Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 214 (2008) 25–49
www.elsevier.com/locate/entcs

Completeness of ASM Refinement

Gerhard Schellhorn[1]

Lehrstuhl für Softwaretechnik und Programmiersprachen
Universität Augsburg
D-86135, Augsburg, Germany

Abstract

ASM refinements are verified using generalized forward simulations which allow to refine m abstract operations to n concrete operations with arbitrary m and n. One main difference to data refinement is that ASM refinement considers infinite runs and termination. Since backward simulation does not preserve termination in general, the standard technique of adding history information to the concrete level is not applicable. The powerset construction also adds infinite runs and is therefore not applicable too. This paper shows that a completeness proof is nevertheless possible by adding infinite prophecy information, effectively moving nondetermism to the initial state. Adding such prophecy information can be done on the semantic level, but also by a simple syntactic transformation that removes the choose construct of ASMs. The completeness proof is also ported to give a completeness proof for IO automata.

Keywords: refinement, completeness, abstract state machines, IO automata

1 Introduction

ASM refinement has been originally introduced by Börger ([4], [5]) into the framework of Gurevich's ASMs ([15], [8]). to solve problems in compiler verification. A formal definition has been given in [31] which originated from the formal correctness proofs ([28], [29], [30]) in the interactive theorem prover KIV [27] for a Prolog compiler [7]. A methodology for the general use of the ASM refinement paradigm has been defined and numerous applications are described in [6]. A comparison to data refinement has been given in [32].

A specific characteristic of ASM refinement is that it requires that infinite runs of the concrete system must have corresponding infinite runs on the

[1] schellhorn@informatik.uni-augsburg.de

doi:10.1016/j.entcs.2008.06.003

abstract level. The concrete system must always terminate if the abstract system does. For the use in compiler verification the requirement is easily motivated: assuming two ASMs are given that define an operational semantics for source code and compiled code, interpreting the compiled code should not lead to infinite runs, if interpreting the source code never does.

Other refinement definitions do not require that termination is preserved. An example for data refinement (for both the contract as well as the behavioral approach) is given in Section 5. Indeed, termination cannot be preserved in some applications, an example being refinement of a security protocols [34]. There an an attacker can cause infinite runs that contain rejected attacks only, which correspond to no abstract step. Therefore a variation of refinement correctness has been developed [33].

Refinement definitions for distributed systems, such as refinement of IO automata [24] typically also do not require that termination is preserved. Often only infinite runs are considered by adding stuttering steps [2]. Nevertheless these refinement definitions face a similar problem, since they have to preserve input and output done on infinite runs, which may be infinite.

Preserving infinite runs (or the input/output done on them) creates a problem for the completeness proof, since backward simulation is no longer correct in all cases. The current solutions either require finite (invisible) nondeterminism [2] or consider preserving finite input/output only ($\leq_{*\mathsf{T}}$ in [24]).

Backward simulation has therefore never been used in ASM refinement (variations of data refinement such as the one used in B [1] also do not use backward simulation).

Instead ASM refinement uses generalized forward simulations, which allow arbitrary commuting "m:n diagrams" which compare m steps of an abstract machine to n steps of the concrete machine. This generalizes forward simulations of data refinement, which allow 1:1 diagrams only. Generalizations of data refinement, e.g. weak refinement [10] and coupled refinement [14] have also suggested more general types of diagrams.

Although some examples which need backward simulation in data refinement can be verified using generalized forward simulation (in particular, the standard V vs. Y-shaped example), avoiding backward simulations seemed unavoidably to come at the price of incompleteness.

This paper shows that a completeness proof is possible anyway. The basic idea is to define a uniform transformation that moves all nondeterminism of the concrete system into the initial state. This is possible on the semantic level by constructing a prophecy automaton, but also on the syntactic level by using choice functions. The refinement between the resulting ASM and an abstract one then can always be verified using a forward simulation.

Before the completeness proof, a summary of the basics underlying ASM refinement is given (more details are in [31] and [32]). Transition systems which are the semantics of ASMs (but also of other operational formalisms) are defined in Section 2. To express proof obligations for syntactic ASM rules, KIV uses a higher-order variant of Dynamic Logic (or wp-calculus) described in Section 3. The definition of refinement correctness for ASMs is given in Section 4 together with an improved criterion for generalized forward simulation.

The completeness proof is given in Section 6. The proof is applicable to the original definition of ASM refinement, but also to the weaker variant of ASM refinement that preserves invariants given in [33]. Instead of giving details of this variation, Section 7 gives a completeness proof for (the stronger version $\leq_{\mathsf{T}}$ of) refinement of IO automata, which is closely related to ASM refinement preserving invariants.

After submitting a first version, a reviewer noted that the work of Wim Hesselink is quite similar to the completeness proof here. In Section 8 we give a first analysis of the relation between the approaches. Finally, Section 9 concludes.

All proofs in this paper have been mechanized with KIV and are available on the KIV web page [23].

2 Transition Systems and ASMs

The definition of ASM refinement is based on the following simple notion of a transition system:

Definition 2.1 (Transition System)
A transition system $\mathsf{M} = (\mathsf{S}, \mathsf{I}, \mathsf{F}, \mathsf{SEM})$ consists of a set S of states, subsets $\mathsf{I}, \mathsf{F} \subseteq \mathsf{S}$ of initial and final states, and a transition relation $\mathsf{SEM} \subseteq \mathsf{S}_\perp \times \mathsf{S}_\perp$ which is defined on $\mathsf{S}_\perp := \mathsf{S} \cup \{\perp\}$. The transition relation is required to be strict $(\perp, \perp) \in \mathsf{SEM}$, and total for non-final states $\mathsf{S}_\perp \setminus \mathsf{F} \subseteq \mathsf{dom}(\mathsf{SEM})$.

For ASMs, the transition relation SEM is given as the semantics $[\![\text{RULE}]\!]$ of an ASM rule. SEM distinguishes between possible and guaranteed nontermination, so it is an instance of erratic semantics. As an example, the semantics of **skip or abort**, where **skip** does nothing, **abort** always diverges, and **or** is nondeterministic choice is the relation $\{(\mathsf{s}, \mathsf{s}) : \mathsf{s} \in \mathsf{S}_\perp\} \cup \{(\mathsf{s}, \perp) : \mathsf{s} \in \mathsf{S}_\perp\}$.

Erratic semantics ([11]) can be defined using a variety of approaches and for a number of formalisms defining operations on a state. For abstract, sequential programs [25] defines the semantics by a pair of relations (r, e) where $\mathsf{r} \subseteq \mathsf{S} \times \mathsf{S}$ describes possible initial and final states, and $\mathsf{dom}(\mathsf{e})$ describes the domain of

guaranteed termination. From the two relations, erratic semantics can be defined as $\mathsf{SEM} = \mathsf{r} \cup \{(\mathsf{s}, \bot) : \mathsf{s} \in \mathsf{dom}(\mathsf{e})\}$. [26] gives a similar definition for commands c, the relation is $\mathsf{s} - \mathsf{c} \rightarrow \mathsf{s}'$, the domain of guaranteed termination is $\mathsf{c}\downarrow$. The variant of ASMs used in KIV is based on a similar definition, except that states include the valuation of higher-order variables which are used to represent the dynamic functions of ASMs. Another alternative is to define SEM based on predicate transformers (see e.g. [17]) and the resulting wp-calculus. The domain of guaranteed termination then consists of those states where $\mathsf{wp}(\alpha, \mathsf{true})$ holds, and the relation $(\mathsf{s}_0, \mathsf{s}_0') \in \mathsf{SEM}$ are those for which $\mathsf{s} = \mathsf{s}_0 \rightarrow \neg\ \mathsf{wlp}(\alpha, \mathsf{s} \neq \mathsf{s}_0')$ holds.

Blocking semantics of operations OP, defined by $\mathsf{OP} \cup (\mathsf{S}_\bot \setminus \mathsf{dom}(\mathsf{OP})) \times \{\bot\}$ as used by the behavioral approach to data refinement [3] can be viewed as an instance. Demonic semantics [13] as used by the contract approach [39] can be viewed as an instance too, using the embedding $\mathsf{OP} \cup (\mathsf{S} \setminus \mathsf{dom}(\mathsf{OP})) \times \mathsf{S}_\bot \cup \{(\bot, \bot)\}$. Although this is slightly different than the standard embedding which adds $\{\bot\} \times \mathsf{S}_\bot$ instead of just $\{\bot\} \times \{\bot\}$ to get the Smyth powerdomain [36], the difference is not relevant and gives the same conditions for refinement as shown in [32].

For the original ASMs defined in [15],[8] states are first-order algebras over some signature Σ: $\mathrm{S} = \mathrm{Alg}(\Sigma)$. Given a state s, the semantics of an ASM rule is defined by first calculating a set of updates $(\mathsf{f}, \underline{\mathsf{a}}, \mathsf{b})$ from the function updates $\mathsf{f}(\underline{\mathsf{t}}) := \mathsf{t}'$ of the ASM rule. For nondeterministic rules this calculation is nondeterministic, a formal definition using SOS rules is given in [8]. If the set contains two updates $(\mathsf{f}, \underline{\mathsf{a}}, \mathsf{b}_1)$ and $(\mathsf{f}, \underline{\mathsf{a}}, \mathsf{b}_2)$ it is inconsistent, and the rule application fails. In implementations this means that the rule will either stop with an error or diverge, and "divergence" and "failure" will be used synonymously in the following. In the semantics divergence results in $(\mathsf{s}, \bot) \in \mathsf{SEM}$. If the updates are consistent, they can be applied on s, modifying dynamic function f at $\underline{\mathsf{a}}$ to have the new value b. Applying all updates gives the new algebra s' with $(\mathsf{s}, \mathsf{s}') \in \mathsf{SEM}$.

