Completeness of fair ASM refinement

Gerhard Schellhorn *

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

A R T I C L E   I N F O

Article history:
Received 15 October 2008
Received in revised form 7 September 2009
Accepted 8 October 2009
Available online 18 October 2009

Keywords:
Refinement
Completeness
Abstract state machines
IO automata
Abadi–Lamport
Fairness

A B S T R A C T

ASM refinements are verified using generalized forward simulations which allow us to refine m abstract operations to n concrete operations with arbitrary m and n. One main difference from 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 to get a completeness proof. The power set construction also adds infinite runs and is therefore not applicable either. This paper shows that a completeness proof is nevertheless possible by adding infinite prophecy information, effectively moving nondeterminism to the initial state. Adding such prophecy information can be done not only on the semantic level, but also by a simple syntactic transformation that removes the choose construct of ASMs. The completeness proof is also translated to a completeness proof for IO automata. Finally, the proof is extended to deal with supplementary predicates, that specify fairness and liveness assumptions, by transferring a related result of Wim Hesselink for refinements that use the Abadi–Lamport setting.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

ASM refinement has been originally introduced by Börger [7,8] into the framework of Gurevich’s ASMs [19,11] to solve problems in compiler verification. A formal definition has been given in [36] which originated from the formal correctness proofs [33–35] in the interactive theorem prover KIV [32] for a Prolog compiler [10]. A methodology for the general use of the ASM refinement paradigm has been defined and numerous applications are described in [9]. A comparison to data refinement has been given in [37].

A specific characteristic of ASM refinement is that it requires that infinite runs of the concrete system must have corresponding infinite runs on the 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 that 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 be 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 protocol [39]. There 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 [38].

Refinement definitions for distributed systems, such as refinement of IO automata [29] typically also do not require that termination be 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 the input and output done on infinite runs, which may be an infinite sequence too.
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 (≤τ in [29]).

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 [14] and coupled refinement [18] 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. If the resulting deterministic ASM refines an abstract one, this can be always verified using a forward simulation.

Before the completeness proof, a summary of the basics underlying ASM refinement is given (more details are in [36] and [37]). 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 a criterion for generalized forward simulation that has been strengthened compared to [37].

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 [38]. Instead of giving details of this variation, Section 7 gives a completeness proof for (the stronger version ≤τ of) refinement of IO automata, which is closely related to ASM refinement preserving invariants.

Our proof is not the first to overcome the problem of combining infinite nondeterminism with infinite runs. For the Abadi–Lamport setting Wim Hesselink gives a completeness theorem in [25]. We summarize his result in Section 8, pointing at similarities and differences. His result deals with an important additional problem: fairness conditions. Section 9 transfers the result to the ASM setting, showing that in addition to using choice functions two other extensions are necessary: counting how many steps have been executed and keeping a copy of the initial state.

Finally, Section 10 concludes the article.

All proofs in this paper have been mechanized with KIV and are available on the KIV web page [28].

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 \( M = (S, I, F, SEM) \) consists of a set \( S \) of states, subsets \( I, F \subseteq S \) of initial and final states, and a transition relation \( SEM \subseteq S_+ \times S_+ \) which is defined on \( S_+: = S \cup \{ \bot \} \). The transition relation is required to be strict, i.e. \((\bot, s) \in SEM \iff s = \bot\), and total for non-final states \( S_+ \setminus F \subseteq \text{dom}(SEM) \).

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 \( \{(s, s) : s \in S_+\} \cup \{(s, \bot) : s \in S_+\} \).

Erratic semantics [15] can be defined using a variety of approaches and for a number of formalisms defining operations on a state. For abstract, sequential programs [30] defines the semantics by a pair of relations \((r, e)\) where \( r \subseteq S \times S \) describes possible initial and final states, and \( dom(e) \) describes the domain of guaranteed termination. From the two relations, erratic semantics can be defined as \( SEM = r \cup \{(s, \bot) : s \in dom(e)\} \). [31] gives a similar definition for commands \( c \), the relation is \( s \rightarrow c \rightarrow s' \), the domain of guaranteed termination is \( c \downarrow \). 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. [21]) and the resulting wp-calculus. The domain of guaranteed termination then consists of those states where wp\((\alpha, \text{true})\) holds, and the relation \((s_0, s_0) \in SEM \) are those for which \( s = s_0 \rightarrow \neg \text{wp}(\alpha, s \neq s_0) \) holds.

Blocking semantics of operations OP, defined by \( OP \cup (S_+ \setminus \text{dom}(OP)) \times \{\bot\} \) as used by the behavioral approach to data refinement [5] can be viewed as an instance. Demonic semantics [17] as used by the contract approach [44] can be viewed as an instance too, using the embedding \( OP \cup (S \setminus \text{dom}(OP)) \times S_+ \cup \{(\bot, \bot)\} \). Although this is slightly different than the standard embedding which adds \( \{(\bot) \times S_+ \) instead of just \( \{\bot\} \) to get the Smyth power domain [41], the difference is not relevant and gives the same conditions for refinement as shown in [37].

For the original ASMs defined in [19] and [11], states are first-order algebras over some signature \( \Sigma: S = \text{Alg}(\Sigma) \). An ASM rule is built up from function updates \( f(t) : = t' \), that are combined using parallel composition (by writing one rule above
another), sequential composition (written \texttt{seq}), conditionals (\texttt{if}) and local variables (\texttt{let}). (Sub-)Rules can be named and defined recursively. Nondeterminism is expressed by

\begin{verbatim}
choose x with \varphi in RULE1, ifone RULE2
\end{verbatim}

The rule binds local variable \(x\) to some value that satisfies \(\varphi\) and executes RULE1. If no suitable value exists, i.e. if \(\exists x. \varphi\) is false, then RULE2 is executed instead. By default, a missing \texttt{ifone} defaults to \texttt{ifone abort}, where \texttt{abort} is the rule that always fails. Nondeterministic choice (RULE1 or RULE2) can be defined as an abbreviation using \texttt{choose}.

Given a state \(s\), the semantics of an ASM rule is defined by first calculating a set of updates \((f, a, b)\) from the function updates \(f(t) := t'\), where \(a\) and \(b\) are the semantics of \(t\) and \(t'\) in \(s\). For nondeterministic rules this calculation is nondeterministic, a formal definition using SOS rules is given in \cite{11}. If the set contains two updates \((f, a, b_1)\) and \((f, a, b_2)\) with \(b_1 \neq b_2\) it is inconsistent, and rule application fails, i.e. \((s, \bot) \in \text{SEM}\). Otherwise the next state is computed by applying all updates; \((f, a, b)\) modifies function \(f\) at a to become \(b\) in the new state \(s'\). In an implementations a failing rule application will either stop with an error or diverge, and “divergence” and “failure” will be used synonymously in the following.

ASMs often use total rules which never fail. For these the \(\bot\) element is redundant. Deterministic ASMs which choose an initial state and then execute a deterministic rule are also an important special case. For this case, the logic of \cite{42} uses the predicate \(\neg \text{def}(\text{RULE})\) to characterize the domain of guaranteed termination.

We have followed \cite{11} to have one ASM rule in every ASM. ASMs with several rules are common too. Such ASMs typically use one of the conventions “all applicable rules fire simultaneously” or “nondeterministic choice between the rules”. Since both conventions are expressible (using \texttt{choose} and parallel composition) they are semantically equivalent to an ASM with a single rule.

