Journal of Universal Computer Science, vol. 7, no. 11 (2001), 952-979
submitted: 7/6/01, accepted: 1/10/01, appeared: 28/11/01

Verification of ASM Refinements Using Generalized Forward Simulation

Gerhard Schellhorn
(University of Augsburg, Germany
schellhorn@informatik.uni-augsburg.de)

Abstract: This paper describes a generic proof method for the correctness of refinements of Abstract State Machines based on commuting diagrams. The method generalizes forward simulations from the refinement of I/O automata by allowing arbitrary m:n diagrams, and by combining it with the refinement of data structures.
Categories: D.2.1, D.2.4, F.3.1, F.3.2, F.4.1
Key Words: Refinement, Forward Simulation, Data Refinement, Commuting Diagrams, Abstract State Machines, Transition Systems, I/O-Automata, Dynamic Logic, Correctness Proofs, Interactive Theorem Proving, Compiler Verification

1 Introduction

Abstract state machine (ASM) refinements have been used in many case studies based on Gurevich's [Gur95] definition, e.g. in [BR95] [BM96], [BS98], [BS00b].

ASM refinements are usually verified using an informal notion of *commuting diagrams* to structure correctness proofs. Commuting diagrams are also used in approaches to the refinement of data structures (such as data refinement [dRE98]) as well as in simulation approaches for I/O automata [LV95], which focus on the refinement of control structure. Since refining one ASM by another may modify the data as well as the control structure, a generic proof technique for the verification of ASM refinements must necessarily combine the proof techniques used in both domains.

This paper gives a generic approach, that formalizes the idea of correctness proofs using commuting diagrams. The motivation for this work is a case study on compiler verification, where commuting diagrams have been regularly used too (e.g. in the very large case study [Moo88]). Our case study consists in the formal specification and verification of a Prolog compiler ([SA97], [SA98], [Sch99]). It is based on [BR95], where a Prolog interpreter (specified as an ASM) is transformed in several refinement steps to an interpreter of assembler code of the Warren Abstract Machine (WAM). Commuting diagrams of various types are used informally in [BR95] to justify correctness of the refinements: one time, 2 rule applications of the first ASM are refined by 3 applications in the next (2:3 diagram), other refinements use 0:1 as well as 1:0 diagrams, which implies, that the correspondence between the states of the ASMs can not be functional. The most complex refinements use m:n diagrams, where m and n depend on the sizes of the data structures stored in the states of the two ASMs involved.

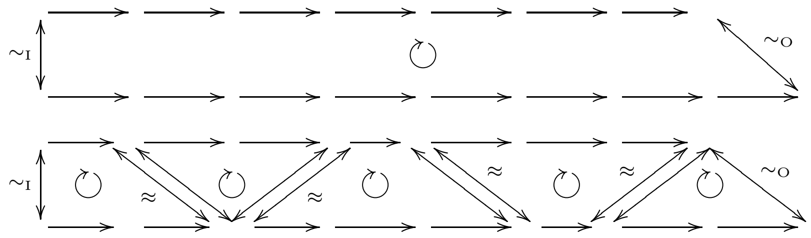

Figure 1: Verification of a refinement using commuting diagrams

The theory developed here justifies the use of arbitrary m:n diagrams, and gives precise verification conditions. It was used successfully in [Sch99] to formalize the proofs of 9 refinements. Although first intended specifically for deterministic ASMs, we found it not too difficult to generalize it to indeterministic transition systems. Transition systems, execution traces, runs and a number of temporal properties are defined in Sect. 2.

Section 3 gives four definitions of refinement correctness: Preservation of partial and total correctness are already known from compiler verification, and targeted to a pre/postcondition semantics, where only the results of computations are of interest. Partial and total preservation of traces should be used for reactive systems which should have comparable intermediate states. A common requirement for ASMs in this case is that both should modify the value of some dynamic functions (output functions) in the same way.

Section 4 is the core of our theory: we define a generic proof method for the verification. The idea is to allow two runs of the involved transition systems to be split into arbitrary diagrams using a coupling invariant $\approx$ as shown in Fig. 1.

Verification of refinement correctness is reduced to the verification of a proof obligation, which expresses the commutativity of a single diagram. We will call the proof method *generalized forward simulation* since it generalizes forward simulations from I/O automata by including data refinement and by allowing arbitrary diagrams instead of m:1 diagrams. We will show, that with minor adaptations the proof method implies each of the four correctness notions.

The proof obligation to verify refinement correctness defined in Sect. 4 is very generic. Section 5 shows how to apply the theory to ASMs by expressing the proof obligation in Dynamic Logic (DL). Thereby the need to encode the semantics of ASM rules as an explicit transition relation is avoided. A higher order-variant of DL is implemented in the interactive specification and verification system KIV ([RSSB98]), which was used to formally check all the proofs of this paper. Instantiating the DL theorems, the KIV system can be directly used to specify and verify ASM refinements. This was done for the 9 refinements which were verified in the Prolog-WAM case study mentioned earlier. Since examples, which require m:n diagrams with m,n = 0 or both m,n > 1 are either too complex, to be presented in this paper or are easily reduced to the standard 1:n (or m:1) case (see Theorem 12 for an explanation), we refer for numerous examples of such refinements to [Sch99].

Section 6 compares our work to data refinement, work from compiler verification and to the refinement of I/O-automata. In particular we will show, that our proof technique generalizes forward simulation, and allows in some case to avoid proofs using backward simulation. Generalized backward simulation is defined, and some of its properties are sketched. Finally, Sect. 7 concludes and gives some topics of further research.

2 Transition Systems and Abstract State Machines

In this section we will define a generic framework for the refinement of transition systems. Although we have refinement of ASMs in mind, we will not be specific about the set of states nor about the way transitions are specified. Our definition is simply:

Definition 1. [Transition System]
A transition system $\mathrm{M} = (\mathrm{S},\mathrm{I},\rho)$ consists of a set of states, a subset $\mathrm{I} \subseteq \mathrm{S}$ of initial states and a transition relation $\rho \subseteq \mathrm{S} \times \mathrm{S}$

For the usual definition of an ASM as given in [Gur95], which the reader is assumed to be familiar with, the set of states is some set of algebras over a given untyped first-order signature, and the transition relation is computed as the set of updates of the ASM rules. We have refrained from being so specific, since we want our approach to be applicable for any set of states, e.g.:

- typed ASMs with a many-sorted (or higher-order) signature.
- ASMs with recursive rules, where a rule application may fail to terminate. In this case we include a "$\perp$" Algebra in our set of states to express divergence.
- ordinary valuations of (first-order or higher-order) variables.
- state sets of I/O automata (this instance will be discussed in Sect. 6).

We have also not been specific about the transition relation or the syntax of rules. Here, we just want to fix the following "standard case", to be used later in specializations of the main theorem:

Definition 2. [standard ASM]
A standard ASM has the form

$$\begin{array}{l} \text{if } \epsilon_1 \text{ then RULE}_1 \\ \text{if } \epsilon_2 \text{ then RULE}_2 \\ \ldots \\ \text{if } \epsilon_n \text{ then RULE}_n \end{array}$$

where $\epsilon_1 \ldots \epsilon_n$ are disjoint predicates over states, i.e. $\neg\,(\epsilon_i \wedge \epsilon_j)$ holds for $i \neq j$.

Typical standard ASMs used in compiler verification (e.g. [BR95] for Prolog and [SSB01] for Java) specify an interpreter, where the tests ϵ_i distinguish between the instructions of the interpreted language. Often, each $\mathrm{RULE_i}$ is just a number of parallel function updates.

To reason over a transition systems M we make use of some typical relational operators: $\rho;\xi$ is the composition of relations $\rho \subseteq \mathrm{S} \times \mathrm{S}'$ and $\xi \subseteq \mathrm{S}' \times \mathrm{S}''$. ρ^{-1} is the inverse relation of ρ. ρ^{n} is the n-fold composition of ρ (ρ^0 is the identity relation). ρ^* is the union of all ρ^{n} for $\mathrm{n} \geq 0$.

Definition 3. [final states, traces, runs and I/O behavior]

- A state s of M is *final*, if it has no successor state with respect to ρ. The set of final states is denoted with O (output states). In a standard ASM, the predicate final(s) is the conjunction of all negated rule tests.
- A trace (or execution) $\sigma : \mathrm{nat} \to \mathrm{S}$ of M is a sequence of states, such that for each i either $\sigma(\mathrm{i})$ is not final and $\rho(\sigma(\mathrm{i}),\sigma(\mathrm{i}+1))$ holds, or otherwise $\sigma(\mathrm{i}+1) = \sigma(\mathrm{i})$. We write *trace($\sigma$)* in this case.
- A *run* of M is a trace σ that starts with an initial state $\sigma(0) \in \mathrm{I}$.
- The *partial* input/output behaviour *PIO(M)* $\subseteq \mathrm{S} \times \mathrm{S}$ of M is the set of pairs $(\mathrm{s},\mathrm{s}_0)$ of states such that $\mathrm{s} \in \mathrm{I}$, $\mathrm{s}_0 \in \mathrm{O}$ and $\rho^*(\mathrm{s},\mathrm{s}_0)$.
- The *total* input/output behavior *TIO(M)* $\subseteq \mathrm{S} \times (\mathrm{S} \cup \{\perp\})$ extends the partial one by adding pairs $(\mathrm{s},\perp)$ for those initial states s with an infinite (i.e. nonterminating) run.

Our formal definition of traces extends a finite trace, which ends in a final state, to an infinite one by repeating the final state. This uniform treatment simplifies the axioms and theorems which will be given later on. Nevertheless we will call a trace finite, when it does repeat a final state. Note that a trace which becomes constant by repeating a state s with $(s,s) \in \rho$ is possible, but *not* called finite.

