{rel}}$) and $\mathfrak{B}'$. The algebra therefore has signature $\Sigma \cup \Sigma' \cup \underline{\mathsf{rel}}$. With this notation we have

Theorem 5. *Given an ASM with main rule R, such that $\mathsf{con}(R_j)$ holds for all rule bodies R_j, and two algebras $\mathfrak{A}$ and $\mathfrak{B}$ that differ in at most $\mathsf{dyn}(R)$, then $\mathfrak{A}^+ \cup \mathfrak{B}', \xi \models \mathsf{rel}(R'_j)(\underline{f}, \underline{f'})$ if and only if there is some (consistent) U such that $[\![R'_j]\!]^{\xi}_{\mathfrak{A}} \triangleright U$ and $\mathfrak{B} = \mathfrak{A} + U$.*

Proof. By applying the following inductive Lemma 8 on the main rule, which has no free variables and $\underline{f} = \mathsf{dyn}(R)$ as reference parameters, by instantiating function variables $\underline{\hat{f}}, \underline{\hat{f}}'$ with $\underline{f}, \underline{f}'$. □

Lemma 8. *Assume a clash-free ASM where* $\mathsf{con}(R_j)$ *holds for all rules. Let* R *be any subrule of rule* R_j, $\underline{f} = \mathsf{dyn}(R_j)$ *and* $\underline{x} = \mathit{free}(R)$. *Let* $\mathfrak{A}$ *be any algebra for the signature of the ASM extended with any interpretation of the formal parameters* $\underline{g}$ *of the declaration of* R_j. *Let* $\mathfrak{A}^+$ *be the unique extension of* $\mathfrak{A}$ *that also interprets* $\underline{\mathsf{rel}}$. *Then*

- $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$ *implies* $\mathfrak{A}^+, \xi\{\underline{\hat{f}}, \underline{\hat{f}}' \mapsto \underline{f}^{\mathfrak{A}}, \underline{f}^{\mathfrak{A}\oplus U}\} \models \mathsf{rel}(R)(\underline{\hat{f}}, \underline{\hat{f}}', \underline{x})$.
- *Conversely, if* $\xi(\underline{\hat{f}}) = \underline{f}^{\mathfrak{A}}$ *and* $\mathfrak{A}^+, \xi \models \mathsf{rel}(R)(\underline{\hat{f}}, \underline{\hat{f}}')$, *then there is an update set* U *such that* $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$ *and* $\underline{f}'^{\mathfrak{A}\oplus U} = \xi(\underline{\hat{f}}')$.

Proof. The proof has two parts for the two directions of the lemma. The "if" direction, which assumes $[\![R]\!]^{\xi}_{\mathfrak{A}} \triangleright U$ is proved by rule induction over the definition of the derivation relation from Fig. 1, while the "only if" direction, which assumes $\mathfrak{A}^+ \cup \mathfrak{B}', \xi \models \mathsf{rel}(R)(\underline{\hat{f}}, \underline{\hat{f}}', \underline{x})$ is by rule induction over rel from Fig. 6. Since the structure of the recursion in both definitions is the same, the two proofs are rather similar. The proof for assignment and **choose** is simple, the proof for calls applies Lemma 2 from Sec. 2.