For transition systems execution \(\text{traces } \sigma = (\sigma(0), \sigma(1), \ldots)\) are finite \((\in S^*_{\bot})\) or infinite \((\in S^\omega_{\bot})\) sequences of elements \(\sigma(i) \in S_{\bot}\) that satisfy

\[
\text{trace}(\sigma) := (\forall i, i + 1 < \#\sigma \rightarrow \sigma(i) \notin F_{\bot} \land (\sigma(i), \sigma(i + 1)) \in \text{SEM}) \\
\land (\#\sigma < \infty \rightarrow \text{last}(\sigma) \in F_{\bot})
\]

The length of a trace \(\sigma\) written \(\#\sigma \in \mathbb{N} \cup \{\infty\}\), \(\text{last}(\sigma)\) 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 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, we define predicate \(\text{run}(\sigma)\) to hold for a sequence \(\sigma\), if \(\sigma\) is a run of the ASM with a possible final \(\bot\) element removed. This gives two disjoint classes of finite sequences for which \(\text{run}(\sigma)\) holds: either the last state is final and \(\sigma\) 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 \(s \in S_{\bot}\) and \(q \subseteq S_{\bot}\), \(\text{AF}(s, q)\) (“all executions starting with \(s\) will reach a state in \(q\)”) and \(\text{EF}(s, q)\) (“some execution starting with \(s\) eventually reaches a state in \(q\)”) are defined by

\[
\text{AF}(s, q) := \forall \sigma. \sigma(0) = s \land \text{trace}(\sigma) \rightarrow \exists n < \#\sigma. \sigma(n) \in q \\
\text{EF}(s, q) := \exists \sigma. \sigma(0) = s \land \text{trace}(\sigma) \land \exists n < \#\sigma. \sigma(n) \in q
\]

Operators \(\text{AF}^+(s, q)\) and \(\text{EF}^+(s, q)\) assert that the number of steps must be positive.

\[
\text{AF}^+(s, q) := s \notin F_{\bot} \land (\forall s_0. \text{SEM}(s, s_0) \rightarrow \text{AF}(s_0, q)) \\
\text{EF}^+(s, q) := s \notin F_{\bot} \land (\exists s_0. \text{SEM}(s, s_0) \land \text{EF}(s_0, q))
\]

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 that uses the syntax of ASM rules directly. In KIV a higher-order variant of wp-calculus \cite{16} is used for this purpose, using notations from Dynamic Logic \cite{26}. This section gives a characterization of the two main operators \(\text{EF}\) and \(\text{AF}\) 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:

- \(\preceq\) \(\varphi\) means “all terminating runs of \(\alpha\) end in a state where \(\varphi\) holds” and corresponds to \(\text{wp}(\alpha, \varphi)\) in wp-calculus.
- \(\langle\alpha\rangle\) \(\varphi\) means “all runs of \(\alpha\) terminate and end in a state where \(\varphi\) holds” and corresponds to \(\text{wp}(\alpha, \varphi)\).
- \(\langle\alpha\rangle\) \(\varphi\) means “there is a terminating run of \(\alpha\) which ends in a state where \(\varphi\) holds” and is the same as \(\neg\text{wp}(\alpha, \neg\varphi)\).

\footnote{The usual precedence \(\neg > \land > \lor > \rightarrow > \leftrightarrow\) for conjunctors is used in all formulas.}
The logic of KIV does not explicitly use a ∐ element. Instead, given an ASM rule \texttt{RULE} that modifies state \( s \in S \) (often written \texttt{RULE}(s) in the following\footnote{In KIV, a named rule (procedure) \texttt{RULE} is used that has state \( s \) as reference parameter. In case studies, \( s \) is instantiated to a vector of parameters and \texttt{RULE} gets an implementation.}) the fact that \( \bot \) is a possible or the guaranteed outcome of \( \texttt{SEM} = \llbracket \texttt{RULE} \rrbracket \) can be defined as

\begin{itemize}
  \item \texttt{maydiverge(s)} := \neg \llbracket \texttt{RULE(s)} \rrbracket \texttt{true} ("the rule may fail in \( s \)"
  \item \texttt{diverges(s)} := \neg \llbracket \texttt{RULE(s)} \rrbracket \texttt{true} ("the rule surely fails in \( s \)"
\end{itemize}

The characterization of \texttt{AF} and \texttt{EF} then depends on whether \( \bot \in q \). In the negative case, given a predicate \( p \) with \( \llbracket p \rrbracket = q \), a state \( s \in S \) and a predicate final with \( \llbracket \texttt{final} \rrbracket = F \) the operators needed are

\[ \texttt{AF}_p(s, p) \leftrightarrow \llbracket \texttt{while } \neg p(s) \land \neg \texttt{final(s) do RULE(s)} \rrbracket p(s) \]
\[ \texttt{EF}_p(s, p) \leftrightarrow \llbracket \texttt{while } \neg p(s) \land \neg \texttt{final(s) do RULE(s)} \rrbracket p(s) \]

In the positive case where \( \bot \in q \) and \( \llbracket p \rrbracket \cup \{ \bot \} = q \) the characterization is:

\[ \texttt{EF}_p(s, p) \leftrightarrow \llbracket \texttt{while } \neg p(s) \land \neg \texttt{final(s) do RULE(s)} \rrbracket (p(s) \lor \neg \texttt{final(s) \land maydiverge(s)}) \]
\[ \texttt{AF}_p(s, p) \leftrightarrow \llbracket \texttt{while } \neg p(s) \land \neg \texttt{final(s) do RULE(s)} \rrbracket \texttt{true} \]
\[ \texttt{do choose } s' \texttt{ with } \llbracket \texttt{RULE(s)} \rrbracket s = s' \]
\[ \texttt{in } s := s' \rrbracket (p(s) \lor \neg \texttt{final(s) \land diverges(s)}) \]

The loop characterizing \texttt{AF}_p is the most complex: it chooses steps of \texttt{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 \( \neg \texttt{diverges(s)} \) 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 \( \texttt{wp} \)-formula as a test in an abstract program.

Given these syntactic characterizations, verification of refinement proof obligations can benefit from using ASM rules directly, by exploiting the control structure of rules to automate proofs by symbolic execution (see [32]).

4. ASM refinement

This section gives a short summary of the definitions of [36] and [37] that characterize correctness of ASM refinement.

The definition refines an "abstract" transition system \( \texttt{AM} = (S_{\textit{A}}, \lambda_{\textit{A}}, F_{\textit{A}}, \texttt{SEM}_{\textit{A}}) \) to a "concrete" transition system \( \texttt{CM} = (S_{\textit{C}}, l_{\textit{C}}, F_{\textit{C}}, \texttt{SEM}_{\textit{C}}) \). States and traces that belong to the abstract and concrete system will be denoted as \( \sigma_{\textit{A}} \) and \( s_{\textit{C}} \), respectively.

The basic idea of refinement is that runs \( \sigma_{\textit{C}} \) of the concrete runs simulate abstract runs \( \sigma_{\textit{A}} \) 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. Instead an arbitrary relation \( \texttt{IO} \subseteq S_{\textit{A}} \times S_{\textit{C}} \) is used\footnote{The definition of ASM refinement in [37] had separate relations \( \texttt{IR} \) and \( \texttt{OR} \) for initial and final states and used \( \texttt{IO} \) for intermediate states only. The relation \( \texttt{IO} \) used here is their disjunction, since in all case studies \( \texttt{IR} \) (and similarly \( \texttt{OR} \)) was \( \texttt{IO} \cap (l_{\textit{A}} \times l_{\textit{C}}) \). Using one relation simplifies the formulas of the completeness proof.} that determine when states are considered to be similar. The encodings of \( \texttt{IO} \) automata and Abadi–Lamport machines as ASMs shown in Sections 7 and 8 will specialize \( \texttt{IO} \) to equality of non-\( \tau \) actions and to equality of observations.

The definition of refinement here is given based on runs related by \( \texttt{IO} \).

**Definition 4.1** (Runs Related by \( \texttt{IO} \)).

Two runs \( \sigma_{\textit{C}} \) of \( \texttt{CM} \) and \( \sigma_{\textit{A}} \) of \( \texttt{AM} \) are related by \( \texttt{IO} \subseteq S_{\textit{A}} \times S_{\textit{C}} \), if either both are finite and their initial and final states are either related by \( \texttt{IO} \) or both \( \bot \) (final states only). Otherwise both must be infinite and there must be two monotonic sequences \( 0 = i_0 < i_1 < \ldots \) and \( 0 = j_0 < j_1 < \ldots \) of natural numbers, such that \( R(\sigma_{\textit{A}}(i_k), \sigma_{\textit{C}}(i_k)) \) holds for all \( k \).

Two finite and two infinite runs related by \( \texttt{IO} \) are shown in Fig. 1. For refinement every concrete run must be related to an abstract run.