To reason over executions of transition systems we define two temporal operators AF(s,p) ("for all executions starting with s predicate p will eventually hold") and EF(s,p) ("for some execution starting from s predicate p will eventually hold"):

$$\begin{aligned} &\mathrm{AF(s,p)} :\leftrightarrow \forall\, \sigma.\ \sigma(0) = \mathrm{s} \wedge \mathrm{trace}(\sigma) \to \exists\, \mathrm{n}.\ \mathrm{p}(\sigma(\mathrm{n})) \\ &\mathrm{EF(s,p)} :\leftrightarrow \exists\, \sigma.\ \sigma(0) = \mathrm{s} \wedge \mathrm{trace}(\sigma) \wedge \exists\, \mathrm{n}.\ \mathrm{p}(\sigma(\mathrm{n})) \end{aligned} \tag{1}$$

It is easy to verify that these operators could also be defined as fixpoints of the following recursions (as was done in PVS [RSS95]):

$$\begin{aligned} &\mathrm{AF(s,p)} \leftrightarrow \mathrm{p(s)} \vee \neg\, \mathrm{final(s)} \wedge \forall\, \mathrm{s}_0.\ \rho(\mathrm{s},\mathrm{s}_0) \to \mathrm{AF}(\mathrm{s}_0,\mathrm{p}) \\ &\mathrm{EF(s,p)} \leftrightarrow \mathrm{p(s)} \vee \neg\, \mathrm{final(s)} \wedge \exists\, \mathrm{s}_0.\ \rho(\mathrm{s},\mathrm{s}_0) \wedge \mathrm{EF}(\mathrm{s}_0,\mathrm{p}) \end{aligned} \tag{2}$$

For finite nondeterminism, i.e. when $\{s_0 : \rho(s,s_0)\}$ is finite for all states s, AF(s,p) and EF(s,p) can also be defined by iteration[1]:

$$\begin{aligned} &\mathrm{AF(s,p)} \leftrightarrow \exists\ \mathrm{n}.\ \forall\ \mathrm{s_0}.\ \rho_{\mathrm{p}}^{\mathrm{n}}(\mathrm{s,s_0}) \wedge \mathrm{p(s_0)} \\ &\mathrm{EF(s,p)} \leftrightarrow \exists\ \mathrm{n}.\ \exists\ \mathrm{s_0}.\ \rho_{\mathrm{p}}^{\mathrm{n}}(\mathrm{s,s_0}) \wedge \mathrm{p(s_0)} \end{aligned} \tag{3}$$

where $\rho_p := \{(s,s_0) : \text{if } \neg\, p(s) \wedge \neg\, \mathrm{final}(s) \text{ then } \rho(s,s_0) \text{ else } s = s_0\}$. Finally, we define abbreviations AF+(s,p) and EF+(s,p) which require p to hold in a state reachable from s with a *positive* number of rule applications:

$$\begin{aligned} &\mathrm{AF{+}(s,p)} :\leftrightarrow \neg\ \mathrm{final(s)} \wedge (\forall\ \mathrm{s_0}.\ \rho(\mathrm{s,s_0}) \rightarrow \mathrm{AF(s_0,p)}) \\ &\mathrm{EF{+}(s,p)} :\leftrightarrow \neg\ \mathrm{final(s)} \wedge (\exists\ \mathrm{s_0}.\ \rho(\mathrm{s,s_0}) \wedge \mathrm{EF(s_0,p)}) \end{aligned} \tag{4}$$

3 Refinement of Transition Systems

In this section we will discuss refinement (also called implementation) of a transition system $M = (S,I,\rho)$ by another $M' = (S',I',\rho')$. M is often called the "abstract" system, M′ the "refined" or "concrete" system. We will use primed versions of all notions defined in the previous sections, when we apply them to M′, so we will use s′ for a state of M′, O′ for its final states, σ', final′, AF′, etc.

We will now give four definitions of refinement correctness. The first two notions do not consider intermediate states of M and M′, but input/output behavior only. They are typically used in compiler verification, where M is an interpreter for a source code program and M′ is an interpreter (or a processor) that executes compiled machine code. Both definitions assume that we have given two relations $\sim_I \subseteq I \times I'$ and $\sim_O \subseteq O \times O'$ which determine, when initial resp. final states are considered to be equivalent. In compiler verification, $\sim_I$ often is defined as a function that maps an initial state of the interpreter M which stores a program p and its input to an initial state of M′ which stores the compiled program compile(p). But if we just give some properties of function compile ("compiler assumptions") that allow various implementations then $\sim_I$ may also be non-functional. Similarly, the usual requirement for $\sim_O$ in compiler verification is, that M′ stores some representation of the output value computed by M.

Definition 4. [preservation of partial correctness]
Given two relations $\sim_I \subseteq I \times I'$ and $\sim_O \subseteq O \times O'$, M′ is a correct refinement of M that preserves partial correctness (with respect to the two relations) iff for any finite run σ' of M′ with final state $\sigma'(n)$ there exists a run σ of M, such that $\sigma(0) \sim_I \sigma'(0)$, and for some m, $\sigma(m)$ is final with $\sigma(m) \sim_O \sigma'(n)$.

[1] the restriction to finite nondeterminism is necessary only for the AF operator

$$\mathrm{M}: \quad s_0 \longrightarrow s_1 \longrightarrow s_2 \qquad\qquad \mathrm{M}': \quad \begin{tikzcd} s'_0 \arrow[r] \arrow[dr] & s'_1 \arrow[r] & s'_2 \\ & s'_3 \arrow[r] & s'_4 \arrow[r] & s'_5 \arrow[r] & \cdots \end{tikzcd}$$

Figure 2: Refinement that preserves partial, but not total correctness

Informally, the definition says, that terminating runs of M′ can not yield other results, than could be obtained by running M. Terminating refined runs simulate terminating abstract runs via the input/output correspondence. Preservation of partial correctness can also be specified using partial input/output behavior as $\mathrm{PIO}(\mathrm{M}') \subseteq {\sim_\mathrm{I}}^{-1};\mathrm{PIO}(\mathrm{M});\sim_\mathrm{O}$. To justify the terminology, note that in the case, where $\mathrm{S} = \mathrm{S}'$ and $\sim_\mathrm{I}$, $\sim_\mathrm{O}$ are both the identity relation, a correct refinement implies that every partial correctness assertion (as definable in Hoare's calculus [Hoa69]) that holds for M also holds for M′. Preservation of partial correctness is a weak form of correctness, since it allows to implement an ASM with one terminating (i.e. finite) run by one, which always diverges. Nevertheless this notion has interesting applications in compiler verification (see [GDG+96]).

To rule out the implementation of terminating computations with nonterminating ones, a first idea would be to require that not only M′ is a correct refinement of M, but also that M is a correct refinement of M′ (we say that the refinement is "complete" in this case). This approach works for deterministic ASMs, but not for nondeterministic ones: Fig. 2 shows an example, where a deterministic ASM M with exactly one terminating run is refined with a nondeterministic M′, that has the same terminating run, but an additional nonterminating one (assuming that the two initial states as well as the two final states are equivalent).

In the example every *terminating* run of M′ simulates a terminating run of M (and vice versa). But still M′ may fail to terminate. Also, completeness prevents implementation of a nondeterministic ASM with several terminating runs by a deterministic one which has only one of them, which is often convenient. A refinement notion, which allows to restrict nondeterminism, but rules out the situation of Fig. 2 is the following:

Definition 5. [preservation of total correctness]
A refinement from M to M′ preserves total correctness iff it preserves partial correctness, and if for any infinite run σ' of M′ there exists an infinite run σ of M such that $\sigma(0) \sim_\mathrm{I} \sigma'(0)$.

The stronger definition implies, that terminating *as well as non-terminating* refined runs simulate an abstract run via the input/output correspondence. The inclusion $\mathrm{TIO}(\mathrm{M}') \subseteq {\sim_\mathrm{I}}^{-1};\mathrm{TIO}(\mathrm{M});(\sim_\mathrm{O} \cup \{(\perp,\perp)\})$ is implied by the preservation of of total correctness. In the special case, where $\mathrm{S} = \mathrm{S}'$ and $\sim_\mathrm{I}$, $\sim_\mathrm{O}$ are both the identity relation, this means, that the refinement preserves all total correctness assertions.

Our third and fourth definition of refinement correctness are targeted towards ASMs, where not only the input/output behaviour matters, but where the ASMs are required to pass through "similar" states. We will give a very generic definition using an arbitrary similarity relation $\sim$. A typical instance of $\sim$ is, that both ASMs give the same outputs (or additionally, have accepted the same inputs). We will discuss this instance in detail when we show that I/O automata refinement is a special case in Sect. 6. The generic definition is:

Definition 6. [partial and total preservation of traces]
Given a relation $\sim \subseteq \mathrm{S} \times \mathrm{S}'$ a refinement from M to M′ totally preserves traces iff it preserves total correctness for $\sim_{\mathrm{I}}$ defined to be $\sim \cap\ (\mathrm{I} \times \mathrm{I}')$ and for $\sim_{\mathrm{O}} := \sim \cap\ (\mathrm{O} \times \mathrm{O}')$, and if for any infinite run σ' of M′ there is an infinite run σ of M and two strictly monotone sequences $\mathrm{i}_0 < \mathrm{i}_1 < \dots$ and $\mathrm{j}_0 < \mathrm{j}_1 < \dots$ of natural numbers, such that for every k $\sigma(\mathrm{i}_\mathrm{k}) \sim \sigma'(\mathrm{j}_\mathrm{k})$ holds. For a refinement to partially preserve traces, the refinement must preserve partial correctness and the sequence $\mathrm{i}_0 \leq \mathrm{i}_1 \leq \dots$ is only required to be monotone.

As for preservation of total correctness, we require that each refined run simulates an abstract run via the input/output correspondence. But additionally we require that an *infinite* refined run must simulate the corresponding abstract one by passing through infinitely many corresponding states. Just like preservation of partial correctness, partial preservation of traces allows to refine a finite run with an infinite run, while total preservation of traces does not. While we prefer total preservation of traces as the more intuitive notion, it should be noted, that refinement of I/O automata views termination as an invisible property, and prefers the weaker notion of partial preservation of traces (see Sect. 6).

In practical applications, we will often have diagrams of minimal size, i.e. the sequences $(\mathrm{i}_1, \mathrm{i}_2, \dots)$ and $(\mathrm{j}_1, \mathrm{j}_2, \dots)$ will additionally satisfy, that there are no i and j with $\mathrm{i}_\mathrm{k} < \mathrm{i} < \mathrm{i}_{\mathrm{k}+1}$ and $\mathrm{j}_\mathrm{k} < \mathrm{j} < \mathrm{j}_{\mathrm{k}+1}$, such that $\mathrm{s}_\mathrm{i} \sim \mathrm{s}_\mathrm{j}$ holds. We did not add this minimality requirement to the definition, since it can be easily shown, that the existence of suitable sequences $(\mathrm{i}_1, \mathrm{i}_2, \dots)$ and $(\mathrm{j}_1, \mathrm{j}_2, \dots)$ implies the existence of sequences that additionally satisfy the minimality requirement.