ASMs often use total rules which never fail. For these the $\bot$ element is redundant. Deterministic ASMs are also an important special case. For this case, the logic of [37] uses the predicate $\neg\ \mathsf{def}(\mathsf{RULE})$ to characterize the domain of guaranteed termination.

For transition systems execution *traces* $\sigma = (\sigma_0, \sigma_1, \ldots)$ are finite or infinite sequences of elements $\sigma_\mathsf{i} \in \mathsf{S}_\bot$ that satisfy

$$\begin{aligned}\mathsf{trace}(\sigma) \equiv \quad & (\forall\ \mathsf{i}.\mathsf{i}+1 < \#\sigma \rightarrow \sigma_\mathsf{i} \notin \mathsf{F}_\bot \wedge (\sigma_\mathsf{i}, \sigma_{\mathsf{i}+1}) \in \mathsf{SEM}) \\ & \wedge\ (\#\sigma < \infty \rightarrow \mathsf{last}(\sigma) \in \mathsf{F}_\bot)\end{aligned}$$

The length of a trace σ written $\#\sigma \in \mathbb{N} \cup \{\infty\}$, $\mathsf{last}(\sigma)$ is the last element

of a finite trace. The definition implies that traces never pass through final states or $\bot$. Finite traces end with $\bot$ when the last rule applied fails, or with a final state.

A *run* of an ASM is a trace that starts with an initial state. To avoid having $\bot$ in runs, which will be necessary in the completeness proof, we define predicate $\mathsf{run}(\sigma)$ to hold for a sequence σ, if σ is a run of the ASM with a possible final $\bot$ element removed. This gives two disjoint classes of finite sequences for which $\mathsf{run}(\sigma)$ holds: either the last state is final and σ is a run, or the last state is non-final and extending it with $\bot$ gives a run.

The definition of refinement in Section 4 will use commuting diagrams which consist of several steps of two ASMs. To characterize these semantically two operators from temporal logic are needed. For $\mathsf{s} \in \mathsf{S}_\bot$ and $\mathsf{p}_\bot \subseteq \mathsf{S}_\bot$ $\mathsf{AF}(\mathsf{s}, \mathsf{p}_\bot)$ ("all executions starting with s will reach a state in $\mathsf{p}_\bot$") and $\mathsf{EF}(\mathsf{s}, \mathsf{p}_\bot)$ ("some execution starting with s eventually reaches a state in $\mathsf{p}_\bot$") are defined by

$$\begin{aligned}\mathsf{AF}(\mathsf{s}, \mathsf{p}_\bot) : &\leftrightarrow \forall\, \sigma.\ \sigma_0 = \mathsf{s} \wedge \mathsf{trace}(\sigma) \rightarrow \exists\, \mathsf{n}.\ \sigma_\mathsf{n} \in \mathsf{p}_\bot \\ \mathsf{EF}(\mathsf{s}, \mathsf{p}_\bot) : &\leftrightarrow \exists\, \sigma.\ \sigma_0 = \mathsf{s} \wedge \mathsf{trace}(\sigma) \wedge \exists\, \mathsf{n}.\ \sigma_\mathsf{n} \in \mathsf{p}_\bot\end{aligned}$$

Operators $\mathsf{AF}^+(\mathsf{s}, \mathsf{p}_\bot)$ and $\mathsf{EF}^+(\mathsf{s}, \mathsf{p}_\bot)$ assert that the number of steps must be positive.

$$\begin{aligned}\mathsf{AF}^+(\mathsf{s}, \mathsf{p}_\bot) : &\leftrightarrow \mathsf{s} \notin \mathsf{F}_\bot \wedge (\forall\, \mathsf{s}_0.\ \mathsf{SEM}(\mathsf{s}, \mathsf{s}_0) \rightarrow \mathsf{AF}(\mathsf{s}_0, \mathsf{p}_\bot)) \\ \mathsf{EF}^+(\mathsf{s}, \mathsf{p}_\bot) : &\leftrightarrow \mathsf{s} \notin \mathsf{F}_\bot \wedge (\exists\, \mathsf{s}_0.\ \mathsf{SEM}(\mathsf{s}, \mathsf{s}_0) \wedge \mathsf{EF}(\mathsf{s}_0, \mathsf{p}_\bot))\end{aligned}$$

3 Reasoning over ASMs in KIV's logic

To reason over transition systems in a theorem prover is possible on the semantic level, by specifying sets and relations using higher-order logic, and a first layer of the specifications in KIV uses such definitions. They are suitable for proving assertions about the refinement theory.

For case studies it is simpler to use a logic which allows to use the syntax of ASM rules directly. In KIV, higher-order wp-calculus [12] is used for this purpose, using notations from Dynamic Logic [22]. This section gives a characterization of the two main operators EF and AF in terms of this logic.

The logic of KIV allows to combine rules which are given operationally as abstract programs with predicate logic formulas. Three operators are defined:

- $[\alpha]\ \varphi$ means "all terminating runs of α end in a state where φ holds" and corresponds to $\mathsf{wlp}(\alpha, \varphi)$ in wp-calculus.

- $\langle\!|\alpha|\!\rangle\ \varphi$ means "all runs of α terminate and end in a state where φ holds" and corresponds to $\mathsf{wp}(\alpha, \varphi)$.
- $\langle\alpha\rangle\ \varphi$ means "there is a terminating run of α which ends in a state where φ holds" and is equivalent to $\neg\ \mathsf{wlp}(\alpha, \neg\ \varphi)$.

The logic of KIV does not talk explicitly about the $\bot$ element. Instead, given an ASM rule $\mathsf{RULE(s)}$ that modifies state $\mathsf{s} \in \mathsf{S}$ the fact that $\bot$ is a possible or the guaranteed outcome of $\mathsf{SEM} = [\![\mathsf{RULE}]\!]$ can be defined as

- $\mathsf{maydiverge(s)} := \neg\ \langle\!|\mathsf{RULE(s)}|\!\rangle\ \mathsf{true}$ ("the rule may fail in s")
- $\mathsf{diverges(s)} := \neg\ \langle\mathsf{RULE(s)}\rangle\ \mathsf{true}$ ("the rule surely fails in s")

The characterization of AF and EF then depends on whether $\bot \in \mathsf{p}_\bot$. In the negative case, given a predicate p with $[\![\mathsf{p}]\!] = \mathsf{p}_\bot$, a state $\mathsf{s} \in \mathsf{S}$ and a predicate final with $[\![\mathsf{final}]\!] = \mathsf{F}$ the operators needed are

$$\mathsf{AF_p(s,p)} \leftrightarrow \langle\!|\mathbf{while}\ \neg\ \mathsf{p(s)}\ \wedge \neg\ \mathsf{final(s)}\ \mathbf{do}\ \mathsf{RULE(s)}|\!\rangle\ \mathsf{p(s)}$$
$$\mathsf{EF_p(s,p)} \leftrightarrow \langle\mathbf{while}\ \neg\ \mathsf{p(s)} \wedge \neg\ \mathsf{final(s)}\ \mathbf{do}\ \mathsf{RULE(s)}\rangle\ \mathsf{p(s)}$$

In the positive case where $\bot \in \mathsf{p}_\bot$ and $[\![\mathsf{p}]\!] \cup \{\bot\} = \mathsf{p}_\bot$ the characterization is:

$$\mathsf{EF_p(s,p)} \leftrightarrow \langle\mathbf{while}\ \neg\ \mathsf{p(s)} \wedge \neg\ \mathsf{final(s)} \wedge \neg\ \mathsf{maydiverge(s)}\ \mathbf{do}\ \mathsf{RULE(s)}\rangle\ (\mathsf{p(s)} \vee \neg\ \mathsf{final(s)} \wedge \mathsf{maydiverge(s)})$$

$$\mathsf{AF_p(s,p)} \leftrightarrow \langle\!|\ \mathbf{while}\ \neg\ \mathsf{p(s)} \wedge \neg\ \mathsf{final(s)} \wedge \neg\ \mathsf{diverges(s)}\ \mathbf{do\ choose}\ \mathsf{s'}\ \mathbf{with}\ \langle\mathsf{RULE(s)}\rangle\ \mathsf{s} = \mathsf{s'}\ \mathbf{in}\ \mathsf{s} := \mathsf{s'}|\!\rangle\ (\mathsf{p(s)} \vee \neg\ \mathsf{final(s)} \wedge \mathsf{diverges(s)})$$

The loop characterizing $\mathsf{AF_p}$ is the most complex: it chooses steps of RULE(s) that do not fail, ignoring possible failures. This is done as long as the state is not final and as long as such steps exist (i.e. as long as $\neg$ diverges(s) holds). Finally, either p must hold, or the next step must definitely be a diverging step. Note that several of the formulas require to use a wp-formula as a test in an abstract program.

Given these syntactic characterizations, verification of refinement proof obligations can benefit from using ASM rules directly, which allows to exploit the control structure of rules to automate proofs by symbolic execution (see [27]).

4 ASM Refinement

This section gives a short summary of the definitions of [31] and [32] that characterize correctness of ASM refinement.