**Definition 4.2** (Correctness of ASM Refinement). A refinement is correct with respect to \( \texttt{IO} \) if for every run \( \sigma_{\textit{C}} \) a run \( \sigma_{\textit{A}} \) related by \( \texttt{IO} \) exists.

Informally, an ASM refinement is correct, if finite runs implement finite runs and if infinite refined runs pass through infinitely many corresponding states. It is easy to see that a correct refinement preserves partial and total correctness assertions modulo \( \texttt{IO} \) (see [37] for details). States that correspond to each other via \( \texttt{IO} \) are often called states of interest, since these are regarded as observably equal.

To prove refinement correctness commuting diagrams are used like the ones shown in Fig. 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 \( \texttt{IO} \): i.e. \( \texttt{IO} \) states that both ASMs have done the same output, \( R \) may be a conjunction of \( \texttt{IO} \) and other properties necessary to prove invariance.
generalized forw ardsimulation as follows:

Z operation are perfectly legal.

which is neither an option for ASM refinement nor for data refinement, since the following ASM rule as well as the equivalent

is simil ar to IO automata where backwardsimulation is only allowed when infinite (invisible) nondeterminism is forbidden,

particular it does not preserve infinite runs as required by ASM refinement correctness.

Section 7

Fig. 1. Refinement correctness for finite and infinite runs.

Fig. 2. Commuting diagrams to verify ASM refinement.

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 relation \(<_{0n}\) 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 backwardsimulation can be defined (some results are in [36]) 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.

\[
\begin{align*}
\text{OP} &\quad \text{choose } n' \text{ with } n' > n \\
\text{in } n := n' &\quad \text{OP} \\
\Delta n: \mathbb{N} &\quad n' > n
\end{align*}
\]

Given total rules, for which runs ending in \(\perp\) are absent, the proof obligation for a commuting diagram with R to be a

generalized forward simulation is as follows:

\[
\begin{align*}
R(as, cs) &\land \neg (as \in F_A \land cs \in F_C) \\
\rightarrow &\quad \text{EF}^+(as, \lambda as', R(as', cs)) \land (as', cs) <_{m0} (as, cs) \\
&\lor \text{AF}^+(as, \lambda cs', R(as', cs')) \\
&\lor R(as, cs') \land (as, cs') <_{0n} (as, cs)
\end{align*}
\]

(1)

The formula overloads the temporal operators. EF\(^+(as, \ldots)\) refers to a positive number of executions of ARULE, while the
definition of EF\(^+(cs, \ldots)\) uses CRULE. 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, \(<_{m0}\) 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 m, n > 0, or it may be zero and \(<_{0n}\) must decrease. Note that the case where one of as and cs is final and the

other is not is allowed by the second precondition. The ASM with the non-final state must then do additional steps to finalize.
For rules that may fail, the condition on commuting diagrams becomes significantly more complex, since in wp-calculus the $\bot$ element is implicit in the semantics. Nevertheless it is possible to use the definitions of AF and EF of the previous section to derive a syntactic condition equivalent to the proof obligation above.

$$R(as, cs) \land \text{as} = as_0 \land cs = cs_0 \land \neg (\text{final}(as) \land \text{final}(cs)) \rightarrow \neg \text{final}(as) \land \langle \text{ARULE}(as) \rangle$$

$$\lor \text{if } EF_n(as, \lambda as'. R(as', cs) \land (as', cs) <_{m_0} (as_0, cs)) \rightarrow \text{maydiverge}(as')$$

$$\text{then } \langle \text{CRULE}(cs) \rangle \text{ AF}_p(cs, \lambda cs'. \neg \text{final}(cs') \land \text{diverges}(cs') \lor \varphi)$$

$$\text{else } \langle \text{CRULE}(cs) \rangle \text{ AF}_n(cs, \lambda cs'. \varphi)$$

where $\varphi = R(as, cs') \land (as, cs') <_{on} (as, cs_0)$

$$\lor \neg \text{final}(as) \land \langle \text{ARULE}(as) \rangle EF_n(as, \lambda as'. R(as', cs'))$$

The condition given here improves on the sufficient condition given in [36] in that it is provably maximal. Note that the free variables $as_0$ and $cs_0$ are used to hold the initial values of $as$ and $cs$. Saving these is necessary since the formulas after $\langle \text{ARULE}(as) \rangle$, $\langle \text{CRULE}(cs) \rangle$ and $\langle \text{CRULE}(cs) \rangle$ refer to modified values of $as$ and $cs$, but the well-founded relations needs the initial ones.

Fig. 3 gives a pictorial description, which shows the four types of diagrams that are allowed by (VC). States and relations after a “$\bot$” 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 may apply a failing rule. If such a run does not exist (the else case), at least one non-failing rule of CM is executed, and all runs of the ASM must reach a proper state ($\text{AF}_n$) where a diagram of the form (B) or (D) commutes. These two cases are given by the two disjuncts of $\varphi$. Otherwise, if AM has a failing run, in addition to (B) and (D) diagrams of type (C) are possible. This is described by the then case in line five. Such diagrams start with a rule application of CM, that may fail ($\langle \text{CSTEP}(cs) \rangle$). If it succeeds, all runs from the resulting state that do not fail ($\text{AF}_n$) must either reach a definitely failing state ($\neg \text{final}(cs') \land \text{diverges}(cs')$), or allow a diagram of type (B) or (D) (formula $\varphi$ again). With this verification condition for commuting diagrams correctness of generalized forward simulation can be shown.

**Theorem 4.3** (Generalized Forward Simulation).

A refinement from AM to CM is correct if

- $\forall cs. \text{init}(cs) \rightarrow \exists as. \text{init}(as) \land R(as, cs)$ (“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 $\text{init}(as)$ and $\text{init}(cs)$ are given that specify the initial states $I_A$ and $I_C$. The proof of the theorem in [36] intuitively follows the construction of commuting diagrams as shown in Fig. 2.
5. Data refinement and infinite runs

In this section we give an example using $Z[43]$ notation, that shows that in the presence of infinite nondeterminism data refinement allows one 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:

**AS**

- `abound : N`
- `actr : N`
- `actr \leq abound`

**GS**

- `arestalt : N`

**AINIT**

- `GS`
- `AS`
- `AINIT` : `actr = 0`

**AOP**

- `ΔAS`
- `actr < abound`
- `actr' = actr + 1`
- `abound' = abound`

**AFIN**

- `AS`
- `GS'`
- `arestalt' = actr`

The abstract data type starts with `actr = 0` and increments the counter up to a bound `abound`. Finalization extracts the current value of `actr`. The state of the data type can therefore be described by pairs `(actr, abound)`, and its runs are $(0, m), (1, m), \ldots, (k, m)$ for any $k < m$. The data type has infinite nondeterminism for choosing the initial value of `abound`. The concrete data type $ADT = (GS, CS, CINIT, COP, CFIN)$ has the same structure, except that it allows the bound to be infinity ($\infty$, with the usual convention $n < \infty$):

**CS**

- `cbound : N \cup \{\infty\}`
- `cctr : N`
- `cctr \leq cbound`

**GS**

- `cresult : N`

**CINIT**

- `GS`
- `CS`
- `cresult' = 0`

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 `cctr = actr`.

For the main commutativity condition a step of the concrete data type from $cs = (n, m^\infty)$ to $cs' = (n + 1, m^\infty)$ is given (with $m^\infty \in N \cup \{\infty\}$ and $n \leq m^\infty$) together with an abstract state $as'$ related to $cs'$ by the backward simulation, which implies $cs = (n + 1, m')$. The abstract state needed to generate a commuting diagram then is $as = (n, m')$. All other refinement conditions (of the contract [44] as well as the behavioral [5] approach) trivially hold.

The example we give here is not new, similar examples are used in various disguises: in [29] to show that the two refinement notions for IO automata $\leq_T$ and $\leq_{\ast T}$ differ; in papers on semantics of programming languages, to show that predicate transformers are no longer continuous but just monotone [21,17]; 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 [29].