The "if" proof for sequential composition must prove that the assertion $\mathfrak{A}^+, \xi\{\underline{\hat{f}}, \underline{\hat{f}}' \mapsto \underline{f}^{\mathfrak{A}}, \underline{f}^{\mathfrak{A}\oplus U}\} \models \mathsf{rel}(R_1 \textbf{ seq } R_2)(\underline{\hat{f}}, \underline{\hat{f}}', \underline{x})$, assuming $[\![R_1 \textbf{ seq } R_2]\!]^{\xi}_{\mathfrak{A}} \triangleright U$ holds. Since all rules are clash-free by assumption, so is set U. Therefore the set can only be the result of applying *sem-seq-cons* from Fig. 1 (since *sem-seq-incons* derives an inconsistent set), implying there is a consistent set U_1 and an arbitrary set U_2 with $[\![R_1]\!]^{\xi}_{\mathfrak{A}} \triangleright U_1$ and $[\![R_2]\!]^{\xi}_{\mathfrak{A}+U_1} \triangleright U_2$ such that $U = U_1 \cup U_2$. U_2 must be consistent too, otherwise U would be inconsistent. The induction hypothesis for R_1 and R_2 therefore gives $\mathfrak{A}^+, \xi\{\underline{\hat{f}}, \underline{\hat{f}}' \mapsto \underline{f}^{\mathfrak{A}}, \underline{f}^{\mathfrak{A}\oplus U_1}\} \models \mathsf{rel}(R_1)(\underline{\hat{f}}, \underline{\hat{f}}')$ and $\mathfrak{A}^+, \xi\{\underline{\hat{f}}, \underline{\hat{f}}' \mapsto \underline{f}^{\mathfrak{A}\oplus U_1}, \underline{f}^{\mathfrak{A}\oplus U}\} \models \mathsf{rel}(R_2)(\underline{\hat{f}}, \underline{\hat{f}}')$. Renaming the intermediate function variables suitably and applying axiom *rel-seq* implies the desired result. The reverse direction for the sequential case is similar.

It remains to prove **par** and **forall**. Like in the proof for Theorem 4 the proof for **par** can be viewed as the special case of **forall** with binary choice, so we only give the "only if" direction for **forall**. The "if" direction is again similar.

The proof has to show that when $\mathfrak{A}^+, \xi \models \mathsf{rel}(\textbf{forall } x \textbf{ do } R)(\underline{\hat{f}}, \underline{\hat{f}}', \underline{x})$ and $\xi(\underline{\hat{f}}) = \underline{f}^{\mathfrak{A}}$ hold, an update set U can be found with $[\![\textbf{forall } x \textbf{ do } R]\!]^{\xi}_{\mathfrak{A}} \triangleright U$ and $\underline{f}'^{\mathfrak{A}\oplus U} = \xi(\underline{\hat{f}}')$. Expanding the assumption using *rel-forall* and its quantifiers, we get that there are (binary) functions $\underline{G}$ such that for all a $\mathfrak{A}^+, \xi\{\underline{F}, x \mapsto \underline{G}, a\} \models \mathsf{rel}(R)(\underline{\hat{f}}, \underline{\hat{f}}', \underline{F}_x, \underline{x}, x)$ holds, and also that $\mathfrak{A}^+, \xi\{\underline{F} \mapsto \underline{G}\} \models \mathsf{merge}(\underline{F}_x, \underline{\hat{f}}')$ is true. Applying the induction hypothesis for each a we get that there are update sets U_a with $[\![R]\!]^{\xi\{x\mapsto a\}}_{\mathfrak{A}} \triangleright U_a$ and that function $\lambda b.\, G(a, b)$ agrees with $\underline{f}^{\mathfrak{A}\oplus U_a}$.

Using *sem-forall* implies $[\![\textbf{forall } x \textbf{ do } R]\!]_{\mathfrak{A}}^{\xi\{x\mapsto a\}} \triangleright U$ for the union U of all update sets U_a. The modification of function variables $\underline{F}$ to be $\underline{G}$ can be dropped by coincidence. U (and all U_a) are consistent since the rule is assumed to be consistent, so at most one U_a contains an update for each location $f_i(b)$, where $f_i \in \underline{f}$. The result of $\mathsf{merge}(F, \hat{\underline{f}}')$ therefore is defined and we finally have to show that the definition of merge implies $\xi(\hat{\underline{f}}') = \underline{f}^{\mathfrak{A}\oplus U}$. The consistency of U ensures that for each function $f_i \in \underline{f}$ and each argument b there is at most one U_a with an update of $(f_i, b, c) \in U_a$. If the update exists we have $G_i(a, b) = c$. Therefore, if $c \neq {f_i}^{\mathfrak{A}}(b)$ then a and b will be the values for x and y used in $F_{ix}(y)$ in the definition of merge and we get that $\xi(\hat{f}_i)(b) = c = {f_i}^{\mathfrak{A}\oplus U}(b)$ as desired. If there is no update or if the update does not modify f_i, the "otherwise" clause of the merge definition applies, which ensures $\xi(\hat{f}_i)(b) = {f_i}^{\mathfrak{A}}(b) = {f_i}^{\mathfrak{A}\oplus U}(b)$ as desired. □