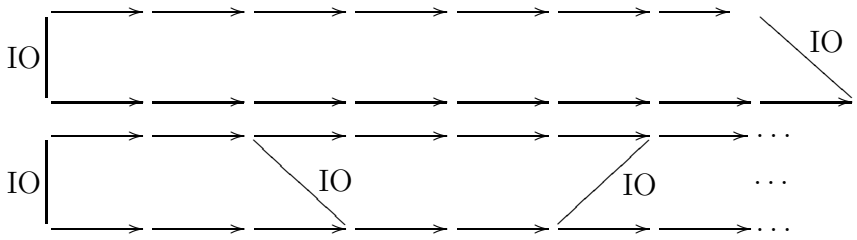


Fig. 1. Refinement correctness for finite and infinite runs

The definition refines an "abstract" transition system $\mathsf{AM} = (\mathsf{S_A}, \mathsf{I_A}, \mathsf{F_A}, \mathsf{SEM_A})$ to a "concrete" transition system $\mathsf{CM} = (\mathsf{S_C}, \mathsf{I_C}, \mathsf{F_C}, \mathsf{SEM_C})$. States and traces that belong to the abstract and concrete system will be denoted as as, σ_A and cs, σ_C respectively.

The basic idea of refinement is that runs σ_C of the concrete runs simulate abstract runs σ_A the abstract system. "Simulation" is given a rather liberal definition: it is not required that there is a 1:1 correspondence between steps nor that a specific visible component must be identical (although these are common cases). Instead a relation $\mathsf{IO} \subseteq \mathsf{S_A} \times \mathsf{S_C}$ is used [2] that determine when initial, final and intermediate states are considered to be similar. The definition is given based on the following criteria:

Definition 4.1 (Correctness of ASM Refinement)

- A refinement of AM to CM preserves partial correctness with respect to IO, if for every *finite* run σ_C of CM there exists a finite run σ_A such that initial states are related by IO and the final states are either related by IO too or both $\perp$.
- A refinement of AM to CM preserves total correctness if it preserves partial correctness an if in addition for every $\mathsf{infinite}$ σ_C run there exists an infinite abstract infinite run σ_A that starts in an initial state related by IO.
- A refinement is correct with respect to IO if it preserves total correctness and if for every infinite σ_C run there exists an infinite abstract run and two strictly monotone sequences $0 = \mathsf{i}_0 < \mathsf{i}_1 < \ldots$ and $0 = \mathsf{j}_0 < \mathsf{j}_1 < \ldots$ of natural numbers, such that for every k $\mathsf{IO}(\sigma_\mathsf{A}(\mathsf{i_k}), \sigma_\mathsf{C}(\mathsf{j_k}))$ holds.

Informally, an ASM refinement is correct, if finite runs implement finite runs and *infinite* refined runs pass through infinitely many corresponding states, see Fig. 1. States that correspond to each other via IO are often called *states of interest*, since these are regarded as observably equal.

To prove refinement correctness commuting diagrams are used like the ones

[2] the definition of ASM refinement in [32] had separate relations IR and OR for initial and final states and used IO for intermediate states only. The relation IO used here is their disjunction. Using one relation simplifies the formulas of the completeness proof.

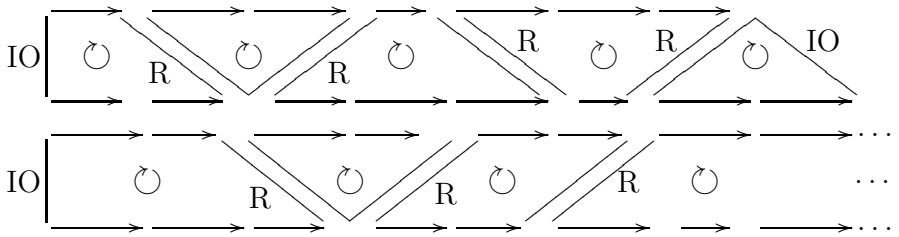


Fig. 2. Commuting diagrams to verify ASM refinement

shown in Figure 2 and a *coupling invariant* R between abstract and concrete states. In data refinement this relation is usually called a simulation. To preserve traces R must be stronger than IO: if e.g. IO states that both ASMs have done the same output, R may be a conjunct of IO and other properties necessary to prove invariance.

Unlike data refinement, where diagrams have to match one rule application (1:1 diagram), ASM refinement diagrams may have any shape. "m:n diagrams" are possible, where m abstract transitions match n concrete transitions. Even triangular shapes are allowed, but the case of an infinite consecutive sequence of 0:n diagrams must be prevented using some well-founded predicate $<_{0n}$ on pairs of states. Similarly, infinite sequences of m:0 diagrams must also be ruled out using $<_{m0}$ to preserve total correctness and refinement correctness.

The verification condition propagates the invariant forwards through the traces (so it is a forward or downward simulation). Since it does not preserve R in every step, R is called a "generalized forward simulation". The dual notion, generalized backwards simulation can be defined (some results are in [31]) but has rather weak properties in general. In particular it does not preserve infinite runs as required by ASM refinement correctness. Section 7 discusses that the situation is similar to IO automata where backward simulation is only allowed when infinite (invisible) nondeterminism is forbidden, which is neither an option for ASM refinement nor for data refinement, since the following ASM rule as well as the equivalent Z operation are perfectly legal (even if n is not considered by finalization).

OP
choose n' **with** $\mathsf{n}' > \mathsf{n}$
in $\mathsf{n} := \mathsf{n}'$

OP
$\Delta\ n : \mathbb{N}$
$n' > n$

Given total rules, for which runs ending in $\bot$ are absent, the proof obligation for a commuting diagram with R to be a generalized forward simulation is as follows:

$$
\begin{array}{ll}
 & \mathsf{R}(\mathsf{as},\mathsf{cs}) \wedge \neg\ (\mathsf{as} \in \mathsf{F_A} \wedge \mathsf{cs} \in \mathsf{F_C}) \\
\rightarrow & \quad \mathsf{EF}^+(\mathsf{as}, \lambda\, \mathsf{as}'.\mathsf{R}(\mathsf{as}',\mathsf{cs}) \wedge (\mathsf{as}',\mathsf{cs}) <_{\mathsf{m0}} (\mathsf{as},\mathsf{cs})) \\
 & \vee\ \mathsf{AF}^+(\mathsf{cs}, \lambda\, \mathsf{cs}'. \quad \mathsf{EF}^+(\mathsf{as}, \lambda\, \mathsf{as}'.\mathsf{R}(\mathsf{as}',\mathsf{cs}'))) \\
 & \qquad\qquad\qquad \vee\ \mathsf{R}(\mathsf{as},\mathsf{cs}') \wedge (\mathsf{as},\mathsf{cs}') <_{\mathsf{0n}} (\mathsf{as},\mathsf{cs}))
\end{array}
$$

Intuitively, the proof condition says: if states as and cs are related by R and not both final, then it must be possible to add a commuting diagram, such that R holds at the end. Either this diagram may consist of abstract steps only to form a triangular m:0 diagram (first disjunct, $<_{\mathsf{m0}}$ must decrease), or (second disjunct) it must finally be possible to complete a diagram for whatever concrete steps are chosen (the size of the diagram may depend on the choices). The number of abstract steps needed to complete the diagram may be positive, resulting in a m:n diagram where both $\mathsf{m}, \mathsf{n} > 0$, or it may be zero and $<_{\mathsf{0n}}$ must decrease.

For rules that may fail, the condition on commuting diagrams becomes significantly more complex, since in Logic we can talk about the $\bot$ element only implicitly using wp-calculus. Nevertheless it is possible to use the definitions of AF and EF of the previous section to derive a syntactic condition equivalent to (VC).

$$
\begin{array}{ll}
 & \mathsf{R}(\mathsf{as},\mathsf{cs}) \wedge \mathsf{as} = \mathsf{as}_0 \wedge \mathsf{cs} = \mathsf{cs}_0 \wedge \neg\ (\mathsf{final}(\mathsf{as}) \wedge \mathsf{final}(\mathsf{cs})) \\
\rightarrow & \quad \neg\ \mathsf{final}(\mathsf{as}) \wedge \langle \mathsf{ARULE}(\mathsf{as}) \rangle \\
 & \qquad\qquad\qquad \mathsf{EF_n}(\mathsf{cs}, \lambda\, \mathsf{cs}'.\mathsf{R}(\mathsf{as},\mathsf{cs}') \wedge (\mathsf{as},\mathsf{cs}') <_{\mathsf{m0}} (\mathsf{as},\mathsf{cs})) \\
 & \vee\ \textbf{if}\ \mathsf{EF_n}(\mathsf{as}, \lambda\, \mathsf{as}'.\neg\ \mathsf{final}(\mathsf{as}') \wedge \mathsf{maydiverge}(\mathsf{as}')) \\
 & \quad \textbf{then}\ \mathsf{AF_p}(\mathsf{cs}, \lambda\, \mathsf{cs}'.\neg\ \mathsf{final}(\mathsf{cs}') \wedge \mathsf{diverges}(\mathsf{cs}') \vee \varphi) \\
 & \quad \textbf{else}\ \mathsf{AF_n}(\mathsf{cs}, \lambda\, \mathsf{cs}'.\varphi)
\end{array}
$$

$$
\begin{array}{ll}
\text{where } \varphi = & \quad \mathsf{R}(\mathsf{as},\mathsf{cs}') \wedge (\mathsf{as},\mathsf{cs}') <_{\mathsf{0n}} (\mathsf{as},\mathsf{cs}) \\
 & \vee\ \neg\ \mathsf{final}(\mathsf{as}) \wedge \langle \mathsf{ARULE}(\mathsf{as}) \rangle\ \mathsf{EF_n}(\mathsf{as}', \mathsf{R}(\mathsf{as}',\mathsf{cs}'))
\end{array}
$$

(VC)

The condition given here improves on the sufficient condition given in [31] in that it is provably maximal. Figure 3 gives a pictorial description, which shows the four types of diagrams that are allowed by (VC).