The second standard construction to prove completeness is the power set construction (used e.g. in [13]). 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 (PAS, PAINIT, PAOP, PAFIN). Its carrier set is the power set of abstract states: PAS = \( \{ (S) \mid S \in \mathrm{S} \} \), PAINIT = \( \{ (n, (0) \times \mathbb{N}) : n \in \mathbb{N} \} \), and PAOP \( \subseteq \) PAS \( \times \) PAS is

\[
\text{PAOP} = \{ (\text{actr}, \text{bd}) : \text{actr} \leq \text{bd} \} \times \{ (\text{actr} + 1, \text{bd}), \text{actr} + 1 \leq \text{bd} \}
\]

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 system Det(M) has deterministic steps only:

**Definition 6.1 (Corresponding Deterministic Transition System).**

Given a transition system \( M = (S, I, F, \text{SEM}) \) the corresponding deterministic transition system Det(M) := \( (S', I', F', \text{SEM}') \) is defined as:

- \( S' := S^+ \cup S^\omega \), the set of all finite and infinite sequences of states
- \( I' \) consists of all \( \sigma \) with run(\( \sigma \)).
- \( \text{SEM}' := \{ (\sigma, \text{tail}(\sigma)) : \#\sigma > 1 \} \cup \{ ((s), \bot) : s \notin F \} \) where tail(\( \sigma \)) removes the first state from \( \sigma \). 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.
- \( F' := \{ (s) : s \in F \} \) consists of all sequences consisting of a single final state.

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

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

The proof is almost trivial by inspecting the definitions. The construction is uniform, so there is no need to prove equivalence of M and Det(M) in every case study. As will be discussed below, the construction is also possible on the syntactic level.

Note that the relation IO is a backward simulation between M and Det(M). It is a harmless one, since it preserves infinite traces. Nevertheless, in general it is not an (infinitely often [27]) image-finite relation, which would guarantee that infinite traces are preserved (as shown in [29] and in [12]).

It turns out that the construction of Det(M) together with forward simulations is already sufficient for completeness: given a refinement from AM to CM it is always possible to define a generalized forward simulation between Det(CM) and AM.

**Theorem 6.3 (Completeness of ASM Refinement).**

If CM is a correct ASM refinement of AM with respect to IO, then a generalized forward simulation exists between Det(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:
The invariant states that $as$ and $cs$ are related if they are states of interest on two runs $as$ and $cs$ which correspond via the original definition of refinement. In more detail the formula requires two runs $as$ and $cs$ whose initial states are related by IO. If $as$ is finite then $as$ 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 ($m = n = 0$) or the final state ($m + 1 = \#as$ and $n + 1 = \#cs$), in the second case both states must be infinite. If $as$ is infinite then $as$ must be infinite too, and the pair of states must be one of the pairs of states that correspond via IO. $mon(i)$ and $mon(j)$ abbreviate the conditions $0 = j_0 < j_1 < \ldots$ and $0 = i_k < i_1 < \ldots$ respectively.

**Proof.** The proof that $R$ is a generalized forward simulation has to show the three conditions of Theorem 4.3. For the first “initialization” condition, for every concrete state $cs$ an abstract state has to be defined, such that $R(as, cs)$ holds. Now it is easy to prove that for any state $cs$ a trace $as$ starting with this state exists (in this case even a unique one). The definition of refinement then guarantees the existence of a corresponding run $as$. Choosing as to be the initial state of this run together with $m = n = 0$ it is easy to see that $R$ is satisfied.

Second the “correctness” condition has to be shown: for two states with $R(as, cs)$ 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 $as$ 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 $as(i_{k+1})$ and $cs(i_{k+1})$. These states of course satisfy $R$ again using the same traces $as$ and $cs$, and $i_{k+1}$ as values for $m$ and $n$. For two finite traces the diagram starts with $as$ and $cs$ and ends with the two final states of $as$ and $cs$. If both states as and cs are final, then this is an $m:n$ diagram with $m > 0$ and $n > 0$, and the correctness condition is already satisfied. For the special case, where $as$ consists of a single state which is both initial and final, while $cs$ has $n > 0$ 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.

Third, to complete the proof, it remains to show that $R(as, cs)$ implies $IO(as, cs)$ (condition “invariant implies IO”). This is obvious, since $R$ relates states of interest only, which are related by IO by the definition of refinement. □

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 syntactic transformation, since ASMs explicitly specify nondeterminism with choose. To remove a nondeterministic choose of the form

$$\text{RULE} \ := \ \text{choose} \ x \ \text{with} \ \varphi(x, y) \ \text{in} \ \text{RULE}, \ \text{if} \ n \ \text{in} \ \text{RULE}_2$$

where formula $\varphi$ has free variables $x$ and $y$, a (static) choice function $\text{choice}(n, y)$ is axiomatized with the constraint

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

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

$$\text{Det(RULE)} \ := \ \text{if} \ \exists x. \varphi(x, y) \ \text{then let} \ x := \text{choice}(n, y) \ \text{in} \ n := n + 1 \ \text{seq} \ \text{RULE}_1 \ \text{else} \ \text{RULE}_2$$

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 variables $y$ always have the same values. seq is ASM notation for sequential composition.
It can be proved that for every run $\sigma$ of the original ASM a corresponding run $\sigma'$ of the modified ASM exists and vice versa. $\sigma'$ starts in a state (algebra) that defines a suitable semantics of the choice function which predicts the choices made when running $\sigma$. Therefore $\sigma$ and $\sigma'$ agree on all program variables (i.e. dynamic functions) except that $\sigma'$ 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 [42], sketched in [11] (Remark 2.4.1 on p. 76) and we already used it in [37] 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 nondeterministically, and on which other variables the outcome depends, in order not to introduce unnecessary parameters for the choice function. As a simple example consider the translation of a $Z$ 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 $choice(n, ctr) > ctr$.

\[
\begin{array}{c|c}
\text{OP} & \text{Det}(OP) \\
\hline
ctr : \mathbb{N} & n : \mathbb{N} \\
ctr > ctr & n' = n + 1 \\
\end{array}
\]

### 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 [29].

**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 $\tau$, and a transition relation $SEM \subseteq S \times A \times S$.

The set $F$ of final states can be defined as the set of states that satisfy $\neg \exists a, s'. (s, a, s') \in SEM$.

**Definition 7.2 (Fragments, Executions and Action Traces).**

- An execution fragment is a finite or infinite sequence $\sigma = (s_0, a_0, s_1, \ldots)$, such that all $(s_i, a_{i+1}, s_{i+1})$ are in $SEM$.
- An execution is a fragment starting with an initial state.
- For an execution $\sigma = (s_0, a_0, s_1, \ldots)$, its action trace\(^4\) $trace(\sigma)$ is defined to be the sequence $(a_0, a_1, \ldots)$, but with all $\tau$'s removed.
- $frag(IOM)$, $exec(IOM)$ and $trace(IOM)$ 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-$\tau$ actions done so far. The ASM rule for an IO automaton is:

\[
\text{choose } a, s' \text{ with } (s, a, s') \in SEM \\
\text{in } s := s' \\
\text{if } a \neq \tau \text{ then } al := al, a
\]

**Definition 7.3 (Refinement of IO Automata).**

An IO automaton $CIOM = (S_c, I_c, A, SEM_c)$ refines $AIOM = (S_a, I_a, A, SEM_a)$ (written $CIOM \leq_T AIOM$) iff $trace(CIOM) \subseteq trace(AIOM)$.

The given definition is the stronger one of the two given in [29]. The weaker one, $CIOM \leq_{\tau T} AIOM$, requires only that the finite action traces of $CIOM$ are a subset of those of $AIOM$. This is a similar requirement as in data refinement. Completeness is proven for this weaker requirement in [29].

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 $(s_0, a, s_1)$ can be refined to one with one infinite run $(s_0, a, s_1, \tau, s_2, \tau, \ldots)$ and vice versa. In general adding “stuttering” steps $(s, \tau, s)$ for final states, where no step is applicable does not change the set of traces. Second, IO automata also allow one to refine any run with a shorter run. E.g. $(s_0, a_0, s_1, a_1, s_2, a_2, s_3)$ can be refined to $(s_0, a_0, s_1)$.