4 Generalized Forward Simulation

In this section we consider verification of ASM refinements by using commuting diagrams. The idea already suggested by Fig. 1 is to split similar computations into (finitely or infinitely) many subcomputations (of finite length), and to verify that each pair of subcomputations preserves a *coupling invariant* $\approx$, i.e. a formula that describes some similarity relation. If we want to verify preservation of traces,

$$\begin{tikzcd}
\mathrm{s} \arrow[r] \arrow[d, leftrightarrow, "\approx"'] & \mathrm{s}_1 \arrow[r] & \cdots \arrow[r] & \mathrm{s}_\mathrm{m} \arrow[dr, leftrightarrow, "\approx"] & \\
\mathrm{s}' \arrow[r] & \mathrm{s}'_1 \arrow[r] & \cdots \arrow[rr] & & \mathrm{s}'_\mathrm{n}
\end{tikzcd}$$

Figure 3: m:n diagram

the coupling invariant may be equal to the similarity relation $\sim$, but often a generalization is needed.

Our proof method will propagate the invariant forward through traces, so it is a form of forward simulation (we will consider backward simulation in Sect. 6). The main problem we have to solve, is that we want to be able to use arbitrary diagrams to cover two similar runs of both ASMs, not just 1:1 or 1:n diagrams which are often found in the literature.

Given a similarity relation $\approx$ between two ASMs M and M$'$, a first attempt to define a "most general" commuting diagram is to require that for two similar states $\mathrm{s} \approx \mathrm{s}'$, not both final, there must be m,n, not both zero, such that M reaches a state s_m with m rule applications (i.e. $\rho^\mathrm{m}(\mathrm{s},\mathrm{s}_\mathrm{m})$) and M$'$ reaches a similar state in n steps (i.e. $\rho'^\mathrm{n}(\mathrm{s}',\mathrm{s}'_\mathrm{n})$ and $\mathrm{s}_\mathrm{m} \approx \mathrm{s}'_\mathrm{n}$) as shown in Fig. 3.

To make sure, that every trace of M$'$ is a refinement of an M trace, we must require a suitable state s_m to exist for every choice of s'_n. Unfortunately this approach is still not general enough: if M$'$ is nondeterministic, we may have to choose the size n of the diagram depending on the actual trace that M$'$ takes. In the worst case, when M$'$ simulates one step of M with a randomly chosen number of steps, there is no fixed upper bound to n. Therefore we allow the number of rule applications of M$'$ to depend on the actual trace chosen, and require only, that M$'$ *eventually* reaches a state s'_n which is again similar to some state s_m of M. Using the AF and EF operators of Sect. 1 we can express the commutativity requirement for a diagram as follows:

$$\begin{array}{ll}
 & \mathrm{s} \approx \mathrm{s}' \wedge \neg\ (\mathrm{final}(\mathrm{s}) \wedge \mathrm{final}'(\mathrm{s}')) \\
\rightarrow & \quad \mathrm{EF}{+}(\mathrm{s},\lambda\ \mathrm{s}_0.\ \mathrm{s}_0 \approx \mathrm{s}') \\
 & \vee\ \mathrm{AF}'{+}(\mathrm{s}',\lambda\ \mathrm{s}'_0.\ \mathrm{EF}(\mathrm{s},\lambda\ \mathrm{s}_0.\ \mathrm{s}_0 \approx \mathrm{s}'_0))
\end{array} \tag{5}$$

The first disjunct considers (triangular) m:0 diagrams where after a positive number of rule applications of M a state s_0 with $\mathrm{s}_0 \approx \mathrm{s}'$ is reached. The second disjunct considers m:n diagrams where $\mathrm{n} > 0$: Then s' is not final and on every trace of M$'$ starting from s' a state s'_0 must be reached after some positive number of rule applications, that allows to complete the diagram with a state s_0, that can be reached by M with some (possibly zero) rule applications.

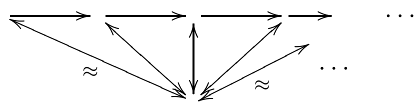

Figure 4: infinite chain of m:0 diagrams

Condition (5) is too general to establish refinement correctness: the reason are m:0 diagrams, that allow to refine an infinite trace of M with a trace of M′ that does no steps at all as shown in Fig. 4. This violates preservation of partial correctness.

To avoid this situation, we use a wellfounded order $<_{m0}$ that must decrease during an m:0 diagram. Dually, to avoid infinite chains of 0:n diagrams, that would violate preservation of total correctness (but are acceptable, when only preservation of partial correctness is required), we use an order $<_{0n}$. In the Prolog-WAM case study we found, that it is usually easy, to find a suitable ordering, when successive 0:n (or m:0) diagrams are possible: either M′ decreases the size of some data structure stored in its state, or it builds up a data structure already present in the similar state of M. Then the order is defined as the difference of these sizes. To be suitable for the latter case we define both orders to be predicates on two pairs of states: $<_{m0}, <_{0n} \subseteq (S \times S') \times (S \times S')$.

With these predicates our generic proof obligation for generalized forward simulation becomes:

$$\begin{array}{ll} & s \approx s' \land \neg\,(\text{final}(s) \land \text{final}'(s')) \\ \rightarrow & \text{EF+}(s, \lambda\, s_0.\ s_0 \approx s' \land (s_0,s') <_{m0} (s,s')\ ) \\ & \lor\ \text{AF}'\text{+}(s', \lambda\, s'_0.\ \text{EF+}(s, \lambda\, s_0.\ s_0 \approx s'_0) \lor s \approx s'_0 \land (s,s'_0) <_{0n} (s,s')) \end{array} \tag{VC}$$

The first disjunct requires $<_{m0}$ to decrease for m:0 diagrams, the second requires $<_{0n}$ to decrease, if the state s'_0 reached by M′ is immediately equivalent to the s (i.e. when we have a 0:n diagram). With this generic proof obligation, we can now prove the main result of this paper:

Theorem 7. [soundness of generalized forward simulation]
A refinement from M to M′ totally preserves traces, if

1. for each initial state $s' \in I'$ there is $s \in I$ with $s \approx s'$
2. verification condition (VC) holds
3. the coupling invariant $\approx$ is stronger than $\sim$ (i.e. $s \approx s' \rightarrow s \sim s'$ holds)

For partial preservation of traces, the condition that $<_{0n}$ decreases for 0:n diagrams may be dropped in (VC).

Since preservation of traces is stronger than preservation of correctness the theorem immediately implies

Corollary 8. If

1. for each initial state $s' \in I'$ there is $s \in I$ with both $s \approx s'$ and $s \sim_I s'$
2. verification condition (VC) holds
3. for final states s and s', $s \approx s'$ implies $s \sim_O s'$

then the refinement preserves partial and total correctness. For preservation of partial correctness, the condition that $<_{0n}$ decreases for 0:n diagrams may be dropped in (VC).

The proof of the theorem is based on the idea of composing diagrams. We will give the main lemmas here, full details are given in the appendix.

The proof starts by showing that the situation of Fig. 4 is now impossible. Wellfounded induction proves

Lemma 9. $s \approx s' \to AF'(s', \lambda\, s'_0.\ EF{+}(s, \lambda\, s_0.\ s_0 \approx s'_0) \vee final'(s'_0) \wedge s \approx s'_0)$

i.e. that M′ must progress to a state s'_0, that either allows to complete a diagram with a *positive* number of steps of M, or is final and similar to s.

Similarly, using wellfounded induction on $<_{0n}$ it can be proved that, if no m:n diagram with $n > 0$ is possible for $s \approx s'$, then, by adding m:0 diagrams, M will reach a state s_0, for which $s_0 \approx s'$ still holds, but which does not allow to add any more m:0 diagrams:

Lemma 10.

$$\begin{aligned} & s \approx s' \wedge \neg\, AF'{+}(s', \lambda\, s'_0.\ EF(s, \lambda\, s_0.\ s_0 \approx s'_0)) \\ \to\ & EF(s, \lambda\, s_0.\ s_0 \approx s' \wedge \neg\, EF{+}(s_0, \lambda\, s_1.\ s_1 \approx s')) \end{aligned}$$

Combining both lemmas, we get that by adding diagrams, both M and M′ will actually make progress in non-final states:

Lemma 11.

$$\begin{aligned} & s \approx s' \wedge \neg\, final(s) \wedge \neg\, final'(s') \\ \to\ & AF'{+}(s', \lambda\, s'_0.\ EF{+}(s, \lambda\, s_0.\ s_0 \approx s'_0)) \end{aligned}$$

Starting from $s \approx s'$ both systems will reach similar states s_0 and s'_0 with a positive number of steps. Intuitively, composing diagrams will result in an m:n diagram with both m,n > 0. If the condition, that $<_{0n}$ decreases is missing in (VC), (11) can be proved with AF′ replacing AF′+.

To complete the proof we now choose an arbitrary run σ' of M′. Then the first precondition of our theorem guarantees that initially $s \approx \sigma'(0)$ for some suitable initial state s of M. Iterated application of Lemma 11 guarantees that we can construct a trace σ and an infinite, strictly increasing sequence of state pairs on the traces σ and σ', which are similar with respect to $\approx$. Although the proof is intuitively simple — infinitely many m:n- diagrams with m,n > 0 are composed

— the technical details are actually quite complicated, since constructing a full trace σ of M with the required property from finite pieces must use the axiom of choice. Since adding a m:n diagram with $m > 0$ ($n > 0$) is not possible for a final state — EF+(...) resp. AF′+(...) are false for a final state — the trace σ' of M′ is finite iff σ is finite. This implies that the refinement totally preserves traces with respect to $\approx$. Since we have required $\approx$ to imply $\sim$ this completes the proof of the main theorem.

The main theorem implies, that our refinement technique results in an invariant of M′:

Theorem 12. If the proof obligations of Theorem 7 hold, the formula

$$\mathrm{INV}(s') :\leftrightarrow \exists\, s.\ s \in I \wedge \mathrm{AF}'(s', \lambda\, s'_0.\ \mathrm{EF}(s, \lambda\, s_0.\ s_0 \approx s'_0)) \tag{6}$$

is an invariant of M′ (i.e. each reachable state s′ of M′ satisfies INV).