States and relations after a "$\vdash$" symbol, as well as dotted lines must be shown to exist, assuming the rest of the diagram is given. Given two states as and cs, both not final, lines two and three of the condition allows diagrams of type (A). Line four checks whether AM has a run that ultimately applies a failing rule. If such a run exists, failing runs are allowed for CM: line five of the condition uses $\mathsf{AF_p}$ and allows diagrams of type (C). Otherwise all runs of AM must at some time reach a proper state ($\mathsf{AF_n}$) and be of one of the forms (B) or (D). These two cases are the two disjuncts of φ. With this verification condition for commuting diagrams correctness of generalized

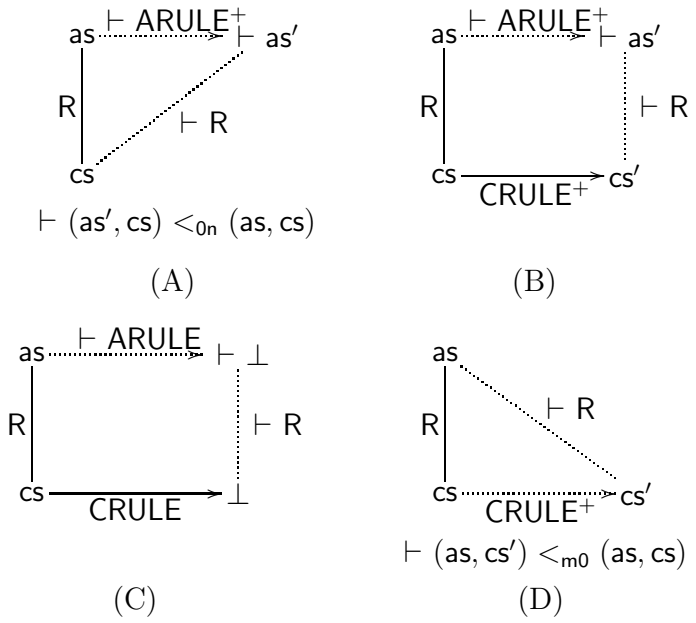


Fig. 3. Commuting diagrams in ASM refinement

forward simulation can be shown.

Theorem 4.2 (Generalized Forward Simulation)
A refinement from AM *to* CM *is correct if*

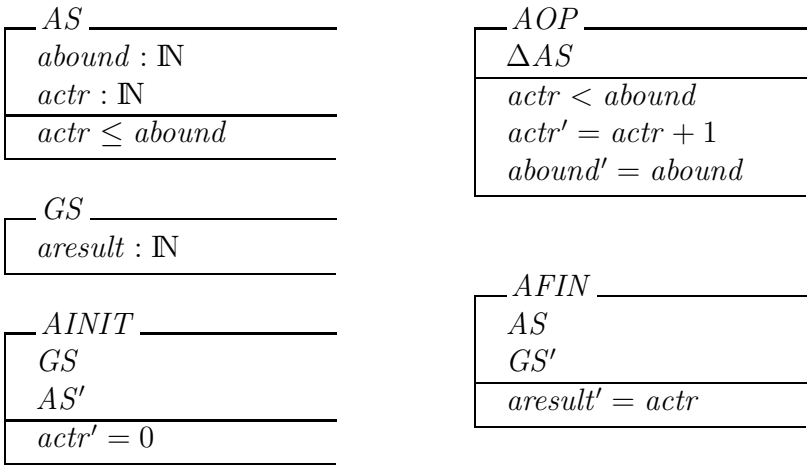
- $\forall$ cs.init(cs) $\rightarrow \exists$ as.init(as) $\wedge$ R(as, cs) *("initialization")*
- *verification condition (VC) holds ("correctness")*
- R(as, cs) $\rightarrow$ IO(as, cs) *("invariant implies IO")*

The "initialization" condition guarantees that every initial state cs has a corresponding initial state as with R(as, cs), assuming two predicates init(as) and init(cs) are given that specify the initial states I_A and I_C. The proof of the theorem in [31] intuitively follows the construction of commuting diagrams as shown in Figure 2.

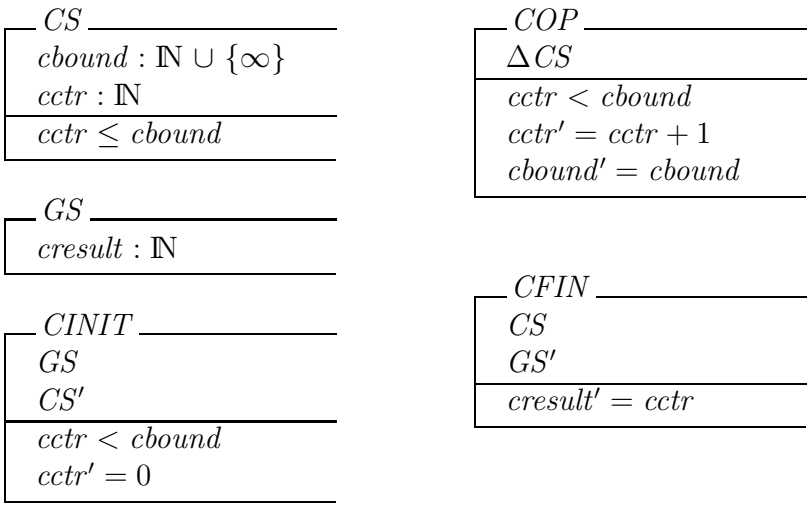
5 Data Refinement and Infinite Runs

In this section we give an example using Z [38] notation, that shows that in the presence of infinite nondeterminism data refinement allows to introduce infinite runs on the concrete level, when the abstract level has finite runs only.

The example defines an abstract data type ADT = (GS, AS, AINIT, AOP, AFIN) with a single operation as follows:



The abstract data type starts with $\mathsf{ctr} = 0$ and increments the counter up to a bound bound. Finalization extracts the current value of ctr. The state of the data type can therefore be described by pairs $(\mathsf{ctr}, \mathsf{bound})$, and its runs are $(0, \mathsf{n}), (1, \mathsf{n}) \ldots, (\mathsf{k}, \mathsf{n})$ for any $\mathsf{k} < \mathsf{n}$. The data type has infinite nondeterminism for choosing the initial value of bound. The concrete datatype has the same structure, except that it allows the bound to be infinity (∞, with the usual convention $\mathsf{n} < \infty$):



The concrete data type has the same runs as the abstract one, plus the additional infinite run $(0, \infty), (1, \infty), \ldots$. ASM refinement would reject this as a correct refinement, since the abstract data type has finite runs only, so

the concrete data type must not have infinite runs for a correct refinement.

Data Refinement considers finite runs only, the refinement is correct in the contract as well as in the behavioral approach. Both cases can be proved using a backward simulation, which just requires $\mathsf{cctr} = \mathsf{actr}$.

For the main commutativity condition a step of the concrete data type from $\mathsf{cs} = (\mathsf{m}^\infty, \mathsf{n})$ to $\mathsf{cs}' = (\mathsf{m}^\infty, \mathsf{n}+1)$ is given (with $\mathsf{m}^\infty \in \mathbb{N} \cup \{\infty\}$ and $\mathsf{n} < \mathsf{m}^\infty$) together with an abstract state $\mathsf{as}' = (\mathsf{m}, \mathsf{n}+1)$ related to cs' by the backward simulation. The abstract state needed to generate a commuting diagram then is $\mathsf{as} = (\mathsf{m}, \mathsf{n})$. All other refinement conditions (of the contract as well as the behavioral approach) trivially hold.

The example we give here is not new, similar examples are used in various disguises: in [24] to show that the two refinement notions for IO automata $\leq_T$ and $\leq_{*T}$ differ. In papers on semantics of programming languages, to show that predicate transformers are no longer continuous but just monotone [17], [13]. In refinement of TLA [2] to show that specifications need not have finite invisible nondeterminism.

6 The Completeness Proof

As was shown in the previous section, ASM refinement differs from data refinement in considering termination of all ASM runs from a specific initial state as an important property that should be preserved by refinement. As a consequence, general backward simulation is not acceptable for ASM refinement.

This makes the first of two standard constructions used for proving refinement completeness unavailable, which adds history information to the concrete data type and then proves that there is a backward simulation between the resulting data type and the abstract one. Examples of this construction are the proofs in Abadi and Lamport [2] and Lynch and Vaandrager for IO automata [24].