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

\(^4\) In [29] action traces are just called traces.
Definition 7.4 (Corresponding Deterministic Automaton). The corresponding deterministic automaton $\text{Det}(\text{IOM}) = (S', I', A', \text{SEM}')$ to $\text{IOM} = (S, I, A, \text{SEM})$ consists of:

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

This construction is similar to the guess automaton ([29], p. 27), but it does not guess just finite executions, but any execution.

Theorem 7.5 (Equivalence of IOM and Det(IOM)).

$IOM$ and $\text{Det}(\text{IOM})$ have the same action traces, both $\text{IOM} \leq_T \text{Det}(\text{IOM})$ and $\text{Det}(\text{IOM}) \leq_T \text{IOM}$ hold.

Again the proof is immediate by comparing executions. The only run $\text{Det}(\text{IOM})$ has from an initial state $\sigma \in \text{exec}(\text{IOM})$ is the run executing the actions of $\sigma$, shortening the execution on every step. All nondeterminism has been moved to the initial state, $\text{Det}(\text{IOM})$ is deterministic.\(^5\)

This result is stronger than what can be proved using the standard backward simulation which maps traces to initial states. Backward simulation only implies $\text{IOM} \leq_T \text{Det}(\text{IOM})$, the stronger result $\text{IOM} \leq_T \text{Det}(\text{IOM})$ does not follow from Proposition 2.5 of [29], 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.

$\text{Det}(\text{CIOM})$ 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 $R \subseteq S_A \times S_C$ between the states of two IO automata CIOM and AIOM is a forward simulation if

- $I_C(cs) \rightarrow \exists \text{ as. } R(\text{as, cs}) \wedge I_A(\text{as})$ ("initialization")
- $R(\text{as, cs}) \wedge (\text{cs, a, cs}' \in \text{SEM})$
  $\rightarrow \exists \sigma = (\text{as}, a_1, a_2, \ldots, a_s) \in \text{frag}(\text{AIOM}) : \text{trace}(\sigma) = (a) \wedge R(\text{as}', \text{cs}')$

("correctness")

$\text{CIOM} \leq_F \text{AIOM}$ means that a forward simulation exists between the two automata. CIOM $\leq_F$ AIOM implies CIOM $\leq_T$ AIOM as shown in [29].

The correctness condition allows $n:1$ diagrams (with $n \geq 0$). This is sufficient for ASM refinement too as shown in [36], 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 $\text{CIOM} \leq_T \text{AIOM}$ can be verified by proving $\text{Det}(\text{CIOM}) \leq_F \text{AIOM}$.

The theorem follows directly from

Theorem 7.8 (Completeness for Deterministic Concrete IO automata). If $\text{CIOM} \leq_T \text{AIOM}$ and CIOM is deterministic, then $\text{CIOM} \leq_F \text{AIOM}$.

The forward simulation needed for the proof is

$R(\text{as, cs})$

$\leftrightarrow \exists \sigma_C, \sigma_A, m, n.$

$\sigma_A \in \text{exec}(\text{AIOM}) \land \sigma_C \in \text{exec}(\text{CIOM})$

$\land m < #\sigma_A \land n < #\sigma_C$

$\land \text{ as = } \sigma_A[m] \land \text{ cs = } \sigma_C[n]$

$\land (\#\sigma_C < \infty \rightarrow \text{last}(\sigma_C) \in F )$

$\land \exists (i_p)_{p \in \mathbb{N}}. i_n = m \land \text{monotone}(i)$

$\land \forall k. n \leq k \land k < #\sigma_C$

$\rightarrow i_k < #\sigma_A \land \text{trace}(\sigma_A \text{ to } k) = \text{trace}(\sigma_A \text{ to } i_k)$

For $\sigma = (s_0, a_1, s_1, \ldots, s_j)$ the definition uses $\#\sigma$ to denote the length $j$ (which may be $\infty$). $\sigma[m]$ is $s_m$ and for $n < j$, $\sigma$ to $n$ denotes $(s_0, a_1, a_2, \ldots, s_n)$. For a finite execution $\text{last}(\sigma) = s_j$ and $\text{monotone}(i)$ means $i_k \leq i_{k+1}$ for all $k$.

To establish 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 $\sigma_C$ exists. Refinement implies that a corresponding trace $\sigma_A$ 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 $\sigma_C[n]$ to $\sigma_C[n+1]$. The corresponding fragment of $\sigma_A$ then is the one from $\sigma_A[i_n]$ to $\sigma_A[i_{n+1}]$, therefore $n, m$ after the step are chosen to be $n+1$ and $i_{n+1}$. The sequence ($i_k$) can remain unchanged.

---

\(^5\) Note, that this definition of a deterministic IO automaton is different from the one of [29].
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 [29], first proved in [40].

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. [20] 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. Completeness of Abadi–Lamport refinement

Wim Hesselink has published a series of papers [22–25] that give completeness proofs for refinement in the Abadi–Lamport setting. The papers step by step get rid of the assumptions (finite nondeterminism, machine closedness and internal continuity) that were used in the original completeness proof in [2] for the Abadi–Lamport setting of refinement. In this section we summarize the main results of [25], pointing out similarities and differences to our work. The following section will transfer the result to a completeness result for ASM refinement with additional fairness conditions.

The Abadi–Lamport setting is based on the following definition of transition systems.

**Definition 8.1 (Transition systems of the Abadi–Lamport setting).**

An Abadi–Lamport transition system $\text{ALM} = (S, I, \text{SEM}, \text{Sup}, O, \text{Obs})$ (often called a machine) consists of

- a set $S$ of states
- a subset $I \subseteq S$ of initial states
- a reflexive transition relation $\text{SEM} \subseteq S \times S$
- a supplementary property $\text{Sup} \subseteq S^\omega$
- a set of observations $O$
- an observation function $\text{Obs} \subseteq S \times O$ which defines the observable (or external) part of the state. $O$ is often specified to be a subsequence of the variables that form the full state.

The main distinguishing feature of the formalism is the reflexive transition relation, which always includes so-called stutter steps. The transition relation is typically given as a predicate logic formula between unprimed and primed variables. When constructing a transition relation, typically only the non-stuttering steps are given, and we write 'plus stutter steps' to indicate that identity has to be added.

As a consequence of stuttering, all runs are necessarily infinite (no final states). Also there are no failing steps. Failing rules and termination would have to be added explicitly using explicit error states and a "has terminated" flag (states which have this flag set, should then allow stutter steps only).

Another consequence of stuttering is, that runs are not obliged to any non-stuttering steps. Therefore liveness conditions are necessary to ensure that a machine executes any non-stuttering steps at all. This is done using the supplementary predicate $\text{Sup}$, which constrains runs of the machine. The predicate is typically given using temporal logic: e.g. $\square \diamond s \neq s'$ would specify that live runs of the machine must have infinitely many non-stuttering steps.

The supplementary predicate is also used to specify fairness constraints. If the transition relation consists e.g. of a nondeterministic choice between two operations, the fairness constraint could specify that each of them must be executed infinitely often, if it is infinitely often enabled.

To be consistent with the idea of stuttering, the supplementary predicate has to be a property: adding or removing stuttering must not change its truth.

The semantics of the machine therefore consists of those runs which satisfy the supplementary property. These are called behaviors. If $\sigma$ is one behavior of $\text{ALM}$, then adding or removing stutter steps will always give another behavior.

Refinement is based on the observable part of behaviors: $\text{Obs}(\sigma)$ is defined to be the result of applying $\text{Obs}$ point-wise to the states of a behavior $\sigma$.

**Definition 8.2 (Refinement in the Abadi–Lamport setting).**

A concrete machine $\text{CALM} = (S_C, I_C, \text{SEM}_C, \text{Sup}_C, O_C, \text{Obs}_C)$ refines an abstract machine $\text{AALM} = (S_A, I_A, \text{SEM}_A, \text{Sup}_A, O_A, \text{Obs}_A)$ if

- both machines have the same observations $O_A = O_C$
- for each behavior $\sigma_C$ of the concrete machine there is an abstract behavior $\sigma_A$ machine, such that $\text{Obs}_C(\sigma_C)$ and $\text{Obs}_A(\sigma_A)$ are equal modulo adding and removing stutter steps.