Informally, the invariant says that from any reachable state s′ of M′ a state s'_0 can be reached that is similar to some reachable state s_0 of M. Intuitively, Theorem 12 follows from the main theorem just by stepping forward from s′ in the trace of M′ until a state is found, that completes a commuting diagram. The formal proof uses the same basic lemmas as the one of Theorem 7 in the appendix. It can be found in [Sch99]. Theorem 12 allows to use invariants in stepwise refinements: assume we have two refinements from M to M′ and from M′ to M″. Then after verification of the first refinement, we can use the invariant (6) (or any formula implied by it) as an additional precondition in the proof of the verification condition for the second refinement. In the Prolog-WAM case study, where 9 successive ASM refinements were verified, typically the size of the coupling invariant could be halved by using an invariant from the previous refinement. An immediate corollary of Theorem 12 is the following:

Corollary 13. If $\approx$ is a coupling invariant, that allows to verify the proof obligations of the main theorem, then relation $\simeq$, defined as

$$s \simeq s' :\leftrightarrow \mathrm{AF}'(s', \lambda\, s'_0.\ \mathrm{EF}(s, \lambda\, s_0.\ s_0 \approx s'_0))$$

is a coupling invariant too, that allows to verify preservation of traces using m:1 diagrams ($m \geq 0$) only.

The corollary shows, that in theory, we can avoid m:n diagrams with $n > 1$, but that this restriction may have the price of a more complex coupling invariant, that must talk about all intermediate states present in the next commuting diagram (these are the future states about which operators AF′ and EF make an assertion). Also using $\simeq$ instead of $\approx$ will result in a correctness proof that partly reiterates the generic proof of the main theorem. Our experience with the refinements of the Prolog-WAM case study shows, that when [BR95] proposed the use of m:n diagrams with m,n > 1, using m:1 diagrams instead, as was done in [Pus96], indeed complicates proofs.

5 ASM Refinements and Dynamic Logic

Verification condition (VC) talks (via AF and EF) about the transition relation ρ. This is inconvenient for the verification of ASM refinement, since it requires to encode the semantic transition relation of ASM rules explicitly into the syntax of the logic used for the verification. This is possible, and has been done in Isabelle [Pus96] and PVS [Dol98] using a tuple of (function) variables to represent a state (i.e. an algebra). But the encoding can be avoided by using Dynamic Logic (DL). Either the original definition of DL ([Har79]) can be used (using a data type of dynamic functions or using higher-order function variables), or variants of DL that directly deal with function updates (see e.g. [GR95],[Sch95],[SN01]).

DL extends predicate logic (first-order or higher-order does not matter) with two operators $[\alpha]\phi$ (read: "box α ϕ") and $\langle\alpha\rangle\phi$ (read: "diamond α ϕ") where α may be an imperative program or an ASM rule and ϕ may be a DL formula again. The informal meaning of the two formulas is "for all terminating executions of α the final state satisfies ϕ" and "α has a terminating execution such that ϕ holds afterwards". We will use an extended version, which also defines an operator $\langle\!|\alpha|\!\rangle\phi$ (read: "strong diamond α ϕ") that formalizes "all executions of α terminate in a state satisfying ϕ". This operator is not present in ordinary DL (but defined in the KIV system), since it requires to define the relational semantics of α with an explicit "bottom-state" $\perp$ to express non-termination. For imperative programs such a semantics is defined e.g. in [dRE98], ASM rules which may diverge (e.g. recursive rules as in [GS97], [BS00a] or [SN01]) can be given a similar semantics. For a first-order formula ϕ, $\langle\!|\alpha|\!\rangle\phi$ and $[\alpha]\phi$) are just another way to denote the weakest resp. weakest liberal precondition of α with respect to ϕ.

In the following we will consider (ordinary) ASM rules first, which always terminate. In this case the AF and EF operator are equivalent to the termination of a while-loop which checks after each rule application, whether p is true:

$$\mathrm{EF}(s,p) \leftrightarrow \langle \textbf{while}\ \neg\ p(s) \wedge \neg\ \mathrm{final}(s)\ \textbf{do}\ \mathrm{RULE}\rangle p(s)$$
$$\mathrm{AF}(s,p) \leftrightarrow \langle\!| \textbf{while}\ \neg\ p(s) \wedge \neg\ \mathrm{final}(s)\ \textbf{do}\ \mathrm{RULE}|\!\rangle p(s)$$

Note that while the abbreviation EF(s,p) mentions the full state s, there is no need for the DL formula to do so: predicates p and final as well as the ASM rule RULE can just access the relevant dynamic functions of the algebra s.

An important special case is finite nondetermism. Distributed ASMs as defined in [Gur95] ony have finite nondeterminism, since the agent to apply a rule is always chosen from a finite set of active ones. In this case, EF and AF become repeated rule application:

$$\begin{array}{l} \mathrm{EF(s,p)} \leftrightarrow \exists\ \mathrm{n.}\ \langle(\textbf{if}\ \neg\ \mathrm{p(s)}\ \textbf{then}\ \mathrm{RULE})^{\mathrm{n}}\rangle \mathrm{p(s)} \\ \mathrm{AF(s,p)} \leftrightarrow \exists\ \mathrm{n.}\ [(\textbf{if}\ \neg\ \mathrm{p(s)}\ \textbf{then}\ \mathrm{RULE})^{\mathrm{n}}] \mathrm{p(s)} \end{array} \tag{7}$$

To avoid an additional check, we have assumed that calling RULE in a final state does nothing. The strong diamond operator is no longer needed, since termination of the whole loop is no longer an issue. Iteration of rules becomes a kind of "for" loop. Such loops are already used in DL to axiomatize while loops. If the ASM is deterministic, or if the number of steps necessary to complete a commuting diagram is independent of the trace chosen, the check for "$\neg$ p(s)" in (7) may be dropped.

In practical applications, verification of (VC) will usually split into several cases, one for each type of diagram. In this case the coupling invariant $\approx$ is equivalent to a disjunction. Each disjunct $\approx_{\mathrm{k}}$ describes one situation, where it is possible to add a specific next diagram. For two standard ASMs, a typical situation has the form $\approx \wedge\ \epsilon_i \wedge \epsilon'_j$ (k ranges over all possible pairs i,j) which means that it fixes RULE_i and RULE'_j as the first rules of the commuting diagram.

If the diagram for case k (i.e. s $\approx_{\mathrm{k}}$ s$'$ holds for states s,s$'$, not both final) has a fixed size of $\mathrm{m_k}$ rule applications of M and $\mathrm{n_k}$ rule applications of M$'$, then the quantifiers in (7) can be instantiated and it is sufficient to prove one the three conditions:

$$\begin{array}{l} \underbrace{[\mathrm{RULE'}]\ldots[\mathrm{RULE'}]}_{\mathrm{n_k\ times}}\underbrace{\langle\mathrm{RULE}\rangle\ldots\langle\mathrm{RULE}\rangle}_{\mathrm{m_k\ times}}\ \mathrm{s} \approx \mathrm{s'} \\ \mathrm{s'} = \mathrm{s'_0} \rightarrow \underbrace{[\mathrm{RULE'}]\ldots[\mathrm{RULE'}]}_{\mathrm{n_k\ times}}(\mathrm{s} \approx \mathrm{s'} \wedge (\mathrm{s,s'}) <_{\mathrm{0n}} (\mathrm{s,s'_0})) \\ \mathrm{s} = \mathrm{s_0} \rightarrow \underbrace{\langle\mathrm{RULE}\rangle\ldots\langle\mathrm{RULE}\rangle}_{\mathrm{m_k\ times}}(\mathrm{s} \approx \mathrm{s'} \wedge (\mathrm{s,s'}) <_{\mathrm{m0}} (\mathrm{s_0,s'})) \end{array} \tag{8}$$

Proof obligations become more complex, if we consider ASM rules which may fail to terminate, since equivalence (7) does no longer hold, and strong diamonds can no longer be replaced with boxes. To deal with this case, we have to use a state space, which includes a $\perp$ element for nontermination. The transition relation ρ and (VC) then *explicitly* mention nontermination, e.g. ρ(s,$\perp$) will hold for states s, where the system may diverge. To use Dynamic Logic in this case we have to transform formula (VC) equivalently into a formula, which talks about nontermination only *implicitly* via the diamond-operator. For deterministic ASMs the resulting condition is

$$
\begin{array}{ll}
& s \approx s' \wedge s = s_0 \wedge s' = s'_0 \wedge \neg\,(\mathrm{final}(s) \wedge \mathrm{final}'(s')) \\
\rightarrow & \quad \exists\, m, n.\ \langle(\mathrm{RULE})^m\rangle\langle(\mathrm{RULE}')^n\rangle s \approx s' \\
& \vee\ \exists\, m.\ \langle(\mathrm{RULE}')^n\rangle(s \approx s' \wedge (s_0,s') <_{m0} (s,s')) \\
& \vee\ \exists\, n.\ \langle(\mathrm{RULE})^n\rangle(s \approx s' \wedge (s,s'_0) <_{0n} (s,s')) \\
& \vee\ \exists\, m, n.\ \langle(\mathrm{RULE})^m\rangle \mathrm{diverges}(s) \wedge \langle(\mathrm{RULE}')^n\rangle \mathrm{diverges}(s')
\end{array}
\tag{9}
$$

The states s,s′ etc. mentioned in (9) can *not* be $\perp$, divergence of rules is encoded only in the abbreviation *diverges(s)*, which expands to "$\neg\ \langle\mathrm{RULE}\rangle$true". Compared to always terminating rules we now have an extra case: alternatively to showing that an m:n diagram follows (with the appropriate checks for m,n = 0), both ASMs may diverge. Note that the condition for the deterministic case is fully symmetric in ASM and ASM′, corresponding to the fact, that preservation of total correctness and total preservation of traces are symmetric too in this case: a correct refinement of deterministic ASMs is always complete.

The most problematic case are rules which may nondeterministically diverge. If the application of such a rule is allowed only as the first rule of a diagram we proved that the condition

$$
\begin{array}{l}
\quad s \approx s' \wedge \neg\,(\mathrm{final}(s) \wedge \mathrm{final}'(s')) \\
\rightarrow \quad \mathrm{EF}{+}(s, \lambda\, s_0.\ s_0 \approx s' \wedge (s_0,s') <_{m0} (s,s'))) \\
\quad \vee \quad \neg\ \mathrm{final}'(s') \\
\qquad \wedge\ (\mathrm{maydiv}(s') \rightarrow \mathrm{EF}(s, \lambda\, s_0.\ \neg\ \mathrm{final}'(s_0) \wedge \mathrm{maydiv}(s_0))) \\
\qquad \wedge\ [\mathrm{RULE}'(s')] \\
\qquad\qquad \mathrm{AF}(s', \lambda\, s'_1. \quad \mathrm{EF}{+}(s, \lambda\, s_0.\ s_0 \approx s'_1) \\
\qquad\qquad\qquad\qquad \vee\ \mathrm{inv}(s,s'_1) \wedge (s,s'_1) <_{0n} (s,s'_0) \\
\qquad\qquad\qquad\qquad \vee \quad \mathrm{diverges}(s'_1) \wedge \neg\ \mathrm{final}'(s'_1) \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{EF}(s, \lambda\, s_0.\ \neg\ \mathrm{final}(s_0) \wedge \mathrm{maydiv}(s_0)))
\end{array}
\tag{10}
$$

is equivalent to the original condition (VC). The proof obligation now uses (7) to define AF and EF as abbreviations. *maydiv(s)* abbreviates the potential for divergence, formalized as "$\neg\ \langle\!|\mathrm{RULE}|\!\rangle$true". The second disjunct, which considers m:n diagrams with $n > 0$ (so s′ is not final) now handles diverging rules: If the first rule application of M′ may diverge, then M must be able to reach a nonfinal state s_0, where it may diverge too. If the first rule application of M′ terminates, it must be possible to reach a state s'_1 which satisfies one of the following two conditions: either s'_1 allows to complete an m:n diagram (with $<_{0n}$ decreasing, when $m = 0$) or the next rule of M′ applied on s'_1 surely diverges, and M has a nonterminating trace from s.