The second standard construction to prove completeness is the power set construction (used e.g. in [9]). It is not applicable either, since this construction may introduce infinite runs, when the original data type has none. To see this, consider again the abstract data type of the previous section, which has no infinite runs. The result of the power set construction would be a data type $(\mathsf{PAS}, \mathsf{PAINIT}, \mathsf{PAOP}, \mathsf{PAFIN})$. Its carrier set is the power set of abstract states: $\mathsf{PAS} = \mathcal{P}(\mathsf{AS})$. $\mathsf{PAINIT} = \{(\mathsf{n}, \{0\} \times \mathbb{N}) : \mathsf{n} \in \mathbb{N}\}$, and $\mathsf{PAOP} \subseteq \mathsf{PAS} \times \mathsf{PAS}$ is

$$\mathsf{PAOP} = \{(\mathsf{ctr}, \mathsf{bd}) : \mathsf{ctr} \leq \mathsf{bd}\} \times \{(\mathsf{ctr}+1, \mathsf{bd}), \mathsf{ctr}+1 \leq \mathsf{bd}\}$$

The power data type therefore has the infinite run

$$\{0\} \times \mathbb{N}, \{0\} \times (\mathbb{N} \setminus \{0\}), \{1\} \times (\mathbb{N} \setminus \{0,1\}), \{2\} \times (\mathbb{N} \setminus \{0,1,2\}), \ldots$$

Since both constructions are not applicable, it seemed for a long time impossible to find a completeness proof. The key idea to finding a solution anyway is to dualize the standard constructions of completeness proofs: both, adding the full history and the power set construction remove all nondeterminism from the past and move it into the future, enabling backward simulation. Instead moving all nondeterminism to the initial state enables a forward simulation. Moving the nondeterminism of a transition system M to the initial state is done by predicting the full run of the system. The resulting sytem $\mathsf{Det}(\mathsf{M})$ has deterministic steps only:

Definition 6.1 (Corresponding Deterministic Transition System) Given a transition system $\mathsf{M} = (\mathsf{S}, \mathsf{I}, \mathsf{F}, \mathsf{SEM})$ the corresponding deterministic transition system $\mathsf{Det}(\mathsf{M}) := (\mathsf{S}', \mathsf{I}', \mathsf{F}', \mathsf{SEM}')$ is defined as:

- $\mathsf{S}' := \mathsf{S}^+ \cup \mathsf{S}^\omega$, the set of all finite and infinite sequences of states
- I' consists of all σ with $\mathsf{run}(\sigma)$.
- $\mathsf{SEM}' := \{(\sigma, \mathsf{tail}(\sigma)) : \#\sigma > 1\} \cup \{((\mathsf{s}), \bot) : \mathsf{s} \notin \mathsf{F}\}$ where $\mathsf{tail}(\sigma)$ removes the first state from σ. The second set of the union adds transitions to $\bot$ iff the run ends with a failed rule application. Note that these final failing transitions were removed in the definition of run.
- $\mathsf{F}' := \{(\mathsf{s}) : \mathsf{s} \in \mathsf{F}\}$ consists of all sequences consisting of a single final state.

Theorem 6.2 (Equivalence of M and Det(M))

- $\mathsf{Det}(\mathsf{M})$ *is deterministic: each state has at most one successor state.*
- *For every run* σ *of* M $\mathsf{Det}(\mathsf{M})$ *has a run* $(\sigma, \mathsf{tail}(\sigma), \mathsf{tail}(\mathsf{tail}(\sigma)), \ldots)$ *of the same length. In particular the run is infinite iff* σ *is infinite.*
- *Every run of* $\mathsf{Det}(\mathsf{M})$ *has the form* $(\sigma, \mathsf{tail}(\sigma), \mathsf{tail}(\mathsf{tail}(\sigma)), \ldots)$ *for a run* σ *of* M.
- M *is a correct ASM refinement of* $\mathsf{Det}(\mathsf{M})$ *using* $\mathsf{IO}(\mathsf{s}, \sigma) \equiv \sigma = (\mathsf{s}, \ldots)$.

The proof is almost trivial by inspecting the definitions. Note that the relation IO is a backward simulation between M and $\mathsf{Det}(\mathsf{M})$. It is a harmless one, since it preserves infinite traces. It turns out that this is the only one that is needed: given a refinement from AM to CM it is always possible to define a generalized forward simulation between AM and $\mathsf{Det}(\mathsf{CM})$.

Theorem 6.3 (Completeness of ASM Refinement) *If* CM *is a correct ASM refinement of* AM *with respect to* IO*, then a generalized forward simula-*

tion exists between $\mathsf{Det}(\mathsf{CM})$ *and* AM.

The theorem follows directly from

Theorem 6.4 (Completeness for Deterministic Concrete Systems) *Given a correct refinement of* AM *to a deterministic transition system* CM *with respect to* IO*, then this refinement can always be proved using the following generalized forward simulation:*

$$
\begin{array}{l}
\mathsf{R}(\mathsf{as},\mathsf{cs}) \\
\equiv \exists\ \sigma_\mathsf{C}, \mathsf{n}, \sigma_\mathsf{A}, \mathsf{m}. \\
\qquad \mathsf{run}(\sigma_\mathsf{C}) \wedge \mathsf{run}(\sigma_\mathsf{A}) \wedge \mathsf{IO}(\sigma_\mathsf{A}[0], \sigma_\mathsf{C}[0]) \\
\quad \wedge\ \mathsf{m} < \#\sigma_\mathsf{C} \wedge \sigma_\mathsf{C}(\mathsf{m}) = \mathsf{cs} \wedge \mathsf{n} < \#\sigma_\mathsf{A} \wedge \sigma_\mathsf{A}(\mathsf{n}) = \mathsf{as} \\
\quad \wedge\ \textbf{if}\ \#\sigma_\mathsf{C} < \infty \\
\qquad \textbf{then}\quad \#\sigma_\mathsf{A} < \infty \\
\qquad\qquad \wedge\ \textbf{if}\ \mathsf{last}(\sigma_\mathsf{A}) \in \mathsf{F_C} \\
\qquad\qquad\quad \textbf{then}\quad \mathsf{last}(\sigma_\mathsf{A}) \in \mathsf{F_A} \wedge \mathsf{IO}(\mathsf{last}(\sigma_\mathsf{A}), \mathsf{last}(\sigma_\mathsf{C})) \\
\qquad\qquad\qquad\qquad \wedge\ (\mathsf{m} = \mathsf{n} = 0 \vee \mathsf{m}+1 = \#\sigma_\mathsf{C} \wedge \mathsf{n}+1 = \#\sigma_\mathsf{A}) \\
\qquad\qquad\quad \textbf{else}\ \mathsf{last}(\sigma_\mathsf{A}) \notin \mathsf{F_A} \wedge \mathsf{m} = \mathsf{n} = 0 \\
\qquad \textbf{else}\ \#\sigma_\mathsf{A} = \infty \wedge \exists\ (\mathsf{i_k})_{\mathsf{k}\in\mathbb{N}}, (\mathsf{j_k})_{\mathsf{k}\in\mathbb{N}}, \mathsf{k}. \\
\qquad\qquad\qquad\qquad \mathsf{i_k} = \mathsf{m} \wedge \mathsf{j_k} = \mathsf{n} \wedge \mathsf{mon}(\mathsf{i}) \wedge \mathsf{mon}(\mathsf{j}) \\
\qquad\qquad\qquad\quad \wedge\ \forall\ \mathsf{k}' \geq \mathsf{k}.\ \mathsf{IO}(\sigma_\mathsf{A}(\mathsf{i_{k'}}), \sigma_\mathsf{C}(\mathsf{i_{k'}}))
\end{array}
$$

The invariant basically states that as and cs are related if they are states of interest on two runs σ_A and σ_C which correspond via the original definition of refinement.

In more detail the formula requires two runs σ_A and σ_C whose initial states are related by IO. If σ_C is finite then σ_A must be finite too, and either both end in final states related by IO or both end with failed rule applications. In the first case as and cs being states of interest means that they are either both the initial ($\mathsf{m} = \mathsf{n} = 0$) or the final state ($\mathsf{m}+1 = \#\sigma_\mathsf{C}$ and $\mathsf{m}+1 = \#\sigma_\mathsf{A}$), in the second case both states must be initial.

If σ_C is infinite then σ_A must be infinite too, and the pair of states must be one of the pairs of states that correspond via IO. $\mathsf{mon}(\mathsf{j})$ and $\mathsf{mon}(\mathsf{i})$ abbreviate the conditions $0 = \mathsf{j}_0 < \mathsf{j}_1 < \ldots$ and $0 = \mathsf{i}_0 < \mathsf{i}_1 < \ldots$ respectively.

The proof that this is a generalized forward simulation first has to show that for a concrete state cs an abstract state with $\mathsf{R}(\mathsf{as},\mathsf{cs})$ exists. Now it is easy to prove that for any state cs a trace σ_C starting with this state exists (in this case even a unique one). The definition of refinement then guarantees the existence of a corresponding run σ_A. Choosing as to be the initial state of this run together with $\mathsf{m} = \mathsf{n} := 0$ it is easy to see that R is satisfied.

Second the correctness condition has to be shown: for two states with

$\mathsf{R}(\mathsf{as}, \mathsf{cs})$ and each trace starting with cs a commuting diagram must be attached, except if both states are final. Since CM is deterministic, there is only one such trace: the part of σ_C that starts with cs. Therefore the commuting diagram that has to be constructed is just the next commuting diagram that exists according to R: for two infinite traces the diagram ends with states $\sigma_\mathsf{A}(\mathsf{i}_{k+1})$ and $\sigma_\mathsf{C}(\mathsf{i}_{k+1})$. These states of course satisfy R again using the same traces σ_A and σ_C, and i_{k+1} j_{k+1} as values for m and n. For two finite traces the diagram ends with the two final states of σ_A and σ_C. If both states as and cs are not final, then both numbers of steps to the final states are positive, and the correctness condition is already satisfied. For the special case, where σ_A consists of of a single state which is both initial and final, while σ_C has $\mathsf{n} > 0$ states, a triangular diagram 0:n diagram results. The well-founded relation needed to allow one such diagram makes non-final states bigger than final ones. The case of an m:0 diagram is proved with a dual argument.