Abadi–Lamport refinement allows different speeds of the machines by allowing different amounts of stuttering. This roughly corresponds to the freedom of having $m:n$ diagrams in ASM refinement. Compared to the relation IO used in ASM Refinement the definition is specialized to equality of observations ($\text{IO}(\text{as}, \text{cs}) :\leftrightarrow \text{Obs}_A(\text{as}) = \text{Obs}_C(\text{cs})$) and to having an observation for every state (the observation function has a similar role as finalization in data refinement, which typically is also required to be total). Compared to ASM refinement the restriction has the advantage, that refinements are automatically
transitive, and that a notion of refinement "modulo IO" is not needed. ASM refinement is transitive only, when the states of interest of the middle machine are the same in both refinements (see [35] for a discussion).

As a disadvantage of the restriction, consider a refinement, where output of a 16-bit word in one abstract step is replaced by two concrete steps outputting the low and high byte separately. With the natural observation function, that sees the list of bytes output so far, this would be a correct ASM refinement (with IO = id), but not a correct refinement in the Abadi–Lampert setting. To get a correct refinement, the observation function of the concrete level would have to be tweaked to ignore the last low byte. The example is of course just a special case of any refinement with 1:n diagrams, that replaces atomic output in one step with a pretty printer, that generates output incrementally.

To verify refinement, Abadi and Lampert define forward simulations R and refinement mappings f.

**Definition 8.3 (Forward simulations and Refinement Mappings).**

A forward simulation between two machines CALM and AALM with the same observations is a relation $R \subseteq S_C \times S_A$ that satisfies

- $\text{l}_C(cs) \rightarrow \exists \text{as}. R(cs, as) \land \text{l}_A(as)$
- $R(cs, as) \land \text{SEM}_C(cs, cs') \rightarrow \exists as'. \text{SEM}_A(as, as') \land R(cs', as')$
- $R(cs, as) \rightarrow \text{Obs}_C(cs) = \text{Obs}_A(as)$
- For every concrete behavior $c\sigma$ and every run $\sigma_A$ such that $R(c\sigma(i), \sigma_A(i))$ holds for all $i, \sigma_A$ is a behavior.

A refinement mapping $f : S_C \rightarrow S_A$ is a total function which, when viewed as a relation, is a forward simulation.

The first three conditions are the usual forward simulation conditions with 1:1 diagrams (compared to data refinement, the observation function takes the role of finalization). The last global condition is necessary in general, since the abstract runs that are constructed by a forward simulation do not automatically satisfy the supplementary property. As a simple example consider a concrete level and an identical abstract level, that has an additional, non-trivial fairness constraint: for such a refinement, the first three conditions are obviously true using the identity function as refinement mapping, while the last condition is obviously violated.

A drawback of the last condition is, that it has to be checked globally. For some specific cases, where the supplementary conditions just specifies fairness constraints, [24] gives sufficient local conditions that imply the global condition.

As usual, forward simulations alone are not sufficient to imply completeness. Wim Hesselink gives the following completeness theorem using three extensions of the concrete machine, that will be discussed afterwards.

**Theorem 8.4 (Completeness of Abadi–Lampert refinement [25]).**

Every correct refinement from AALM to CALM can be proved by

- adding a clock to CALM to give Clk(CALM)
- forming the universal eternity extension $\text{UEt}(\text{Clk}(\text{CALM}))$ of Clk(CALM)
- adding a suitable temporization to give $\text{Tmp}(\text{UEt}(\text{Clk}(\text{CALM})), g)$
- defining a suitable refinement map $f$ between $\text{Tmp}(\text{UEt}(\text{Clk}(\text{CALM})), g)$ and AALM.

The first extension adds a counter of executed steps.

**Definition 8.5 (Clocking Extension).**

The states of the clocking extension Clk(ALM) of a machine ALM are pairs $(s, i)$, where $s \in S$ and $i \in \mathbb{N}$. The observation function observes $\text{Obs}(s)$. Initial states are $(s, 0)$ with $s \in I$ and the transitions are from $(s, i)$ to $(s', i + 1)$ when $(s, s') \in \text{SEM}$ (plus stutter steps). The supplementary property of Clk(ALM) requires that the supplementary property of ALM holds for the state sequence, and that the clock is infinitely often incremented.

Note that Clk(ALM) may count stutter steps, since $(s, s) \in \text{SEM}$ implies $((s, i), (s, i + 1)) \in \text{SEM}(\text{Clk}(\text{ALM}))$. It must do so on traces that stutter infinitely to satisfy the supplementary property. In an ASM setting adding a clock corresponds to putting an assignment $i := i + 1$ for a new counter $i$ (starting with 0) in parallel to the original rule. An additional liveness constraint or counting stutter steps is unnecessary in this setting, since an ASM cannot stutter (in particular, incrementing can stop when the ASM terminates).

The second extension is the most important one.

**Definition 8.6 (Universal Eternity Extension).**

The universal eternity extension $\text{UEt}(\text{Clk}(\text{CALM}))$ of a machine ALM has states which are pairs $(\sigma(i), \sigma)$, where $\sigma$ is a stutterfree behavior of ALM. The observation function observes $\text{Obs}(\sigma(i))$. Its initial states are $(\sigma(0), \sigma)$ and transitions are from $(\sigma(i), \sigma)$ to $(\sigma(i + 1), \sigma)$ (plus stutter steps). The supplementary property of UEt(ALM) is the supplementary property of ALM for the first components of the states.

The two extensions together construct a machine $\text{UEt}(\text{Clk}(\text{CALM}))$ from CALM. The stutterfree behaviors of Clk(CALM) are of the form $\rho = ((c_0, 0), (c_1, 1), (c_2, 2), \ldots)$ where $\sigma_C = (c_0, c_1, c_2, \ldots)$ is an arbitrary behavior of CALM. Therefore a state of $\text{UEt}(\text{Clk}(\text{CALM}))$ is of the form $(\rho, (c_k, k))$ for some $k$. Since the counters in $\rho$ are always $(0, 1, 2, \ldots)$ and $c_k = \sigma_C(k)$ a less redundant representation of these states is $(\sigma_C, k)$. Stutterfree behaviors of this system therefore have
essentially the form \((\sigma_0, 0), (\sigma_0, 1), (\sigma_C, 2), \ldots\). Every state stores full information to which behavior \(\sigma_C\) it belongs and how many steps have been executed. Both pieces of information are essential for the completeness proof, which we sketch below using this simplified state representation.

Compared to our semantic construction of the corresponding deterministic system \(\text{Det}(\text{ALM})\) for \(\text{ALM}\) in Definition 6.1 this definition does not just save the future of a state but also its past, so it keeps more information.

Our syntactic construction of an ASM with choice functions is even closer, since the choice functions does not forget old choices. This can be seen by encoding the transitions of an Abadi–Lamport system as the ASM rule

\[
\text{choose } cs' \text{ with SEM}(cs, cs') \text{ in } cs := cs'
\]

Replacing the \textit{choose} with a choice function gives the rule

\[
\text{let } cs' = \text{choice}(cs, n) \text{ in } n := n + 1 \text{ seq } cs := cs'
\]

where the counter \(n\) introduced is exactly the clock defined by \(\text{UEt}(\text{Clk}(\text{ALM}))\). Two differences remain: the resulting deterministic rule does not stutter (it always increments the clock), and it does not keep information about the initial state. Additionally saving the initial state \(cs_0\) would allow to retrieve all intermediate states recursively with \(cs_{i+1} = \text{choice}(cs_i, i)\).