7. Calculus

This section defines a weakest liberal precondition calculus for ASMs based on the relational encoding provided in Sec. 6. The calculus is only applicable and sound if all rules are proven to be clash-free, either by the static check given in Sec. 5 or some other method. The main feature of the calculus is that parallel rules are executed sequentially and results are merged.

We extend the formulas of first-order logic by the modality $[R]\,\psi$ (weakest liberal precondition) with the semantics given by Def. 7.

Definition 7 (Weakest Liberal Precondition)**.**

$$[\![[R]\,\psi]\!]_{\xi}^{\mathfrak{A}} \qquad \textit{iff for all consistent } U \textit{ with } [\![R]\!]_{\xi}^{\mathfrak{A}} \triangleright U\colon \quad \mathfrak{A} + U, \xi \models \psi$$

Fig. 7 shows the rules of the calculus, which are similar to dynamic logic [15]. For simplicity, to avoid conversions between function symbols and function variables as in the relational encoding, we assume the signature to contain a supply of (uninterpreted) fresh function symbols f', f_1, f_2, F etc. for every dynamic function f that is used in the ASM. This is sufficient, since in contrast to the relational encoding the symbolic execution calculus does not need to quantify over functions.

The rule *calc-asg* for assignment introduces a fresh updated state f'. The postcondition ψ is then expressed over this updated state. Note that for the syntactic renaming of f with f' in ψ it is crucial that calls (ψ may be a box-formula with a call) mention all their parameters. Otherwise, when the call is opened up, the body would still modify the original state instead of the renamed one. A useful calculus for symbolic execution for a logic which is able (in contrast to Hoare's Logic) to have program formulas with several programs, such as $[R_2]\,\varphi \to [R_2]\,\varphi$ to express correctness of program transformations, must be able to execute one box-formula yielding an intermediate state without affecting the state, that another box-formulas starts with. This enforces that the initial state

$$\frac{\psi}{[\mathbf{skip}]\,\psi} \qquad \text{(calc-skip)}$$

$$\frac{f' = f(t \mapsto u) \rightarrow \psi^{f'}_{f}}{[f(t) \coloneqq u]\,\psi}\ f' \text{ fresh} \qquad \text{(calc-asg)}$$

$$\frac{[R^{\underline{t}\ \underline{h(u)}}_{\underline{x}\ \underline{g}}]\,\psi}{[\rho(\underline{t};\underline{h(u)})]\,\psi}\ \text{with declaration } \rho(\underline{x};\underline{g}).R \qquad \text{(calc-call)}$$

$$\frac{[R_1]\,[R_2]\,\psi}{[R_1\ \mathbf{seq}\ R_2]\,\psi} \qquad \text{(calc-seq)}$$

$$\frac{\varphi \rightarrow [R_1]\,\psi \qquad \neg\,\varphi \rightarrow [R_2]\,\psi}{[\mathbf{if}\ \varphi\ \mathbf{then}\ R_1\ \mathbf{else}\ R_2]\,\psi} \qquad \text{(calc-if)}$$

$$\frac{\forall\ x.\ \varphi(x) \rightarrow [R]\,\psi}{[\mathbf{choose}\ x\ \mathbf{with}\ \varphi(x)\ \mathbf{in}\ R]\,\psi} \qquad \text{(calc-choose)}$$