The semantic construction of moving all nondeterminism to the initial state is not really convenient for proving actual refinements since it involves recording all the details of future states in the initial state. In practice this is unnecessary. It is sufficient to record the outcomes of nondeterministic choices using additional dynamic functions. This can be done by a purely synactic transformation, since ASMs explicitly specify nondeterminism with a **choose** construct of the form

$$\mathsf{RULE} \equiv \textbf{choose } \mathsf{x} \textbf{ with } \varphi(\mathsf{x}, \underline{\mathsf{y}}) \textbf{ in } \mathsf{RULE}_1 \textbf{ ifnone } \mathsf{RULE}_2$$

The rule binds local variable x to some value that satisfies $\varphi(\mathsf{x}, \underline{\mathsf{y}})$ and executes RULE_1. If no suitable value exists, i.e. if $\exists\, \mathsf{x}.\varphi(\mathsf{x}, \underline{\mathsf{y}}))$ is false for the current values $\underline{\mathsf{y}}$, then RULE_2 is executed instead.

To remove nondeterminism a (static) choice function $\mathsf{choice}(\mathsf{n}, \underline{\mathsf{y}})$ is specified by the constraint

$$\forall\, \underline{\mathsf{y}}.(\exists\, \mathsf{x}.\varphi(\mathsf{x}, \underline{\mathsf{y}})) \rightarrow \varphi(\mathsf{choice}(\mathsf{n}, \underline{\mathsf{y}}), \underline{\mathsf{y}})$$

which guarantees that $\mathsf{choice}(\mathsf{n}, \underline{\mathsf{y}})$ returns a suitable value for each n that can be used as a value for x. The nondeterminism is then removed by transforming the rule to

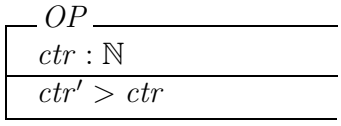
$\mathsf{Det}(\mathsf{RULE}) \equiv$
if $\exists\, \mathsf{x}.\varphi(\mathsf{x}, \underline{\mathsf{y}})$
then let $\mathsf{x} = \mathsf{choice}(\mathsf{n}, \underline{\mathsf{y}})$
  in $\mathsf{n} := \mathsf{n} + 1$ **seq**
    RULE_1
else RULE_2

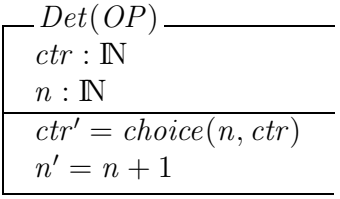
The new rule uses n as a counter that is incremented each time a choice is needed. This guarantees that all choices are independent and may yield different results, even when y always have the same values. **seq** is ASM notation for sequential composition.

It can be proved that for every run σ of the original ASM a corresponding run σ' of the modified ASM exists and vice versa. σ' starts in a state (algebra) that defines a suitable semantics of the choice function which predicts the choices made when running σ. Therefore σ and σ' agree on all program variables (i.e. dynamic functions) except that σ' adds the counter n. Repeatedly replacing all **choose** rules with choice functions gives an ASM which has the same finite *as well as infinite* runs as the original ASM.

The idea of removing nondeterminism from an ASM is not new: it was already described in [37], sketched in [8] (Remark 2.4.1 on p. 76) and we already used it in [32] to prove various equivalences between data types and ASMs.

The construction of moving nondeterminism to the initial state is also not limited to ASMs. For languages based on relational calculus the task is a little more complicated since one has to check for each variable, whether it is changed nondetermistically, and on which other variables the outcome depends, in order not to introduce unneccessary parameters for the choice function. As a simple example consider the translation of Z operation OP to the deterministic operation Det(OP). Since ctr is changed nondeterministically depending on the old value, a function choice(n, ctr) is needed that is specified by choice(n, ctr) > ctr.





7 Completeness of IO Automata Refinement

IO automata are a formalism that has similarities to ASMs. In this section we show, that the completeness proof for ASM refinement also applies to IO automata. We first give a short summary of the necessary definitions, following [24].

Definition 7.1 (IO Automata) An IO Automaton IOM = (S, I, A, SEM) consists of a set of states S, a subset of initial states I, a set of actions

A which always contains the empty action τ, and a transition relation $\mathsf{SEM} \subseteq \mathsf{S} \times \mathsf{A} \times \mathsf{S}$.

Definition 7.2 (Fragments, Executions and Action Traces)

- An *execution fragment* is a finite or infinite sequence $\sigma = (\mathsf{s}_0, \mathsf{a}_0, \mathsf{s}_1, \ldots)$, such that all $(\mathsf{s}_i, \mathsf{a}_{i+1}, \mathsf{s}_{i+1})$ are in SEM
- An *execution* is a fragment starting with an initial state.
- For an execution $\sigma = (\mathsf{s}_0, \mathsf{a}_0, \mathsf{s}_1, \ldots)$, it's *action trace* [3] $\mathsf{trace}(\sigma)$ is defined to be the sequence $(\mathsf{a}_0, \mathsf{a}_1, \ldots)$, but with all τ's removed.
- $\mathsf{frag}(\mathsf{IOM})$, $\mathsf{exec}(\mathsf{IOM})$ and $\mathsf{trace}(\mathsf{IOM})$ are the sets of all fragments, all executions and all action traces of IOM.

IO Automata can be easily translated to ASMs. A state of the corresponding ASM is composed of the automaton state s and the list al of all non-τ actions done so far. Final states $(\mathsf{s}, \mathsf{al})$ of the ASM are such that $(\mathsf{s}, \mathsf{al}) \in \mathsf{F} := \{\mathsf{s} : \neg\, \exists\, \mathsf{a}, \mathsf{s}'.(\mathsf{s}, \mathsf{a}, \mathsf{s}') \in \mathsf{SEM}\}$. The ASM rule for an IO automaton is:

choose a, s' **with** $(\mathsf{s}, \mathsf{a}, \mathsf{s}') \in \mathsf{SEM}$
in $\mathsf{s} := \mathsf{s}'$
  if $\mathsf{a} \neq \tau$ **then** $\mathsf{al} := (\mathsf{al}, \mathsf{a})$

Definition 7.3 (Refinement of IO Automata) An IO automaton $\mathsf{CIOM} = (\mathsf{S_C}, \mathsf{I_C}, \mathsf{A}, \mathsf{SEM_C})$ refines $\mathsf{AIOM} = (\mathsf{S_A}, \mathsf{I_A}, \mathsf{A}, \mathsf{SEM_A})$ written $\mathsf{CIOM} \leq_\mathsf{T} \mathsf{AIOM}$ iff $\mathsf{traces}(\mathsf{CIOM}) \subseteq \mathsf{traces}(\mathsf{AIOM})$.

The given definition is the stronger one of the two given in [24]. The weaker one, $\mathsf{CIOM} \leq_{*\mathsf{T}} \mathsf{AIOM}$, requires only that the *finite* action traces of CIOM are a subset of those of AIOM, which is a similar requirement than the one of data refinement. Completeness is proven for this weaker requirement in [24].

Refinement of IO automata is weaker than ASM refinement for two reasons. First, it ignores termination of runs: an IO automaton with a single one-step run $(\mathsf{s}_0, \mathsf{a}, \mathsf{s}_1)$ can be refined to one with one infinite run $(\mathsf{s}_0, \mathsf{a}, \mathsf{s}_1, \tau, \mathsf{s}_2, \tau, \mathsf{s}_3, \tau, \ldots)$ and vice versa. In general adding "stuttering" steps $(\mathsf{s}, \tau, \mathsf{s})$ for final states, where no step is applicable does not change the set of traces. Second, refinement of IO automata also allows to refine any run with a shorter run. E.g. $(\mathsf{s}_0, \mathsf{a}_0, \mathsf{s}_1, \mathsf{a}_1, \mathsf{s}_2, \mathsf{a}_2, \mathsf{s}_3)$ can be refined to $(\mathsf{s}_0, \mathsf{a}_0, \mathsf{s}_1)$.

By translating IO Automata to ASMs and choosing relation IO to be identity on the action list the completeness proof for ASMs can be translated to

[3] In [24] action traces are just called traces.

a proof purely in terms of IO automata. The proof constructs a deterministic automaton similar to ASMs.

Definition 7.4 (Corresponding Deterministic Automaton)
The corresponding deterministic automaton $\mathsf{Det}(\mathsf{IOM}) = (\mathsf{S}', \mathsf{I}', \mathsf{A}', \mathsf{SEM}')$ to $\mathsf{IOM} = (\mathsf{S}, \mathsf{I}, \mathsf{A}, \mathsf{SEM})$ consists of:

- $\mathsf{S}' := \mathsf{frag}(\mathsf{IOM})$
- $\mathsf{I}' := \mathsf{exec}(\mathsf{IOM})$
- $\mathsf{SEM}' = \{(\sigma, \mathsf{a}, \sigma') : \sigma = (\mathsf{s}, \mathsf{a}, \sigma')\}$

This construction is similar to the guess automaton ([24], p. 27), but it does not guess just finite executions, but any execution.