**Definition 8.7 (Temporization).**

Temporization adds stutter steps to slow down a machine using a function \(g : S \rightarrow \mathbb{N}\). When the original machine \(\text{ALM}\) has a transition \((s, s') \in \text{SEM}\), the resulting machine \(\text{Tmp}(\text{ALM}, g)\) first does \(g(s)\) stutter steps, before going to \(s'\). Formally, \(\text{Tmp}(\text{ALM}, g)\) starts in states \((s, g(s))\) with \(s \in I\) and counts down from \((s, n)\) to \((s, n - 1)\) whenever \(n \neq 0\). For \((s, 0)\) it has transitions to \((s', g(s'))\) for every \((s, s') \in \text{SEM}\) (plus stutter steps). The observation function observes \(\text{Obs}(s)\) and the supplementary property is the same as the one of \(\text{ALM}\) for the first components of the pairs. Every behavior \(\sigma\) of \(\text{ALM}\) has a corresponding slowed down behavior \(\text{Tmp}(g, \sigma)\) in \(\text{Tmp}(\text{ALM}, g)\).

Temporization constructs a refinement with diagrams of sizes \(1 : g(s)\) for a step starting in state \(s\). It is necessary because refinement allows different speeds of the machines (being defined modulo stuttering) while a refinement mapping requires the same speed: consider a correct refinement and a concrete behavior \(\sigma_C\). Refinement guarantees the existence of an abstract behavior \(\sigma_A\) and \(\sigma_C\) with the same observations modulo stuttering. But in order to get a refinement map the abstract behaviors \(\sigma_A\) must have \textit{exactly the same} observations, since any refinement map guarantees \(\text{Obs}_A(\sigma_A) = \text{Obs}_C(\sigma_C)\). If \(\sigma_A\) changes the observations too fast, this can be corrected by choosing a slower \(\sigma'_A\) with more stutter steps. But if \(\sigma_C\) changes the observations too fast, there may be no suitable abstract trace that is fast enough. Therefore it is instead necessary to construct a version of the concrete machine which has all behaviors slowed down to match the speed of a suitable abstract run. This is what temporization achieves.

Based on the three extensions above the completeness proof roughly works as follows: first, it must be proved that each of the extensions is a correct refinement. This is relatively easy. Second, define \(\text{BALM} := \text{UEt}(\text{Clk}(\text{CALM}))\). Then a function \(g\) and a refinement mapping \(f\) must be found, such that \(\text{Tmp}(\text{BALM}, g)\) refines \(\text{AALM}\) with \(f\). This is done as follows: define \textit{a priori} a function \(c2a^g\) that maps every behavior \(\sigma_C\) to a fixed abstract behavior \(c2a(\sigma_C) = \sigma_A\) with the same observations modulo stuttering (by using the fact, that \(\text{CALM}\) is a correct refinement of \(\text{AALM}\) and the axiom of choice). \(c2a(\sigma_C)\) can be chosen not to change observations faster than \(\sigma_C\) as discussed above. Therefore a function \(g'\) exists, such that runs of \(\text{Tmp}(g', \sigma_C)\) are exactly as fast as \(c2a(\sigma_C)\), i.e. \(\text{Obs}_C(\text{Tmp}(g', \sigma_C)) = \text{Obs}_A(c2a(\sigma_C))\). Note that \(g'\) is uniquely defined, since every state \((\sigma_C, i)\) appears only once in one run of \(\text{BALM}\). Now, define \(g\) as \(g(\sigma_C, n) := (g'(\sigma_C), n)\). Since the behaviors of \(\text{BALM}\) without stutter steps have the form \(((\sigma_C, 0), (\sigma_C, 1), \ldots\) the behaviors of \(\text{Tmp}(\text{BALM}, g)\) have the form \(((g'(\sigma_C), 0), (g'(\sigma_C), 1), \ldots\). By construction, these runs change observations exactly as the run \(c2a(\sigma_C)\), so the refinement map \(f\) needed must just map \(g'(\sigma_C, n)\) to \(c2a(\sigma_C)(n)\).

For ASM refinement temporization is not a relevant construction, since generalized forward simulations already allow \(1:n\) diagrams. As a side remark, stuttering forward simulations (defined similar to Definition 7.6 for \(\text{IO automata}\)) allow the construction of \(m:1\) diagrams in the Abadi–Lamport setting.

Summarizing the results, we find that there are several technical differences between the ASM setting and the Abadi–Lamport setting: the first has failing rules and terminating runs, while the second has built-in stuttering and considers fairness. Nevertheless the universal eternity construction is very similar to the ASM refinement proof: it essentially adds a global choice function.

**9. Completeness of fair ASM refinement**

The result of the previous section considers one additional problem present in the Abadi–Lamport setting, but not in ASMs: arbitrary fairness constraints. In this section we transfer the result to the ASM setting by considering fair transition systems.

\(\text{In [25], the function is called } \varepsilon.\)
Definition 9.1 (Fair Transition System). 
A fair transition system \((M, \text{Sup})\) extends a transition system \(M\) with an arbitrary predicate \(\text{Sup} \subseteq S^* \cup S^o\) that restricts the runs of \(M\). A behavior \(\sigma\) is a run of \(M\) that satisfies \(\text{Sup}\).

Note that since ASMs do not consider stuttering, the predicate does not have to be a property. This means that full temporal logic (e.g. CTL\(^*\), or the temporal logic for programs used in KIV [3,6,4] can be used for specification: for the Abadi–Lamport setting, operators like \(X\) (“in the next state”) must be forbidden. Also a liveness constraint that rules out infinite runs which just stutter is unnecessary.

Correctness of fair ASM refinement replaces runs with behaviors.

Definition 9.2 (Correctness of Fair ASM Refinement). A refinement is correct with respect to IO if for every behavior \(\sigma_C\) a behavior \(\sigma_A\) related by IO exists.

To cope with the supplementary predicate, generalized forward simulations (like the forward simulations in the Abadi–Lamport setting) now need an additional global condition, to ensure that the abstract supplementary predicate is satisfied. To understand the necessary condition, consider a behavior \(\sigma_C\). The generalized forward simulation will construct commuting diagrams as shown in Fig. 2. Putting them together constructs a run \(\sigma_A\) that is related to \(\sigma_C\) by R (and therefore also by IO) according to Definition 4.1. The run \(\sigma_A\) consist of pieces, each belonging to one commuting diagram. Each of these pieces will not be unique, since the commuting diagram guarantees existence of the piece, but not uniqueness. Putting together the pieces to a run will result in a behavior sometimes, in some other cases the supplementary predicate will be violated. For the refinement to be correct, it is sufficient that at least one selection of pieces leads to a behavior.

Definition 9.3 (Generalized Fair Forward Simulation).
A generalized fair forward simulation \(R\) between \((CM, C\text{Sup})\) and \((AM, A\text{Sup})\) is a generalized forward simulation between \(CM\) and \(AM\) that additionally satisfies the following global condition:

for every behavior \(\sigma_C\) of \((CM, C\text{Sup})\) and every run \(\sigma_A\) of \(AM\) related by \(R\) there must be a behavior \(\sigma'_A\) of \((AM, A\text{Sup})\) which goes through the same states of interest as \(\sigma_A\) that is related to \(\sigma_C\) by \(R\).

To prove completeness, an intermediate ASM \(\text{Det}(M, \text{Sup})\) with states \((\sigma_C, i)\) is constructed that corresponds to the construction of \(\text{UEt}(\text{Clk}(\text{ALM}))\).

Definition 9.4 (Corresponding Fair Deterministic Transition System).
Given a fair transition system \((M, \text{Sup})\) with \(M = (S, I, F, \text{SEM})\) the corresponding deterministic transition system \(\text{Det}(M, \text{Sup}) := ((S', \i', F', \text{SEM'}), \text{Sup'})\) is defined as:

- \(S' := (S^* \cup S^o) \times \mathbb{N}\), the set of all finite and infinite sequences of states.
- \(\i'\) consists of all \(\sigma, 0\) where \(\sigma\) is a behavior of \((M, \text{Sup})\).
- \(\text{SEM'} := \{(\sigma, n), (\sigma, n + 1) \} \cup \{(\sigma, n), \bot \} : s \notin F\). Again the second set of the union adds transitions to \(\bot\) if the run ends with a failed rule application. Note that these final failing transitions were removed in the definition of runs and behaviors.
- \(F' := \{(s, n) : s \in F\}\) consists of all sequences consisting of a single final state and an arbitrary counter value.
- \(\text{Sup'}\) is the trivial predicate that is always true.