$$\frac{\underline{f}_1 = \underline{f} \wedge \underline{f}_2 = \underline{f} \rightarrow [R_1{}^{\underline{f}_1}_{\underline{f}}]\,[R_2{}^{\underline{f}_2}_{\underline{f}}]\,\left(\underline{f}' = \mathsf{merge}(\underline{f}_1, \underline{f}_2, \underline{f}) \rightarrow \psi^{\underline{f}'}_{\underline{f}}\right)}{[R_1\ \mathbf{par}\ R_2]\,\psi} \qquad \text{(calc-par)}$$

$$\frac{[R]\,\chi(x) \qquad \left(\forall\ x.\ \chi(x)^{F_x}_{\underline{f}}\right)\ \wedge\ \underline{f}' = \mathsf{merge}(\underline{F}, \underline{f}) \rightarrow \psi^{\underline{f}'}_{\underline{f}}}{[\mathbf{forall}\ x\ \mathbf{do}\ R]\,\psi} \qquad \text{(calc-forall)}$$

Figure 7: Rules of the Calculus

of all functions that are modified in the body of a call must be syntactically represented as arguments of the call.

For clash-free parallel rules we use merge similar to the relational encoding. Note that it is sufficient that $\underline{f}$ ranges over all modified functions $\underline{f} = \mathsf{mod}(R) = \mathsf{mod}(R_1) \cup \mathsf{mod}(R_2)$.

For **par** (*calc-par*) two fresh states $\underline{f}_1$ and $\underline{f}_2$ with an initial value of $\underline{f}$ are introduced for R_1 resp. R_2. R_1 then updates $\underline{f}_1$ and R_2 updates $\underline{f}_2$. Afterwards these two are merged into the fresh variable $\underline{f}'$. The postcondition is then expressed over the merged state $\underline{f}'$.

The rule *calc-forall* abstracts each of the parallel computations to a formula χ that characterizes each individual execution of R in isolation (first premise). The second premise can intuitively be understood as proving the postcondition ψ from the results $\chi(x)$ for all indices x. However, $\chi(x)$ for a particular x must be weakened by potential interference of other parallel executions of R.

Note that both rules execute the computations in the initial state, i.e., its reads yield values from the initial state, unless that value was overwritten in the computation that does the reading. The rule *calc-par* could be organized like *calc-forall*. However, it seems more convenient to execute the branches sequentially. This avoids the need to define postconditions $\chi(1)$ and $\chi(2)$ in advance.

Example 14. A simple example is the formula

$$[\textbf{forall } x \textbf{ do } f(x) \coloneqq x] \forall\, x.f(x) = x$$

Application of the calculus rule *calc-forall* with $\chi(x) \equiv f(x) = x$ yields the two premises

$$(1.) \qquad [f(x) \coloneqq x]\, f(x) = x$$
$$(2.) \qquad (\forall\, x.\ F_x(x) = x) \wedge f' = \mathsf{merge}(F, f) \rightarrow \forall\, x.f'(x) = x$$

The first premise is trivially true. In the second premise the two conjuncts imply $f'(x) = F_x(x) = x$ for every x, which yields the post condition. □

The choice of the formula $\chi(x)$ requires some insight into the algorithm and a creative step, similar to the invariant rule for while programs in dynamic logic or Hoare logic. Viewing **forall** as a for-loop that iterates over all x can give an idea, though the effects of "earlier" iterations are invisible in "later" ones.

Example 15. If we have proven the property $[p(\underline{t}; g)]\, \psi(g)$ for a procedure p, we can use it trivially for rules of the form

$$\textbf{forall } x \textbf{ do } p(\underline{t}; f(x))$$

by using $\chi(x) \equiv \psi(f(x))$ for *calc-forall*. □

The soundness of the calculus is proven based on the relational encoding.