Theorem 7.5 (Equivalence of IOM and Det(IOM)) *IOM and* $\mathsf{Det}(\mathsf{IOM})$ *have the same action traces, both* $\mathsf{IOM} \leq_\mathsf{T} \mathsf{Det}(\mathsf{IOM})$ *and* $\mathsf{Det}(\mathsf{IOM}) \leq_\mathsf{T} \mathsf{IOM}$ *hold.*

Again the proof is immediate by comparing executions. The only run $\mathsf{Det}(\mathsf{IOM})$ has from an initial state $\sigma \in \mathsf{exec}(\mathsf{IOM})$ is the run executing the actions of σ, shortening the execution on every step. All nondeterminism has been moved to the initial state, $\mathsf{Det}(\mathsf{IOM})$ is *deterministic* [4].

This result is stronger than what can be proved using the standard backward simulation which maps traces to initial states. Backward simulation only implies $\mathsf{IOM} \leq_{*\mathsf{T}} \mathsf{Det}(\mathsf{IOM})$, the stronger result $\mathsf{IOM} \leq_\mathsf{T} \mathsf{Det}(\mathsf{IOM})$ does not follow from Proposition 2.5 of [24], since CIOM is not required to have finite invisible nondeterminism. It also does not follow from Theorem 3.17, since the backward simulation constructed is not image-finite in general.

$\mathsf{Det}(\mathsf{CIOM})$ is now linked to AIOM via a forward simulation. For IO automata the definition is

Definition 7.6 (Forward Simulation for IO automata) A relation $\mathsf{R} \subseteq \mathsf{S_A} \times \mathsf{S_C}$ between the states of two IO automata CIOM AIOM is a forward simulation if

- $\mathsf{I_C}(\mathsf{cs}) \rightarrow \exists\ \mathsf{as}.\mathsf{R}(\mathsf{as}, \mathsf{cs}) \wedge \mathsf{I_A}(\mathsf{as})$ ("initialization")
- $\mathsf{R}(\mathsf{as}, \mathsf{cs}) \wedge (\mathsf{cs}, \mathsf{a}, \mathsf{cs}') \in \mathsf{SEM}$
  $\rightarrow \exists\ \sigma = (\mathsf{as}, \mathsf{a_1}, \mathsf{as_1}, \ldots, \mathsf{as}') \in \mathsf{frag}(\mathsf{AIOM}) : \mathsf{trace}(\sigma) = (\mathsf{a}) \wedge \mathsf{R}(\mathsf{as}', \mathsf{cs}')$
  ("correctness")

$\mathsf{CIOM} \leq_\mathsf{F} \mathsf{AIOM}$ means that a forward simulation exists between the two automata. $\mathsf{CIOM} \leq_\mathsf{F} \mathsf{AIOM}$ implies $\mathsf{CIOM} \leq_\mathsf{T} \mathsf{AIOM}$ as shown in [24].

[4] note, that this definition of a deterministic IO automaton is different from the one of [24].

The correctness condition allows n:1 diagrams (with $\mathsf{n} \geq 0$). This is sufficient for ASM refinement too as shown in [31], although less flexible in applications. There is no criterion, that some order must be decreased in 0:1 diagrams, since preserving termination is not required.

Theorem 7.7 (Completeness of IO Automata Refinement) *Every correct IO automata refinement* $\mathsf{CIOM} \leq_\mathsf{T} \mathsf{AIOM}$ *can be verified by proving* $\mathsf{Det}(\mathsf{CIOM}) \leq_\mathsf{F} \mathsf{AIOM}$.

The theorem follows directly from

Theorem 7.8 (Completeness for Deterministic Concrete IO automata) *If* $\mathsf{CIOM} \leq_\mathsf{T} \mathsf{AIOM}$ *and* CIOM *is deterministic, then* $\mathsf{CIOM} \leq_\mathsf{F} \mathsf{AIOM}$

The forward simulation needed for the proof is

$$
\begin{aligned}
&\mathsf{R}(\mathsf{as}, \mathsf{cs}) \\
&\leftrightarrow \exists\, \sigma_\mathsf{C}, \sigma_\mathsf{A}, \mathsf{m}, \mathsf{n}. \\
&\qquad \sigma_\mathsf{A} \in \mathsf{exec}(\mathsf{AIOM}) \wedge \sigma_\mathsf{C} \in \mathsf{exec}(\mathsf{CIOM}) \\
&\qquad \wedge\ \mathsf{m} < \#\sigma_\mathsf{A} \wedge \mathsf{n} < \#\sigma_\mathsf{C} \\
&\qquad \wedge\ \mathsf{as} = \sigma_\mathsf{A}[\mathsf{m}] \wedge \mathsf{cs} = \sigma_\mathsf{C}[\mathsf{n}] \\
&\qquad \wedge\ (\#\sigma_\mathsf{C} < \infty \rightarrow \mathsf{last}(\sigma_\mathsf{C}) \in \mathsf{F}) \\
&\qquad \wedge\ \exists\, (\mathsf{i}_\mathsf{k})_{\mathsf{k} \in \mathbb{N}}.\quad \mathsf{i}_\mathsf{n} = \mathsf{m} \wedge \mathsf{monotone}(\mathsf{i}) \\
&\qquad\qquad\qquad \wedge\ \forall\, \mathsf{k}.\ \mathsf{n} \leq \mathsf{k} \wedge \mathsf{k} < \#\sigma_\mathsf{C} \\
&\qquad\qquad\qquad\qquad \rightarrow \mathsf{i}_\mathsf{k} < \#\sigma_\mathsf{A} \wedge \mathsf{trace}(\sigma_\mathsf{A}\ \mathsf{to}\ \mathsf{k}) = \mathsf{trace}(\sigma_\mathsf{A}\ \mathsf{to}\ \mathsf{i}_\mathsf{k})
\end{aligned}
$$

For $\sigma = ((\mathsf{s}_0, \mathsf{a}_1, \mathsf{s}_1, \ldots, \mathsf{s}_\mathsf{j})$ the definition uses $\#\sigma$ to denote the length j (which may be ∞). $\sigma[\mathsf{m}]$ is s_m and for $\mathsf{n} < \mathsf{j}$ σ to n denotes $(\mathsf{s}_0, \mathsf{a}_1, \mathsf{s}_1, \ldots, \mathsf{s}_\mathsf{n})$. For a finite execution $\mathsf{last}(\sigma) = \mathsf{s}_\mathsf{j}$ and $\mathsf{monotone}(\mathsf{i})$ means $\mathsf{i}_\mathsf{k} \leq \mathsf{i}_{\mathsf{k}+1}$ for all k.

To establishes the initialization condition determinism is exploited, which implies that for an initial state cs only one possible maximal (i.e. the last state is in F if the execution is finite) execution σ_C exists. Refinement implies that a corresponding trace σ_A can be constructed with the same action trace. Setting m and n both to be zero it finally remains to define the sequence (i_k) which defines corresponding initial segments of the executions with the same action traces. This is clearly possible since the full executions have the same action traces.

The proof of the correctness condition for forward simulation then is simple. Since the concrete system has at most one possible step it must be the step from $\sigma_\mathsf{C}[\mathsf{n}]$ to $\sigma_\mathsf{C}[\mathsf{n}+1]$. The corresponding fragment of σ_A then is the one from $\sigma_\mathsf{A}[\mathsf{i}_\mathsf{n}]$ to $\sigma_\mathsf{A}[\mathsf{i}_{\mathsf{n}+1}]$, therefore n, m after the step are chosen to be $\mathsf{n}+1$ and $\mathsf{i}_{\mathsf{n}+1}$. The sequence (i_k) can remain unchanged.

The theorem establishes, that IO automata refinements can be verified using forward simulation, provided one is willing to move nondeterminism to the initial state by using choice functions. The theorem is dual to Theorem 5.6 of [24], first proved in [35].

It should finally be mentioned, that adding choice functions is possible only in interactive verification, where additional functions can be added easily. It is not an option for finite-state systems when an automatic proof using model checking is intended, since the choice function determines infinitely many values. A similar disparity can be noted for the size of diagrams. [16] defines normed simulations as a variation of IO automata refinement, which use "small" 0:1, 1:1 and 1:0 diagrams only. For normed simulations, choosing the number of abstract steps in forward simulations is unnecessary, which simplifies model checking attempts. ASM refinement instead uses "big" diagrams, since these often define the natural correspondence between runs and often give simpler simulation relations for interactive verification.

8 Wim Hesselink's Work

A reviewer of this workshop made us aware that there is a series of papers by Wim Hesselink [18], [19], [20], [21] that give completeness proofs for refinement in the Abadi-Lamport setting [2] that have great similarity to this work.

Indeed, although our setting is technically different (ASMs have finite as well as infinite runs, failing rules are considered, but there is no built-in stuttering) there is a similarity between the eternity variables used in this setting and the choice functions used in ours.

Our impression is that the ASM refinement construction is simpler than the one given [21] , but also proves a weaker result. The main reason is that a system in the Abadi-Lamport setting does not only define a transition system M, but also a supplementary property P with liveness and fairness constraints. A run that satisfies the supplementary property is called a behavior. Abadi-Lamport refinement then requires to construct a corresponding abstract *behavior* for every concrete one, while ASM refinement just requires to constructs an abstract *run* for every concrete run.