It is easy to see, that the behaviors of \(\text{Det}(M, \text{Sup})\) correspond to those of \((M, \text{Sup})\), exactly like the runs of \(\text{Det}(M)\) corresponded to those of \(M\).

Theorem 9.5 (Equivalence of \((M, \text{Sup})\) and \(\text{Det}(M, \text{Sup})\)).

- \(\text{Det}(M, \text{Sup})\) is deterministic: each state has at most one successor state.
- For every behavior \(\sigma\) of \((M, \text{Sup})\), \(\text{Det}(M, \text{Sup})\) has a behavior \(((\sigma, 0), (\sigma, 1), (\sigma, 2), \ldots)\) of the same length. In particular the behavior is infinite iff \(\sigma\) is infinite.
- Every behavior of \(\text{Det}(M, \text{Sup})\) has the form \(((\sigma, 0), (\sigma, 1), (\sigma, 2), \ldots)\) for a behavior \(\sigma\) of \((M, \text{Sup})\).
- \((M, \text{Sup})\) is a correct ASM refinement of \(\text{Det}(M, \text{Sup})\) using \(\text{IO}(s, (\sigma, i)) := \Leftrightarrow (\sigma)_i = s\).

As already argued in the previous section, it is not difficult to see, that \(\text{Det}(M, \text{Sup})\) can be constructed syntactically by introducing choice functions, by adding a global counter, and by keeping a copy of the initial state.

To prove completeness, the same idea as in the previous section is used: define a function \(c2a\), that maps behaviors (now finite as well as infinite ones) of \(\text{Det}(CM, C\text{Sup})\) to behaviors of \((AM, A\text{Sup})\). The correctness of refinement and the axiom of choice guarantee the existence of such a function. To every behavior \(\sigma_C\) the behavior \(c2a(\sigma_C)\) is related by IO. Therefore, if \(\sigma_C\) is infinite, then so is \(c2a(\sigma_C)\), and two functions \(c2i(\sigma_C), c2j(\sigma_C) : \mathbb{N} \rightarrow \mathbb{N}\) can be defined (again using the axiom of choice), that give the states of interest on both behaviors:

\[
\forall \ k. \ \text{IO}(\sigma_C(c2i(\sigma_C)(k)), c2a(\sigma_C)(c2i(\sigma_C)(k)))
\]

holds for all infinite behaviors \(\sigma_C\) of \((CM, C\text{Sup})\). This is already sufficient to prove the completeness theorem.
Theorem 9.6 (Completeness of Fair ASM Refinement).

If \((CM, CSup)\) is a correct fair ASM refinement of \((AM, ASup)\) with respect to IO, then a generalized forward simulation exists between \(Det(CM, CSup)\) and \((AM, ASup)\).

The generalized forward simulation between \(Det(CM, CSup)\) and \((AM, ASup)\) is defined as:

\[
R(as, (\sigma_C, j)) :\leftrightarrow
\begin{align*}
& \text{if } \#(\sigma_C) < \infty \\
& \text{then } (j = 0 \land as = c2a(\sigma_C)(0) \lor j + 1 = \#(\sigma_C) \land \text{final}(as) \land as = last(c2a(\sigma_C))) \\
& \text{else } 3 \ k. \ c2i(\sigma_C)(k) = j \land as = c2a(\sigma_C)c2i(\sigma_C)(k)
\end{align*}
\]

Note that \(R\) is a partial function that generalizes the total refinement mappings of the Abadi–Lamport proof (since not all states are states of interest). It maps exactly those states \((\sigma_C, j)\) to one abstract state of \(c2a(\sigma_C)\), where \(\sigma_C(j)\) was one of the states of interest. The proof that the definition gives a generalized forward simulation is rather similar to the one for the completeness proof of ASM refinement (since the concrete states store even more information than in the original proof, this is to be expected). The new global condition of Definition 9.3 that guarantees that it is a generalized fair forward simulation is satisfied too: a run of \(Det(CM, CSup)\) is already determined by the behavior \(\sigma_C\) stored in every state. When an abstract run \(\sigma_A\) is related to it by \(R\) the definition of \(R\) gives that \(\sigma_A\) must pass through exactly the states of interest of \(c2a(\sigma_C)\). Therefore choosing \(\sigma'_A := c2a(\sigma_C)\) is sufficient to verify the condition.

The definition of \(R\) looks simpler than the one from Theorem 6.4, but it is essentially the same. The existential quantifiers over \(\sigma_A, (l_p)_{p \in \mathbb{N}}, (j_p)_{p \in \mathbb{N}}\) and the properties of these variables (e.g. that \((l_p)\) is monotonic; that \(\sigma_A\) is a behavior; that the last state of \(\sigma_C\) is final iff the last state of \(c2a(\sigma_C)\) is final) are now part of the definitions of the skolem functions \(c2a, c2i\) and \(c2j\). Theorem 6.4 also has an existentially quantified run \(\sigma_C\) which passes through state \(s\). This is now the behavior \(\sigma_C\) which passes through \((\sigma_C, j)\).

The proofs have been mechanized in KIV and are available on the Web [28]. We have also repeated some of Wim Hesselink’s PVS proofs in KIV, to get an understanding of the concepts used in the Abadi–Lamport proofs (this is easy, as all the theorems are correct already). A rough comparison of the proofs is as follows: the Abadi–Lamport proofs are complicated by the presence of stuttering, which must be taken into account everywhere. The algebraic level of reasoning over transition systems underlying ASMs is slightly simpler. For the ASM rules, considering finite runs and failing rules is a significant source of complexity, since it creates special cases in all the proofs. The proofs for ASM refinement are less modular than the ones for the Abadi–Lamport setting, since they merge steps. Combining UEt and Clk is not much of a problem, the combined extension is still easy enough. But the power of generalized forward simulations, which allows us to combine stuttering forward simulations and temporization extensions makes proofs harder, since it becomes necessary to reason with three choice functions \((c2a, c2i\) and \(c2j)\) instead of just one.

A significant difference between the proofs is also their specification style. While the PVS proofs define Abadi–Lamport machines as certain tuples in a PVS specification, and defines a predicate isfwSim\((R, AALM, CALM)\) that characterizes a relation \(R\) to be a forward simulation between two tuples AALM and CALM, in KIV a machine \(M\) is a specification that contains a carrier sort for the states, a SEM relation etc. A forward simulation \(R\) is axiomatized as a relation \(R\) over the union of two specifications that define CM and AM, a predicate isfwSim is not defined. The PVS style has the advantage, that the theory structure can be much smaller, while the KIV style seems more natural when applying the theory in applications: in KIV, the individual proof obligations for e.g. a forward simulation get generated directly for the application instance, instead of one big proof obligation for isfwSim.

10. 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.

Wim Hesselink has given a completeness proof for the Abadi–Lamport setting of refinement that copes with infinite nondeterminism. An analysis of this proof shows that although there are technical differences between the settings, the universal eternity extension corresponds closely to our use of choice functions.

In the Abadi–Lamport setting, the use of a supplementary property to specify liveness and fairness conditions is essential to avoid having machines that never do anything. For ASMs such an extension is useful too, although the applications done in KIV have not needed it yet. Therefore we have given a similar extension for the ASM setting together with a definition of fair ASM refinement. The extended completeness proof shows that generalized forward simulations together with using choice functions is still nearly a complete proof method: sometimes it is also necessary to save a copy of the initial state or to add a global counter, that counts steps of the ASM. The technical details of the completeness proof differ due to the different
formalisms. Nevertheless, a key idea of the proof, to use the axiom of choice to construct a function that maps concrete behaviors to abstract behaviors, was taken over.

While the Abadi–Lamport setting does not consider failing rules and termination, it considers fairness and liveness conditions, which makes the proof of completeness more difficult.

Whether a similar completeness result can be obtained for data refinement 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.

Acknowledgements

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 Grundy, Nina Moebius and Kurt Stenzel for carefully reading the drafts of this paper. Thanks are also due 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 reviewers who directed me to Wim Hesselink’s work and gave lots of helpful comments.

References