Theorem 6 (Soundness of the Calculus)**.** *If the rule R is clash-free and $[R]\,\varphi$ is derivable with the calculus rules shown in Fig. 7 in the algebra $\mathfrak{A}$ with valuation ξ, then $\mathfrak{A} + U, \xi \models \psi$ holds for all U with $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$.*

Proof. By Thm. 5 it only needs to be shown that if $[R]\,\varphi$ holds then for all $\mathfrak{B}'$ with $\mathfrak{A}^+ \cup \mathfrak{B}', \xi \models \mathsf{rel}(R'_j)(\underline{f}, \underline{f}', \underline{x})$ implies $\mathfrak{B}', \xi \models \varphi$, where $\mathfrak{B}$ may only differ with $\mathfrak{A}$ on $\mathsf{mod}(R)$. The proof uses structural induction over the derivation tree and just unfolds the definition of rel. □

8. Related Work

8.1. Clash-Freedom Check

The approach most closely related to ours is the logic for ASMs defined by Stärk and Nanchen [4] (also given in [5]). We use the same syntax and semantics of ASMs with two exceptions. We only allow *static* by-name parameters, which

then coincide with value parameters, to facilitate modular reasoning for calls (cf. Lemma 2). Furthermore, we added explicit reference parameters, which are useful to make the static check for clash-freedom more precise.

The predicate $\mathsf{con}(R)$ we define is similar to the predicate $\mathsf{con}(R)$ defined in [4] but 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 [4], which makes rules and verification conditions at least harder to read.

For hierarchical ASMs without nondeterminism, $\mathsf{con}(R)$ can be expanded by unfolding all calls, so this gives a precise check for clash-freedom. For ASMs with recursive rules just expanding the definitions in [4] would lead to an infinite computation. Note that the completeness theorem of the paper that permits to eliminate modal constructs is for *hierarchical* ASMs only, where recursion is forbidden.

The approach in [14] extends [4] to nondeterminism, and formalizes clash-freedom as a formula scon (strong consistency). The paper does not consider calls, although it should be possible to add them. Unfolding the definition scon gives a precise clash-freedom check for hierarchical nondeterministic ASMs, though the resulting formula is now in a specialized logic with operators $\in^1$ and $\in^2$ and second-order variables X that encode update sets syntactically. It seems to be possible to translate the result into second-order logic by using separate predicates $\in^1_f$ for every dynamic function f.

One motivation when we defined our clash-freedom check was that expanding definitions as necessary in [4] and [14] will yield a very large formula even for medium sized ASMs. When a proof for the formula fails it will be hard to pinpoint the subrule responsible. Their clash-freedom check is also purely first-order, which is implied by the results in [16].

In contrast the clash-freedom check $\mathsf{cfc}(R)$ we define does not use modal constructs ($[R]\ \varphi$) and statically computes a formula for each (sub-)rule separately, even for recursive rules, since the computation stops at calls. Of course for a hierarchical ASM opening up all definitions is still possible and will make the check more precise, though the check will still be stronger than $\mathsf{con}(R)$.

The price we pay for having a computable $\mathsf{cfc}(R)$ for all rules is that our predicate only approximates clash-freedom. There are clash-free rules which our check 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 $\mathsf{cfc}(R)$. An SMT solver should suffice for many practical cases to establish clash-freedom.

8.2. Strong vs. Weak Consistency

[4] also discusses weak consistency $\mathsf{wcon}(R)$ as an alternative to clash-freedom for nondeterministic rules. Weak consistency holds iff for every $\mathfrak{A}$ and ξ there is at least some consistent update sets U with $[\![R]\!]^{\mathfrak{A}}_{\xi} \triangleright U$. The calculus in [14] gives a formal definition.

Checking weak consistency instead of clash-freedom is harder, in particular our approach based on the set of potentially assigned locations is useless.

We object to the use of weakly consistent rules (except maybe when hidden behind aggregation operators [17]) for several reasons. First, in analogy to "a potentially diverging program is as bad as an always diverging one" as used in standard definitions of total correctness we believe that specifying an ASM that randomly crashes when simulating runs is a bad idea.