Note that condition (10) is needed only for those cases, where the diagram under consideration executes nondeterministic, potentially diverging rules. If no such rules are involved, the condition simplifies back to the simpler conditions mentioned before.

6 Related Work

6.1 Data Refinement

Data refinement considers refinement of a set of (abstract) operations by another set of (concrete) operations. A refinement is correct, if for *every* program it is possible to replace the abstract operations with the concrete ones, without changing its "meaning". The meaning of a program is usually described using initialization and finalization operations, or using equivalence relations $\sim_I$ and $\sim_O$ as we have done. Correctness is proved by defining an equivalence relation $\approx$ (often a functional relation, called an "abstraction function" is used) and verification of a 1:1 diagram for each pair of corresponding operations. The actual commutativity proofs may either show, that if $\approx$ holds before the operations, then it will hold afterwards (forward simulation) or they may show, that if $\approx$ holds after the operations, then it will hold before them (backward simulation). A good introduction to data refinement is [dRE98], which also discusses the instances used in the specification languages VDM [Jon90] and Z [Spi88].

It is possible to view ASM refinement as an instance of data refinement, using the two ASMs as monolithic operations. In this case, our definitions of preservation of partial and total correctness coincide with the ones in [dRE98].

On the other hand, it is more interesting to view data refinement as a special case of ASM refinement. This is possible when operations have a relational semantics and are therefore expressible as ASM rules (we do not consider predicate transformer semantics, as defined e.g. in [GM91]).

To simulate the behaviour of every possible program over a data type D, we define an abstract state machine ASM(D), which randomly executes in each state one of the available operations or terminates. Given two data types D and D′, the proof obligations for data refinement then coincide with the special case of 1:1 diagrams for ASM(D) and ASM(D′). Therefore data refinement preserves total resp. partial correctness if forward resp. backward simulation can be shown. Data refinement is more restrictive than ASM refinement: Since a concrete run should simulate an abstract run which calls *the same* operations, it must also preserve the 1:1 correspondence between operations. ASM refinement does not have such a requirement, although for standard ASMs some correspondence according to the form of the diagrams used often can be found.

Recently, a number of definitions of data refinement have been given, that weaken the 1:1 conditions for rule correspondence. An extensive overview is given in [DB01], which also discusses applications to Z and Object-Z specifications: "Non-atomic refinement" allows to refine one abstract operation by a fixed sequence of $n > 0$ concrete operations and requires verification of an 1:n diagram. "Alphabet translation" allows to implement different cases of one abstract operation by different concrete operations. Other generalizations classify operations as external and internal operations, and require a 1:1 correspondence for external operations only. Programs are constrained to execute a finite number of internal operations (assumed to be invisible) between any two external ones. This constraint is assumed for the abstract data type and must be enforced for the concrete one using a well-founded order, that decreases for internal operations. Two approaches for verification are defined: the first uses "stuttering steps" and proves in addition to the commutation of 1:1 diagrams for external operations that 1:0 and 0:1 diagrams commute for internal operations. The second, more liberal approach, called "weak refinement", requires to verify all m:n diagrams, where each data type executes one external operation in between two arbitrary sequences of internal operations.

For each of these generalizations we have shown, that the verification conditions are instances of our generic verification condition (VC): Alphabet translation and stuttering steps are already accommodated by our theory. Non-atomic refinement requires to group the implementing sequence of concrete operations into one ASM rule. Weak refinement requires a small modification to Theorem 7: The first condition has to be weakened to

$$\mathrm{s} \in \mathrm{I} \wedge \mathrm{s}' \in \mathrm{I}' \rightarrow \mathrm{AF}'(\mathrm{s}', \lambda\, \mathrm{s}'_0.\ \mathrm{EF}(\mathrm{s}, \lambda\, \mathrm{s}_0.\ \mathrm{s}_0 \approx \mathrm{s}'_0)) \tag{11}$$

This allows the first commuting diagram not to start in two initial states, but only after some steps (which are internal steps in the case of weak refinement).

6.2 Refinement of I/O Automata

Even more similar to ASM refinement than data refinement is the refinement of I/O automata [LT89],[LV95]. I/O automata are state transition systems similar to ASMs, but their state transitions are labelled with *actions*. Formally an I/O automaton modifies Definition 1 by requiring: $\rho \subseteq \mathrm{S} \times \mathrm{A} \times \mathrm{S}$, where A is a set of actions. Actions are classified as input and output actions. Input actions should be possible in any state, since they are assumed to be under the control of the environment. The set of actions always contains an "empty" (also called "stuttering" or "internal") action τ, which signifies no in- or output. Runs of an automaton consist of finite or infinite sequences $(\mathrm{s}_0, \mathrm{a}_0, \mathrm{s}_1, \mathrm{a}_1, \mathrm{s}_2, \ldots)$, such that $\mathrm{s}_0 \in \mathrm{I}$ and $(\mathrm{s}_\mathrm{i}, \mathrm{a}_\mathrm{i}, \mathrm{s}_{\mathrm{i}+1}) \in \rho$ for every i. A trace of an automaton extracts the nonempty actions from a run: $(\mathrm{a}_0, \mathrm{a}_1, \mathrm{a}_2, \ldots) \setminus \tau$. Refinement is defined as the inclusion relation between the sets of traces of two automata.

The refinement notion between I/O automata can be viewed as an instance of our preservation of traces as follows: define the state of the transition system to be a pair of an automaton state and a list al of actions done so far (i.e. we record the actions in a history variable). The ASM starts with an empty action list, and has a transition from (s,al) to $(\mathrm{s}_0,\mathrm{a}{:}\mathrm{al})$, when the automaton had one

$$
\mathrm{ASM}: \quad
\begin{tikzcd}[row sep=small]
 & s_{1a} \arrow[r] & s_{2a} \\
s_0 \arrow[ur] \arrow[dr] & & \\
 & s_{1b} \arrow[r] & s_{2b}
\end{tikzcd}
\qquad\qquad
\mathrm{ASM}': \quad
\begin{tikzcd}[row sep=small]
 & & s'_{2a} \\
s'_0 \arrow[r] & s'_1 \arrow[ur] \arrow[dr] & \\
 & & s'_{2b}
\end{tikzcd}
$$

Figure 5: refinement that can be verified with generalized forward simulation, but not with forward simulation

from s to s_0 labelled with the nonempty action a. For the empty action τ, the action list remains unchanged. We have proved that refinement of I/O automata ($\leq_T$ in [LV95]) corresponds to the partial preservation of traces, with a similarity relation that requires equality of action lists. Refinement of I/O automata allows the refinement of a finite run with an infinite run, when both runs have no output (i.e. only τ actions), so it does not preserve total correctness. Note, that preservation of partial correctness (with $\sim_O$ defined again to be equality of action lists), corresponds to the subset relation on finite traces ($\leq_{*T}$ in [LV95]).

Actions of I/O automata correspond closely to the modification of input and output functions in ASMs. To prove, that two ASMs modify an output function in the same way also requires a history variable, which records the modifications of the output function, before commuting diagrams can be verified using (VC).

A forward simulation on I/O automata is defined as a relation $\approx$, that allows to verify m:1 diagrams. If the action of the concrete automaton M' is the nonempty action a, then exactly one of the $m > 0$ actions of the abstract automaton M must be nonempty and equal to a. Otherwise, m may be zero and all actions of M must be empty.

Our proof technique for ASM refinement generalizes forward simulation of I/O automata refinement in two ways:

1. It allows to verify m:n diagrams, which execute more than one nonempty action. As a consequence, backward simulation can be avoided in many cases: the standard example of Fig. 5, which can not be verified using forward simulation only (assuming $s_0 \sim_I s'_0$, $s_{2a} \sim_O s'_{2a}$ and $s_{2b} \sim_O s'_{2b}$) can be verified using a 2:2 diagram and generalized forward simulation. Many other examples (e.g. the ones in [DB01]) are similar. Nevertheless, there are still examples which require backward simulation: one is given in Fig. 6, assuming $s_{1a} \sim_I s'_1$, $s_{1b} \sim_I s'_1$, $s_2 \sim_O s'_{2a}$ and $s_{2b} \sim_O s'_{2b}$. Whether such examples are relevant in practice is a topic for further research.

2. Preservation of traces may be used to verify refinements which implement the actions themselves: If the abstract level gives each output as a 16-bit word, but the concrete level does two actions, each giving one byte, this is easy to accommodate in our framework (just use a slightly modified similarity relation), while it is not possible in a framework, where the abstract and concrete level are

$$\mathrm{ASM}: \quad \begin{array}{l} s_{1a} \longrightarrow s_{2a} \\[1em] s_{1b} \longrightarrow s_{2b} \end{array} \qquad\qquad \mathrm{ASM}': \quad \begin{tikzcd}[row sep=tiny] & {s'}_{2a} \\ {s'}_1 \arrow[ur] \arrow[dr] & \\ & {s'}_{2b} \end{tikzcd}$$

Figure 6: refinement that cannot be verified with generalized forward simulation, but with backward simulation

$$\begin{array}{ll} & s_{0,0} \\ & s_{1,0} \longrightarrow s_{1,1} \\ \mathrm{ASM}: & s_{2,0} \longrightarrow s_{2,1} \longrightarrow s_{2,2} \\ & \;\vdots \qquad\quad \vdots \qquad\quad \ddots \\[1em] \mathrm{ASM}': & s'_0 \longrightarrow s'_1 \longrightarrow s'_2 \longrightarrow \cdots \end{array}$$

Figure 7: refinement, which only preserves partial correctness

required to have the *same* set of actions.