The completeness proof becomes more complex, since the abstract run constructed by our forward simulations may not be a behavior. As an example the supplementary property could require that a certain state must be contained in the run. Even if this state is still reachable from every intermediate state of a run σ that is incrementally constructed by a forward simulation, the final run might avoid it, and therefore not be a behavior. To avoid this problem, forward simulations in the Abadi-Lamport setting require to verify an

additional global condition that ensures that all runs constructed by forward simulations are indeed behaviors.

In [21] the problem is solved by adding a clock and a universal eternity extension. Together these essentially construct an intermediate system with states (σ_C, n) where σ_C is a behavior of the concrete system, and n a natural number called a clock, that determines how many steps have been executed. Initial states have $n = 0$.

The intermediate state (σ_C, n) of this extension corresponds to our nth state $\mathsf{tail}^n(\sigma_C)$ of $\mathsf{Det}(\mathsf{CM})$. While our state guesses information about the future, (σ_C, n) also keeps full information about the past.

For the syntactic construction of removing **choose** by choice functions the correspondence becomes even closer: a transition system with an always terminating SEM relation would be encoded as the ASM rule

$$\textbf{choose } cs' \textbf{ with } \mathsf{SEM}(cs, cs') \textbf{ in } cs := cs'$$

Removing the global **choose** from this rule gives the rule

$$\textbf{let } cs' = \mathsf{choice}(cs, n) \textbf{ in } n := n + 1 \textbf{ seq } cs := cs'$$

with a static choice function. This ASM would just have to keep a copy of the initial state cs_0, to be able to inductively reconstruct all states of σ_C as $cs_{n+1} = \mathsf{choice}(cs_n, n)$.

Storing full information about the behavior σ_C in each state would likely be sufficient to allow a similar construction in the ASM setting than what is done in [21] for the Abadi-Lamport setting. There, a function c2a that maps every concrete behavior σ_C to a specific abstract behavior $\sigma_A := \mathsf{c2a}(\sigma_C)$ is defined a priori. Such a function exists, since refinement holds (using the axiom of choice). Instead of defining a forward simulation as in our construction the function is then used to define a refinement mapping f (i.e. a forward simulation that is a function) from the intermediate level to the abstract level as $f(\sigma_C, n) = \mathsf{c2a}(\sigma_C)[n]$. Note that f ensures that successive states of the concrete run are mapped to successive states of *just one abstract behavior* $\mathsf{c2a}(\sigma_C)$. Thereby the problem that the constructed abstract run might not be a behavior is avoided.

A small technical complication is, that in the Abadi-Lamport setting, a transition system without any additional liveness property can stutter infinitely instead of doing useful steps. For the intermediate system such behaviors $((\sigma_C, 0), (\sigma_C, 0), \ldots)$ must be ruled out by adding a supplementary property which requires that the clock is increased infinitely often.

Finally the different amounts of stuttering of the concrete and the abstract

system that are possible in a refinement in the Abadi-Lamport setting are taken into account in [21] by a construction called temporization, which allows to add a finite amount (say n) of stuttering steps in between the original steps of the concrete level. Together with stuttering forward simulations which allow a concrete step to implement m abstract steps, where all but one are stuttering steps, this seems to give a similar flexibility than our use of m:n diagrams.

Setting up a formal correspondence between the two formalisms remains future work. It seems it should be possible to use the idea of having function c2a in the ASM setting to get a completeness result for ASMs with additional properties. It should also be possible to define generalized forward simulations in the Abadi-Lamport setting. The definition could rely on accumulating observations, similar to the ASM for an IO automaton, that accumulates the actions done so far. An alternative would be to require at most one observable state change in every commuting diagram.

9 Conclusion

In this paper we have given a completeness proof for ASM refinement. The proof shows that generalized forward simulation alone is sufficient to prove correctness of any ASM refinement, provided that in some cases nondeterminism of the concrete ASM is moved to the initial state by adding choice functions that predict decisions taken during the run. The construction we have used is dual to well-known completeness proofs that add history information and prove the existence of a backward simulation, which are not applicable here, since termination must be preserved for ASM refinement.

We have also shown that a similar completeness result can be obtained for IO automata refinement. The result is obtained for both settings without an assumption of finite invisible nondeterminism.

Our preliminary analysis has shown several parallels to Wim Hesselink's completeness proof for refinement in the Abadi-Lamport setting. This setting and therefore the completeness proof are more complex than ours, since fairness and liveness constraints have to be considered. Nevertheless some of the steps in both completeness proofs are quite similar. An analysis of the formal correspondence still remains further work.

Whether a similar completeness result can be obtained for data refinement also remains as an open question. The answer is non-obvious, since a fully deterministic intermediate data type is not possible: any intermediate (conformal) data type will always have a nondeterministic choice *between* the operations.

Acknowledgement

I would like to thank Richard Banach for his remark "nondeterminism can be moved anywhere" which was the trigger for this work. I would also like to thank my colleagues Holger Grandy, Nina Moebius and Kurt Stenzel for carefully reading drafts of this paper. Thanks also to John Derrick, Heike Wehrheim and Eerke Boiten, who made me understand some of the intricate details of data refinement. Finally I would like to thank the reviewer who directed me to Wim Hesselink's work.

References

[1] J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 21, 2006.

[2] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 2:253–284, May 1991. Also appeared as SRC Research Report 29.

[3] C. Bolton, J. Davies, and J.C.P. Woodcock. On the refinement and simulation of data types and processes. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the International conference of Integrated Formal Methods (IFM)*, pages 273–292. Springer, 1999.

[4] E. Börger. A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 36–64. Springer, 1990.

[5] E. Börger. A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1–14. Springer, 1990.

[6] E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15 (1–2):237–257, November 2003.

[7] E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence 11, pages 20–90. North-Holland, Amsterdam, 1995.

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

[9] J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. FACIT. Springer, 2001.

[10] J. Derrick, E. A. Boiten, H. Bowman, and M. Steen. Weak Refinement in Z. In J.P. Bowen, M.G. Hinchey, and D.Till, editors, *ZUM '97: The Z Formal Specification Notation*, volume 1212, pages 369–388. Springer LNCS, 1997.

[11] J. deBakker. *Mathematical Theory of Program Correctness*. International Series in Computer Science. Prentice-Hall, 1980.

[12] E. W. Dijkstra. *A Discipline of Programming*, chapter 14. Prentice-Hall, Englewood Cliffs, N. J., 1976.

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

[14] J. Derrick and H. Wehrheim. Using Coupled Simulations in Non-atomic Refinement. In D. Bert, J. Bowen, S. King, and M. Walden, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651, pages 127–147. Springer LNCS, 2003.

[15] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford Univ. Press, 1995.

[16] W.O.D. Griffioen and F.W. Vaandrager. Normed simulations. In A.J. Hu and M.Y. Vardi (eds), editors, *Proceedings CAV'98*, volume 1427 of *LNCS*, pages 332–344, Vancouver, BC, Canada, 1998.

[17] W.H. Hesselink. *Programs, Recursion and Unbounded Choice*, volume 27 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[18] W. H. Hesselink. Eternity variables to simulate specifications. In J. van Leeuwen G. Good, J. Hartmanis, editor, *Mathematics of Program Construction MPC 02*, volume 2386 of *LNCS*, pages 117–130. Springer, 2002.

[19] W. H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Trans. on Computational Logic*, 6:175–201, 2005.

[20] W. H. Hesselink. Splitting forward simulations to cope with liveness. *Acta Informatica 42*, 42:583–602, 2006.

[21] W. H. Hesselink. Universal extensions to simulate specifications. *Information and Computation*, 206:106–128, 2008.

[22] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[23] Web presentation of KIV projects.
URL: http://www.informatik.uni-augsburg.de/swt/projects/.

[24] N. Lynch and F. Vaandrager. Forward and Backward Simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995. also: Technical Memo MIT/LCS/TM-486.b, Laboratory for Computer Science, MIT.

[25] R. D. Maddux. Relation-algebraic semantics. *Theoretical Computer Science*, 160(1–2):1–85, 1996.

[26] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.

[27] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II: Systems and Implementation Techniques, chapter 1: Interactive Theorem Proving, pages 13 – 39. Kluwer Academic Publishers, Dordrecht, 1998.

[28] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 3(4):377–413, 1997. URL: http://www.jucs.org.

[29] G. Schellhorn and W. Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume III: Applications, chapter 3: Automated Theorem Proving in Software Engineering, pages 165 – 194. Kluwer Academic Publishers, 1998.

[30] G. Schellhorn. *Verification of Abstract State Machines*. PhD thesis, Universität Ulm, Fakultät für Informatik, 1999. URL:
http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/.

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

[32] G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, vol. 336, no. 2-3:403–435, May 2005.

[33] G. Schellhorn. ASM Refinement Preserving Invariants. In *Proceedings of the 14th International ASM Workshop, ASM'07, Grimstad, Norway*, 2008. to appear in JUCS, http://ikt.hia.no/asm07/.

[34] G. Schellhorn, H. Grandy, D. Haneberg, N. Moebius, and W. Reif. A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In U. Glässer J.-R. Abrial, editor, *Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, volume 5115 of *LNCS*. Springer, 2008.

[35] A. P. Sistla. Proving correctness with respect to nondeterministic safety specifications. *Infrormation Processing Letters*, 39:45–49, 1991.

[36] M. Smyth. Power domains. *Journal of Computer and System Sciences*, 16:23–36, 1978.

[37] R. F. Stärk and S. Nanchen. A Complete Logic for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 7 (11):981 – 1006, 2001.

[38] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[39] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.