Second, weakly consistent rules are not compatible with standard refinement based approaches, e.g. ASM refinement [5] formalized in [18] as well as most other refinement definitions, that allow to reduce nondeterminism. Implementing a weakly consistent rule that nondeterministically computes either consistent U_1 or inconsistent U_2 with one that always computes the inconsistent update set U_2 is clearly not a desirable refinement. At least special proof obligations would be required to rule out this case.

Third, $\mathsf{wcon}(R)$ does not imply $\mathsf{con}(R)$ in the presence of recursive rules, as it implies the existence of a terminating run. A deterministic rule, that does not terminate is clash-free, but not weakly consistent. An axiomatization of weak consistency for recursive rules therefore also has to consider termination.

Fourth, ASM rules that are close to an implementation often still have nondeterminism, that is resolved when translating the rules to code in a programming language. Examples are calling a scheduler (e.g. the one of the JVM) or calling malloc in C to implement the (nondeterministic) choice of a new memory location. Having random clashes that lead to exceptional behavior of the generated code is clearly undesirable too.

8.3. Other Semantics for ASMs

We have not considered alternative semantics for ASMs that compute multisets instead of sets U. Such a semantics is used in work on parallel ASMs [19, 20, 21], in the concurrent ASM thesis [22], or when aggregating update multisets into single updates. See e.g. the let rule in [17], which is used to model access to databases. The paper [23] defines an extension of the logic in [14] that can handle multi-sets using specialized operators and second-order variables.

8.4. Relational Encoding and Calculus

In parallel to our work, a relational encoding of ASMs to Event-B was developed in [24]. In contrast to ours, the clash-freedom check there is exact and tolerates rules, which compute the same update several times in parallel. 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 nondeterminism, recursion and sequential composition, so it is not sufficient to support the rules used in KIV. An interesting idea used in the approach is that the union $U_1 \cup U_2$ of two consistent sets U_1 and U_2 is consistent iff $U_1 \oplus U_2$ and $U_2 \oplus U_1$ are the same set. For deterministic rules, checking whether R_1 **par** R_2 produces a clash on top-level therefore can be done, by running $(R_1 \textbf{ seq } R_2)\frac{f_1}{\underline{f}}$ and $(R_2 \textbf{ seq } R_1)\frac{f_2}{\underline{f}}$

on two copies $\underline{f_1}$ and $\underline{f_2}$ (that are initialized with $\underline{f}$), and by finally checking for $\underline{f_1} = \underline{f_2}$ at the end. Since Event-B lacks a sequential composition operator, the approach in [24] realizes this idea by sequentially composing update operators.

Note that our calculus needs to run each sub-rule once, not twice as in the approach above. However, running $\mathsf{merge}(\underline{f_1}, \underline{f_2}, \underline{f})$ after $R_1 \frac{f_1}{\underline{f}}$ **seq** $R_2 \frac{f_2}{\underline{f}}$ cannot detect the difference between the inconsistent rule $f(x) := f(x)$ **par** $f(x) := f(x)+1$ and the clash-free rule **skip par** $f(x) := f(x)+1$, while the commutation check can. Therefore we must assume that clash-freedom has been proved in advance to ensure our approach is valid. Of course our check for clash-freedom trivially fails for the first, but succeeds for the second rule.

Our relational encoding rel for clash-free rules shows some similarities to [14], in particular higher-order functions are used in both. For a clash-free rule R our definition of $[R]\,\varphi$ has the same semantics.

A main difference is that we separate the clash-freedom check from the definition of the effect of a rule, which makes the relational encoding simpler than the logic in [14] that mimics the semantic of ASMs in the logic (by having syntactic variables X that describe update sets U), which must be checked to be consistent (a predicate $conUSet(X)$ is axiomatized for this purpose) when proving $[R]\,\varphi$. Instead we only need new functions for the semantics of dynamic functions in state $\mathfrak{A} \oplus U$, since consistency of U has been checked in advance.