Note that our generalization of forward simulation is nontrivial, since the reduction to m:1 diagrams given in Corollary 13 uses a relation $\simeq$, which may hold between states which have done *different* actions, so $\simeq$ is not a forward simulation in this case.

Our refinement notion also generalizes the approach of normed simulations [GV98] for I/O automata, which uses 0:1, 1:1, and 1:0 diagrams. The "norm" is equivalent to our wellfounded order $<_{m0}$. As an interesting aside, [GV98] wants diagrams to be as small as possible to enable the use of automatic model checkers (which cannot compute the m and n of arbitrary diagrams), while we found, that interactive proofs usually become simpler when diagrams are chosen to be as large as possible.

6.3 (Generalized) Backward Simulation

Just like data refinement, simulation theory for I/O automata considers in addition to forward simulations also backward simulations, which propagate the coupling invariant backward through traces. Backward simulation without restrictions only preserves partial correctness. Fig. 7 shows, that nothing more can be achieved in general, since $\approx \, := \{(s_{i,j}, s'_j) : j \geq i\}$ is a backward simulation (assuming, that each transition has the same, nonempty action).

Under the restriction that the relation $\approx$ is total and image-finite, i.e. that the set $\{s\colon s \approx s'\}$ is nonempty and finite for each state s' of M' (the finitary relations in [dRE98] capture the same requirement), backward simulation implies partial preservation of traces for I/O automata and preservation of total correctness for data refinement. Backward simulation is mainly of theoretical interest: combined with forward simulation completeness results can be proved.

Some of these results are also applicable in our setting: generalized backward simulation can be defined, simply by reverting the direction of the transition relation, exchanging the role of initial and final states and using dual past tense operators instead of AF and EF. We have defined this dualization schematically in KIV. We found, that some care has to be taken, when final and initial states are exchanged: since we have defined final states to have no successor states, we must now require that initial states have no predecessor states. This is not always true, but it is simple, to modify an ASM such that this condition is satisfied. Then using the dualized theorems of generalized forward simulation, the proof that generalized backward simulation implies preservation of partial correctness is simple. Unfortunately, the condition of image-finiteness is not sufficient for generalized backward simulation to imply the three stronger notions of refinement correctness: again, Fig. 7 provides an example using $\approx := \{(s_{i,0}, s'_0) : i \geq 0\} \cup \{(s_{j,j}, s'_j) : j > 0\}$. A sufficient additional requirement seems to be the existence of a uniform upper bound to the size n of m:n diagrams, but an exact criterion and a formal proof are yet to be done.

Whether there are cases which satisfy these additional restrictions also remains an open question. Compiler verification does not seem to be a suitable candidate: Fig. 7 suggests, that image-finiteness of $\sim_I$ is a necessary criterion for (generalized) backward simulation to imply preservation of total correctness, but typically infinitely many programs will be compiled to the same target code.

6.4 Compiler verification

For general work on compiler verification and on verification of Prolog compilers, we like to refer to [Sch99], since this could easily fill several more pages. 1:n diagrams with $n > 0$ often occur in compiler verification, when one source code instruction is replaced with several target instructions. They are discussed in many variants, e.g. in [BHMY89] and in [Cyr93]. The second paper also describes a proof technique called "slowing down the specification machine", which splits 1:n diagrams in one 1:1 diagram and $n-1$ 0:1 diagrams. While the "termination condition" is equivalent to decreasing the $<_{0n}$ predicate, the approach requires to add explicit time to the ASMs.

m:n diagrams with positive m,n were already sketched for a special case of coupling invariants in [McG72]. A formal treatment of this case for deterministic ASMs and an abstraction function in PVS is described in [Dol98].

The refinement theory described in this paper generalizes the theory developed in [Sch99] to rules, which may diverge and simplifies some of the notation. For the verification condition, it avoids the use of a function, that predicts whether a 0:m, a m:0 or a m:n diagram will follow.

[Sch99] describes formal application of the theory to 9 refinements specifying a Prolog compiler. Two refinements involve diagrams with a size that depends on the size of datastructures stored in the state (here: the number of Prolog clauses reachable in a code fragment). Two other examples of ASM refinements which make informal use of our theory are [BS00b] and [SSB01]. Both address correct compilation of Java to the JVM. [BS00b] uses an m:n diagram for the case, when an exception is thrown. The size m of the diagram is (roughly) the number of Java statements, that must be jumped over, until the next exception handler is found. Properties of the final states of the diagram are established using an auxiliary lemma, which is typical for proofs using m:n diagrams where m and n are dependent on data structures ([Sch99] also uses such lemmas).

[SSB01] gives another refinement proof for a revised version of the Java ASM. Each case of the proof corresponds to the verification of one 1:n diagram with n = 0,1,2 or 3. The size n of the diagrams is computed as n := σ(m+1) $-$ σ(m), where σ(m) counts the number of steps of the JVM, that are necessary to reach a state equivalent to the m^{th} state of the Java ASM. The cases of 1:0 diagrams correspond to navigation steps which are present in the Java ASM, but avoided in the JVM: either the Java ASM propagates a result (or an exception) upwards or the Java ASM descends into an expression. Infinite repetition of such steps (i.e. of 0:1 diagrams) is impossible, because both kinds of steps reduce the (finite) size of the remaining program to be executed.

7 Conclusion

We have defined a generic framework to verify the refinement of ASMs. Four notions of refinement were defined, and a generic proof technique given, that reduces the correctness proof to the verification of m:n diagrams with m, n $\geq$ 0. Our proof technique combines and generalizes proof techniques in use in the areas of data refinement, I/O automata refinement, and in compiler verification.

The proof technique has been successfully used in [Sch99], a large case study on compiler verification. Verification of some of the refinements would not have been practically possible without the theory we have developed. Using 0:n, m:0, and m:n diagrams with m,n $>$ 1 (instead of using 1:n or m:1 diagrams only) simplified many proofs considerably.

Currently, we have used generalized forward simulation mainly in applications, which use deterministic ASMs. An interesting topic for further research is to apply the theory to case studies which involve nondeterministic ASMs. These pose new questions, such as how to add fairness (and other) constraints to the choices made by the systems. Since such constraints are not directly expressible in Dynamic Logic, we will have to decide between two approaches: Either we can encode the indeterministic choices of ASMs as input streams, i.e. use a static function f, such that f(n) gives the n^{th} choice, and place explicit constraints on f. This construction is also known as "adding prophecy variables" and strongly related to the "construction of the canonical automaton" (see [LV95]). It is proposed in [SN01] for ASMs and allows to stay within Dynamic Logic. With this approach implicit quantification over all runs of an ASM becomes explicit quantification over all possible input streams present in the initial state. Alternatively, we can use temporal logic, e.g. extend the operators AF, EF to a full temporal logic similar to CTL. Refinement of TLA specifications [AL91] is related work that should be considered then.

Acknowledgements

I would like to thank Prof. Börger and 3 anonymous referees for pointing out several errors and for making many helpful suggestions on drafts of this paper.

References

[AL91] 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.

[BHMY89] W. R. Bevier, W. A. Jr. Hunt, J S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[BM96] E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1996.

[BR95] E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In Christoph Beierle and Lutz Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*. North-Holland, Amsterdam, 1995.

[BS98] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*, page 17ff. Springer, 1998.

[BS00a] E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Proc. 14th International Workshop Computer Science Logic (Gurevich Festschrift)*, LNCS 1862, pages 41–60. Springer, 2000.

[BS00b] E. Börger and W. Schulte. A Practical Method for Specification and Analysis Exception Handling - A Java/JVM Case Study. *Journal of Software Engineeering*, 26(9):872–887, September 2000.

[Cyr93] D. Cyrluk. Microprocessor Verification in PVS: A Methodology and Simple Example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, December 1993.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. FACIT. Springer, 2001.

[Dol98] A. Dold. A Formal Representation of Abstract State Machines using PVS. Verifix-Report Ulm 6.2, Universität Ulm, 1998. (revised version).

[dRE98] 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, New York, NY, 1998.

[GDG+96] W. Goerigk, A. Dold, Th. Gaul, G. Goos, A. Heberle, F.W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Rueß, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. Technical Report LiTH-IDA-R-96-12, Linkoping University, 1996.

[GM91] P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87, 1991.

[GR95] R. Groenboom and G. Renardel de Lavalette. A Formalization of Evolving Algebras. In *Proceedings of Accolade95*. Dutch Research School in Logic, 1995.

[GS97] Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *Journal of Universal Computer Science*, 3(4):233–246, 1997.

[Gur95] M. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[GV98] 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.

[Har79] D. Harel. *First Order Dynamic Logic*. LNCS 68. Springer, Berlin, 1979.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *COMM ACM*, pages 576–580, 1969.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 2nd edition, 1990.

[LT89] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly,Centrum voor Wiskunde en Informatica, Amsterdam*, 1989.

[LV95] 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.

[McG72] C.L. McGowan. An inductive proof technique for interpreter equivalence. In R. Rustin, editor, *Formal Semantics of Programming Languages*, pages 139 – 147. Englewood Cliffs, 1972.

[Moo88] J Moore. PITON: A Verified Assembly Level Language. Technical report 22, Computational Logic Inc., 1988. available at the URL: http://www.cli.com.

[Pus96] C. Pusch. Verification of compiler correctness for the WAM. In J.Harrison J. von Wright, J.Grundy, editor, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 347–362. Springer, 1996.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[RSSB98] 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*. Kluwer Academic Publishers, Dordrecht, 1998.

[SA97] 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. available at http://hyperg.iicm.tu-graz.ac.at/jucs/.

[SA98] 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. Kluwer Academic Publishers, 1998.

[Sch95] A. Schönegge. Extending Dynamic Logic for Reasoning about Evolving Algebras. Technical Report 49/95, Fakultät für Informatik, Universität Karlsruhe, 76128 Karlsruhe, Germany, 1995.

[Sch99] G. Schellhorn. *Verification of Abstract State Machines.* PhD thesis, Universität Ulm, Fakultät für Informatik, 1999. (available at `www.informatik.uni-augsburg.de/swt/fmg/papers/`).

[SN01] R. F. Stärk and S. Nanchen. A Complete Logic for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, Abstract State Machines 2001: Theory and Applications, 2001. (this volume).

[Spi88] J. M. Spivey. *Understanding Z.* Cambridge University Press, 1988.

[SSB01] R.F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001.