We are not aware of related work that defines a symbolic execution calculus for ASMs, while symbolic execution for sequential programs is of course well-known [25].

9. Conclusion & Outlook

In this paper we have defined a static clash-freedom check for ASM rules, a relational encoding and a calculus for clash-free ASM rules. Since usually ASM rules are required to be clash-free, the three results contribute to separating proofs of clash-freedom, from proving properties of clash-free ASM rule applications (or of whole ASM runs) using either a direct relational encoding (typically for automatic proofs) or a more intuitive calculus of symbolic execution (for interactive proofs).

The static check we have defined might lead to false positives, i.e., the check is sound, but not complete. If a false positive occurs a stronger method for establishing clash-freedom might be employed. For hierarchical ASM the axioms given in [14] could probably be used in this case (by eliminating all special operators). When restricting calls as we have done in Sec. 2, then for ASMs with recursion it might still be possible to extend this axiomatization using least fixpoints, since still a finite number of predicates is involved. Otherwise finding a relational encoding (with finitely many axioms) is still an open question.

The main feature of our calculus for symbolic execution is that parallel execution (**par**) reduces to sequential execution and merging the results. Furthermore, the **forall** construct allows for a rule that is similar to the invariant rule of dynamic or Hoare logic.

We have verified some of the results of this paper in KIV by a predicate logic embedding of ASM rules and their semantics (see the URL [26]). The embedding is somewhat similar to what we have done for the temporal logic RGITL [27]. In contrast to the latter, the embedding given is a semantic one (shallow embedding), that defines a semantic predicate $\mathsf{rel}(R)(\mathfrak{A}, \xi, \mathfrak{A}')$ only. Such an embedding skips the formalization of substitution and therefore does not allow to formalize Lemma 2. Adding a syntactic embedding (deep embedding) remains future work. The proofs have nevertheless been augmented compared to the ones that were done for [3] to include the least fixpoint theory needed for calls, the new dependency tracking and several other small improvements. The relational encoding has now been shown to be sound for all clash-free rules, not just for those that satisfy $\mathsf{cfc}(R)$. Again formally checking the theorems uncovered several mistakes in initial versions of the definitions. In particular the problem of Example 6 was found by careful analysis of failed proof attempts.

We also implemented a prototypical check in Scala, the programming language of KIV, that checks $\mathsf{cfc}(R)$ for the rules natively implemented in KIV. This is slightly simpler to implement than the check given here, since ASM rules in KIV's higher-order setting update function variables instead of dynamic functions. Therefore all updates in KIV modify the valuation ξ, and new function symbols are available without extending the signature. Extending the sequent calculus used in KIV with rules for **par** and **forall** similar to the ones given here so that KIV would support arbitrary parallel ASM rules and collecting experience with the proposed rules is future work though.

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

Acknowledgement We would like to thank the anonymous reviewers for their thorough analysis of the paper and valuable feedback. In particular, the questions they raised on the relational encoding have lead to a complete revision of Sec. 6.

References

[1] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, W. Reif, Development of a verified flash file system, in: Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ), Vol. 8477 of LNCS, Springer, 2014, pp. 9–24, invited Paper.

[2] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, W. Reif, KIV - Overview and VerifyThis Competition, Software Tools for Techn. Transfer 17 (6) (2015) 677–694.

[3] G. Schellhorn, G. Ernst, J. Pfähler, W. Reif, A Relational Encoding for a Clash-Free Subset of ASMs, in: Proceedings of ABZ 2016, Vol. 9675 of LNCS, Springer, 2016, pp. 237–243.

[4] R. F. Stärk, S. Nanchen, A complete logic for Abstract State Machines, Journal of Universal Computer Science (J.UCS) 7 (11) (2001) 981–1006.

[5] E. Börger, R. F. Stärk, Abstract State Machines—A Method for High-Level System Design and Analysis, Springer, 2003.

[6] R. F. Stärk, J. Schmid, E. Börger, Java and the Java Virtual Machine: Definition, Verification, Validation, Springer, 2001.

[7] E. Börger, A. Cisternino, V. Gervasi, Ambient Abstract State Machines with Applications, J. Comput. Syst. Sci. 78 (3).

[8] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific journal of mathematics 5 (1955) 285–309.

[9] T. Nipkow, Hoare Logics in Isabelle/HOL, Springer Netherlands, 2002, pp. 341–367.

[10] E. Börger, D. Rosenzweig, The WAM—definition and compiler correctness, in: Logic Programming: Formal Methods and Practical Applications, Studies in Computer Science and Artificial Intelligence 11, Elsevier, 1995, pp. 20–90.

[11] G. Schellhorn, W. Ahrendt, The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV, in: W. Bibel, P. Schmitt (Eds.), Automated Deduction — A Basis for Applications, Vol. III: Applications, Kluwer Academic Publishers, 1998, Ch. 3: Automated Theorem Proving in Software Engineering, pp. 165 – 194.

[12] J. Meng, L. C. Paulson, Translating higher-order clauses to first-order clauses, Journal of Automated Reasoning 40 (1) (2008) 35–60.

[13] W. de Roever, K. Engelhardt, Data Refinement: Model-Oriented Proof Methods and their Comparison, Vol. 47 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[14] F. Ferrarotti, K.-D. Schewe, L. Tec, Q. Wang, A logic for non-deterministic parallel Abstract State Machines, in: Proc. of FoIKS, Springer LNCS 9616, 2016, pp. 334–354.

[15] D. Harel, D. Kozen, J. Tiuryn, Dynamic Logic, MIT Press, 2000.

[16] F. Ferrarotti, K.-D. Schewe, L. Tec, Q. Wang, A complete logic for Database Abstract State Machines, Logic Journal of the IGPL.

[17] K.-D. Schewe, Q. Wang, A Customised ASM Thesis for Database Transformations, Acta Cybern. 19 (4) (2010) 765–805.

[18] G. Schellhorn, Verification of ASM Refinements Using Generalized Forward Simulation, Journal of Universal Computer Science (J.UCS) 7 (11) (2001) 952–979, URL: `http://www.jucs.org`.

[19] A. Blass, Y. Gurevich, Abstract state machines capture parallel algorithms, ACM Transactions on Computational Logic (TOCL) 4 (4) (2003) 578–651.

[20] A. Blass, Y. Gurevich, Abstract state machines capture parallel algorithms: Correction and extension, ACM Transactions on Computational Logic (TOCL) 9 (3) (2008) 19.

[21] F. Ferrarotti, K.-D. Schewe, L. Tec, Q. Wang, A new thesis concerning synchronised parallel computing–simplified parallel ASM thesis, Theoretical Computer Science 649 (2016) 25–53.

[22] E. Börger, K.-D. Schewe, Concurrent Abstract State Machines, Acta Informatica 53 (5) (2016) 469–492.

[23] K.-D. Schewe, F. Ferrarotti, L. Tec, Q. Wang, W. An, Evolving concurrent systems: behavioural theory and logic, in: Proceedings of the Australasian Computer Science Week Multiconference, ACM, 2017, p. 77.

[24] M. Leuschel, E. Börger, A compact ecoding of sequential ASMs in Event-B, in: Proc. ABZ of 2016, Springer LNCS, 2016, pp. 119–134.

[25] R. M. Burstall, Program proving as hand simulation with a little induction, Information Processing 74 (1974) 309–312.

[26] A relational encoding for a clash-free subset of ASMs: Formalization and proofs, `https://swt.informatik.uni-augsburg.de/swt/projects/Refinement/ASM-clashfree.html`.

[27] G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, W. Reif, RGITL: A temporal logic framework for compositional reasoning about interleaved programs, Annals of Mathematics and Artificial Intelligence (AMAI) 71 (2014) 131–174.