A Appendix: Proof of the Main Theorem

This appendix gives the formal proof for Theorem 7 of Sect. 4, which consists of five steps described in the following subsections. The first three steps prove lemmas 9, 10 and 11. The fourth step defines an auxiliary lemma that makes the axiom of choice applicable, and the final step proves the main theorem. All proofs use recursions (2) to unfold the definitions of EF and AF, (4) to eliminate AF′+ and EF+ in favor of AF′ and EF, and the basic lemmas

$$\mathrm{EF}(s,p) \leftrightarrow \mathrm{EF}(s,\lambda\, s_1.\ s_1 = s_0) \wedge p(s_0) \tag{12}$$

$$\begin{aligned}&(\forall\, s_0.\ \mathrm{EF}(s,\lambda\, s_1.\ s_1 = s_0) \wedge p_0(s_0) \rightarrow \mathrm{EF}(s_0,p_1))\\ \rightarrow\ &(\mathrm{EF}(s,p_0) \rightarrow \mathrm{EF}(s,p_1))\end{aligned} \tag{EF-step}$$

$$\begin{aligned}&(\forall\, s_0.\ \mathrm{EF}(s,\lambda\, s_1.\ s_1 = s_0) \wedge p_0(s_0) \rightarrow \mathrm{AF}(s_0,p_1))\\ \rightarrow\ &(\mathrm{AF}(s,p_0) \rightarrow \mathrm{AF}(s,p_1))\end{aligned} \tag{AF-step}$$

The definitions (1) of AF and EF are not used in the first three steps, except in the simple proofs of the three lemmas above. Lemma (12) says, that if a state is reachable from s, where p holds, we can give it a name s_0. Lemma (EF-step) says, that if we have to prove the implication "$\mathrm{EF}(s,p_0) \rightarrow \mathrm{EF}(s,p_1)$" then assuming the state s_0 where p_0 holds, is not reached later than the one, in which p_1 holds, it is sufficient to show "$\mathrm{EF}(s,\lambda\, s_1.\ s_1 = s_0) \wedge p_0(s_0) \rightarrow \mathrm{EF}(s_0,p_1)$". The first precondition ("s_0 reachable from s") is usually dropped. This means, that by applying the lemma to reduce a goal we „step forward" from state s to state s_0 in the trace. Stepping from s to s_0 also works for AF instead of EF (now assuming, that *all* states, where p_0 holds are before those, where p_1 holds). Since we use AF′ and EF to express commutativity of diagrams, application of (EF-step) for M and (AF-step) for M′ will step forward in the traces and thereby reduce the size of the diagrams. Special cases of (EF-step) and (AF-step), which will often use are

$$\begin{aligned}&(\forall\ s_0.\ EF(s,\lambda\ s_1.\ s_1 = s_0) \land p_0(s_0) \to p_1(s_0))\\ \to\ &(EF(s,p_0) \to EF(s,p_1))\end{aligned} \tag{EF-imp}$$

$$\begin{aligned}&(\forall\ s_0.\ EF(s,\lambda\ s_1.\ s_1 = s_0) \land p_0(s_0) \to p_1(s_0))\\ \to\ &(AF(s,p_0) \to AF(s,p_1))\end{aligned} \tag{AF-imp}$$

i.e. we have $s_0 = s_1$ and p_0 must immediately imply p_1.

A.1 Proof of Lemma 9

The lemma is proved by induction over (s,s′), using $<_{0n}$ as the wellfounded order. Application of (VC) reduces the goal to three subgoals. The first subgoal deals with the case, where the precondition of (VC) does not hold, i.e. when we have two final states s and s′. The second and third case deal with the two disjuncts of the conclusion of (VC). The first two cases are trivially proven by unfolding the definition of AF′ with (2). In the third case, after eliminating AF′+ with (4) we have to show

$$\begin{aligned}&\quad s \approx s' \land \neg\ final'(s') \land \text{Ind-Hyp}\\ &\land \forall\ s'_0.\ \rho'(s', s'_0) \to AF'(s'_0, \lambda\ s'_1.\quad s \approx s'_1 \land (s, s'_1) <_{0n} (s, s')\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lor EF{+}(s,\lambda\ s_0.\ s_0 \approx s'_0))\\ \to\ &AF'(s', \lambda\ s'_0.\ s \approx s'_0 \land final'(s'_0) \lor EF{+}(s, \lambda\ s_0.\ inv(s_0, s'_0)))\end{aligned}$$

where Ind-Hyp is the induction hypothesis. By unfolding AF′ in the conclusion we get a state s'_0 with $\rho'(s', s'_0)$. This state can be used to instantiate the quantifier in the precondition to give

$$\begin{aligned}&\quad s \approx s' \land \neg\ final'(s') \land \rho'(s', s'_0) \land \text{Ind-Hyp}\\ &\land AF'(s'_0, \lambda\ s'_1.\quad s \approx s'_1 \land (s, s'_1) <_{0n} (s, s')\\ &\qquad\qquad\qquad\qquad \lor EF{+}(s,\lambda\ s_0.\ s_0 \approx s'_1))\\ \to\ &AF'(s'_0, \lambda\ s'_1.\ s \approx s'_1 \land final(s'_1) \lor EF{+}(s, \lambda\ s_0.\ inv(s_0, s'_1)))\end{aligned}$$

Now we apply Lemma (AF-step) on the the two AF-formulae, stepping from s'_0 to the state s'_1 at the end of the 0:n-diagram. After removing now irrelevant formulae the remaining goal is:

$$\begin{aligned}&\quad \text{Ind-Hyp} \land (EF{+}(s, \lambda\ s_0.\ s_0 \approx s'_1) \lor s \approx s'_1 \land (s, s'_1) <_{0n} (s, s'))\\ \to\ &AF'(s'_1, \lambda\ s'_2.\ s \approx s'_2 \land final(s'_2) \lor EF{+}(s, \lambda\ s_0.\ s_0 \approx s'_2)),\end{aligned}$$

The disjunct in the precondition gives two cases to prove. The first is proved by unfolding AF, the second directly follows from the induction hypothesis.

A.2 Proof of Lemma 10

This lemma is also proved by wellfounded induction using $<_{m0}$. Applying (VC) results in three cases as before. The case of final states is trivial again. The third case is simple too, although technically more involved, since we have to unfold AF′+ and EF+ before (AF-imp) and (EF-imp) are applicable. In the second case, after unfolding EF+

$$\begin{aligned} &\mathrm{s} \approx \mathrm{s}' \wedge \rho(\mathrm{s}, \mathrm{s}_0) \wedge \neg\, \mathrm{final}(\mathrm{s}) \wedge \text{Ind-Hyp} \\ &\wedge \mathrm{EF}(\mathrm{s}_0, \lambda\, \mathrm{s}_2.\, \mathrm{s}_2 \approx \mathrm{s}' \wedge (\mathrm{s}_2, \mathrm{s}') <_{m0} (\mathrm{s}, \mathrm{s}')) \\ &\wedge \neg\, \mathrm{AF}'{+}(\mathrm{s}', \lambda\, \mathrm{s}'_0.\, \mathrm{EF}(\mathrm{s}, \lambda\, \mathrm{s}_0.\, \mathrm{s}_0 \approx \mathrm{s}'_0)) \\ \rightarrow\ &\mathrm{EF}(\mathrm{s}, \lambda\, \mathrm{s}_0.\, \mathrm{s}_0 \approx \mathrm{s}' \wedge \neg\, \mathrm{EF}{+}(\mathrm{s}_0, \lambda\, \mathrm{s}_1.\, \mathrm{s}_1 \approx \mathrm{s}')) \end{aligned}$$

has to be shown. The proof proceeds by unfolding EF in the conclusion and by instantiating the resulting quantifier with s_0. Using lemma (EF-step) to step from s_0 to s_2 (exceptionally keeping the reachability precondition) leads to

$$\begin{aligned} &\mathrm{s} \approx \mathrm{s}' \wedge \rho(\mathrm{s}, \mathrm{s}_0) \wedge \neg\, \mathrm{final}(\mathrm{s}) \wedge \text{Ind-Hyp} \\ &\wedge \mathrm{EF}(\mathrm{s}_0, \lambda\, \mathrm{s}_1.\, \mathrm{s}_1 = \mathrm{s}_2) \wedge \mathrm{s}_2 \approx \mathrm{s}' \wedge (\mathrm{s}_2, \mathrm{s}') <_{m0} (\mathrm{s}, \mathrm{s}') \\ &\wedge \neg\, \mathrm{AF}'{+}(\mathrm{s}', \lambda\, \mathrm{s}'_0.\, \mathrm{EF}(\mathrm{s}, \lambda\, \mathrm{s}_0.\, \mathrm{s}_0 \approx \mathrm{s}'_0)) \\ \rightarrow\ &\mathrm{EF}(\mathrm{s}_2, \lambda\, \mathrm{s}_0.\, \mathrm{s}_0 \approx \mathrm{s}' \wedge \neg\, \mathrm{EF}{+}(\mathrm{s}_0, \lambda\, \mathrm{s}_1.\, \mathrm{s}_1 \approx \mathrm{s}')) \end{aligned}$$

Since $(\mathrm{s}_2, \mathrm{s}') <_{m0} (\mathrm{s}, \mathrm{s}')$ we can now apply the induction hypothesis. Its conclusion (i.e. the postcondition of Lemma 10 with s_2 instead of s) is equal to the conclusion of our goal, so we just have to establish the precondition:

$$\begin{aligned} &\mathrm{s} \approx \mathrm{s}' \wedge \rho(\mathrm{s}, \mathrm{s}_0) \wedge \neg\, \mathrm{final}(\mathrm{s}) \wedge \mathrm{EF}(\mathrm{s}_0, \lambda\, \mathrm{s}_1.\, \mathrm{s}_1 = \mathrm{s}_2) \wedge \mathrm{s}_2 \approx \mathrm{s}' \\ &\wedge (\mathrm{s}_2, \mathrm{s}') <_{m0} (\mathrm{s}, \mathrm{s}') \wedge \neg\, \mathrm{AF}'{+}(\mathrm{s}', \lambda\, \mathrm{s}'_0.\, \mathrm{EF}(\mathrm{s}, \lambda\, \mathrm{s}_0.\, \mathrm{s}_0 \approx \mathrm{s}'_0)) \\ \rightarrow\ &\neg\, \mathrm{AF}'{+}(\mathrm{s}', \lambda\, \mathrm{s}'_0.\, \mathrm{EF}(\mathrm{s}_2, \lambda\, \mathrm{s}_0.\, \mathrm{s}_0 \approx \mathrm{s}'_0)) \end{aligned}$$

By expanding both AF′+-formulas with the same state s'_0, such that $\rho'(\mathrm{s}',\mathrm{s}'_0)$ holds, using contraposition and finally applying (AF-imp) to cancel the AF-operators it remains to prove

$$\begin{aligned} &\mathrm{s} \approx \mathrm{s}' \wedge \rho(\mathrm{s}, \mathrm{s}_0) \wedge \neg\, \mathrm{final}(\mathrm{s}) \wedge \rho'(\mathrm{s}',\mathrm{s}'_0) \wedge \mathrm{EF}(\mathrm{s}_0, \lambda\, \mathrm{s}_1.\, \mathrm{s}_1 = \mathrm{s}_2) \\ &\wedge \mathrm{s}_2 \approx \mathrm{s}' \wedge (\mathrm{s}_2, \mathrm{s}') <_{m0} (\mathrm{s}, \mathrm{s}') \wedge \mathrm{EF}(\mathrm{s}_2, \lambda\, \mathrm{s}_0.\, \mathrm{s}_0 \approx \mathrm{s}'_1) \\ \rightarrow\ &\mathrm{EF}(\mathrm{s}, \lambda\, \mathrm{s}_0.\, \mathrm{s}_0 \approx \mathrm{s}'_1) \end{aligned}$$

This is done by unfolding EF in the conclusion, using s_0 to instantiate the resulting quantifier. Finally, (AF-step) steps from s_0 to s_2 and closes the goal.

A.3 Proof of Lemma 11

We start with the proof of the weaker lemma

$$\mathrm{s} \approx \mathrm{s}' \wedge \neg\,\mathrm{final}(\mathrm{s}) \wedge \neg\,\mathrm{final}'(\mathrm{s}') \rightarrow \mathrm{AF}'{+}(\mathrm{s}',\lambda\, \mathrm{s}'_0.\, \mathrm{EF}(\mathrm{s}, \lambda\, \mathrm{s}_0.\, \mathrm{s}_0 \approx \mathrm{s}'_0)) \tag{13}$$

which is need for partial preservation of traces. Its proof starts by applying Lemma 10, adding a precondition, which is by (12) equivalent to

$$\mathrm{EF}(\mathrm{s},\lambda\ \mathrm{s}_1.\ \mathrm{s}_1 = \mathrm{s}_0) \wedge \mathrm{s}_0 \approx \mathrm{s}' \wedge \neg\ \mathrm{EF}{+}(\mathrm{s}_0,\lambda\ \mathrm{s}_1.\ \mathrm{s}_1 \approx \mathrm{s}')$$

In this way, we introduce a state s_0, where no more m:0-diagram can be added. Applying (VC) to s_0 and s', we add

$$\mathrm{AF}'{+}(\mathrm{s}',\lambda\ \mathrm{s}'_0.\ \mathrm{EF}{+}(\mathrm{s}_0,\lambda\ \mathrm{s}_1.\ \mathrm{s}_1 \approx \mathrm{s}'_0) \vee \mathrm{s}_0 \approx \mathrm{s}'_0 \wedge (\mathrm{s}_0,\mathrm{s}'_0) <_{0\mathrm{n}}(\mathrm{s}_0,\mathrm{s}'))$$

i.e. that the following diagram is an m:n-diagram with $n > 0$ as a precondition. Eliminating both $\mathrm{AF}'{+}$ with the same successor state, we can use (AF-imp) to cancel the two resulting AF-Operators away. Expanding EF+ we get

$$\begin{aligned} &\quad \mathrm{s}_0 \approx \mathrm{s}' \wedge \mathrm{EF}(\mathrm{s},\ \lambda\ \mathrm{s}_1.\ \mathrm{s}_1 = \mathrm{s}_0)\\ &\quad \wedge (\rho(\mathrm{s}_0,\mathrm{s}_1) \wedge \mathrm{EF}(\mathrm{s}_1,\lambda\ \mathrm{s}_2.\ \mathrm{s}_2 \approx \mathrm{s}'_0) \vee \mathrm{s}_0 \approx \mathrm{s}'_0 \wedge (\mathrm{s}_0,\mathrm{s}'_0) <_{0\mathrm{n}}(\mathrm{s}_0,\mathrm{s}'))\\ &\rightarrow \mathrm{EF}(\mathrm{s},\ \lambda\ \mathrm{s}_0.\ \mathrm{s}_0 \approx \mathrm{s}'_0)\end{aligned}$$

The proof can now be completed by using (EF-step) to step from s to s_0 and unfolding the resulting $\mathrm{EF}(\mathrm{s}_0,\ldots)$-formula.

To prove Lemma 11, we apply (13), eliminate both $\mathrm{AF}'{+}$ using the same successor state, and use (AF-step) to step from this successor state to s'_0. This leaves the subgoal

$$\mathrm{EF}(\mathrm{s},\ \lambda\ \mathrm{s}_0.\ \mathrm{s}_0 \approx \mathrm{s}'_0) \rightarrow \mathrm{AF}'(\mathrm{s}'_0,\lambda\ \mathrm{s}'_1.\ \mathrm{EF}{+}(\mathrm{s},\ \lambda\ \mathrm{s}_0.\ \mathrm{s}_0 \approx \mathrm{s}'_1))$$

Unfolding EF gives the trivial case, where at least one step is taken to reach a state s_0 with $\mathrm{s}_0 \approx \mathrm{s}'_0$ (intuitively this is the case where the m:n diagram under consideration already has $m > 0$) and the case where we have $\mathrm{s} \approx \mathrm{s}'_0$ immediately. In this case we apply Lemma 9 with s and s'_0 to add the maximal number of 0:n-diagrams. This adds the precondition

$$\mathrm{AF}'(\mathrm{s}'_0,\ \lambda\ \mathrm{s}'_1.\ \mathrm{EF}{+}(\mathrm{s},\ \lambda\ \mathrm{s}_0.\ \mathrm{s}_0 \approx \mathrm{s}'_1) \vee \mathrm{final}'(\mathrm{s}'_1) \wedge \mathrm{s} \approx \mathrm{s}'_1)$$

to our previous goal. Using (AF-imp) to cancel the two AF-operators, we step to the state s'_1 at the end of the 0:n-diagrams. We get two cases: The first, where $\mathrm{EF}{+}(\ldots)$ holds is trivial already, the other is

$$\mathrm{final}'(\mathrm{s}'_1) \wedge \mathrm{s} \approx \mathrm{s}'_1 \rightarrow \mathrm{AF}'(\mathrm{s}'_1,\lambda\ \mathrm{s}'_2.\ \mathrm{EF}{+}(\mathrm{s},\ \lambda\ \mathrm{s}_0.\ \mathrm{s}_0 \approx \mathrm{s}'_2))$$

which is an instance of (VC).

A.4 Formalizing the Addition of a Commuting Diagram

Lemma 11 shows, that given two non-final similar states $\mathrm{s} \approx \mathrm{s}'$, and a trace σ' starting with s', we can find a trace σ starting with s and two states s_0, s'_0 on the two traces, which are both reached with a positive numbers of steps and which are similar again. With other words, we can add an m:n-diagram with m,n > 0. Now intuitively, given a trace σ' and a state s $\approx \sigma'(0)$, adding diagrams can be repeated infinitely to give the trace σ and the sequences of natural numbers (i_0, i_1, ...) and (j_0, j_1, ...) as required by Theorem 7. Unfortunately, the concept of "repeating a construction infinitely" is somewhat hard to formalize. In the following two sections we will give the idea of the formalization and the main lemmas needed. Once the lemmas "fit together", their proofs are simple, so we will skip them (although they are often rather lengthy).

The idea of the formalization is to define a function Σ: T $\rightarrow$ T over a suitable set T, which encodes the "construction". Finite repetition then means iterated application of the function. Infinite repetition will require a diagonalization argument. We will consider repetition in the following section, when we prove Theorem 7. In this section we will only prove the existence of a "construction", which adds another m:n-diagram with m,n > 0. The existence proof defines a lemma that is an instance of the precondition of the axiom of choice (ϕ is a formula with free variables x,y $\in$ T):

$$(\forall\ x.\ \exists\ y.\ \phi(x,y)) \rightarrow (\exists\ \Sigma.\ \forall\ x.\ \phi(x,\Sigma(x)))$$

Applying the axiom of choice on the lemma allows to deduce the existence of a choice function Σ, which encodes adding a diagram. To define the lemma, we define the type T to be the triples consisting of a function σ: Nat $\rightarrow$ S (intended to be a trace of M) and two natural numbers i and j. The idea is: if $\sigma(i) \approx \sigma'(j)$ holds, then applying Σ will add another diagram, i.e. it will construct σ_0, i_0, j_0, such that $i_0 > i$, $j_0 > j$, and $\sigma_0(i_0) \approx \sigma'(j_0)$. The construction also must preserve all commuting diagrams earlier on the trace, i.e. it is necessary to guarantee $\sigma(m) = \sigma_0(m)$ for all positions $m \leq i$. Formally we define the lemma:

$$\begin{aligned}\forall\ \sigma,i,j.\ \exists\ \sigma_0,i_0,j_0.\quad & \mathrm{trace}'(\sigma') \wedge \mathrm{trace}(\sigma) \wedge \sigma(i) \approx \sigma'(j) \\ \rightarrow \quad & \mathrm{trace}(\sigma_0) \wedge i < i_0 \wedge j < j_0 \wedge \sigma_0(i_0) \approx \sigma'(j_0) \\ & \wedge \forall\ m.\ m \leq i \rightarrow \sigma(m) = \sigma_0(m)\end{aligned} \tag{14}$$

Its proof has four cases: If both $\sigma(i)$ and $\sigma'(j)$ are final, then we can choose $(\sigma_0,i_0,j_0) := (\sigma,i+1,j+1)$, since final states are repeated. If only one of $\sigma'(j)$ and $\sigma(i)$ is final we can apply (VC), which will add a m:0 or 0:n-diagram respectively. (i_0, j_0) are set to be (i + m,j +1) and (i +1,j + n) in these two cases. σ_0 can be set to be σ for a 0:n-diagram. Otherwise a new trace must be constructed combining the first i steps of σ and the trace σ_1 starting with the m steps of the m:0 diagram. σ_1 is the result of expanding EF with its definition (1). Finally, we use Lemma 11 for the case of two non-final states to add an m:n-diagram with m,n > 0. σ_0 is defined similarly to the case of a m:0-diagram. (i_0, j_0) are set to be (i + m,j + n). Like in the 0:n-case, n is determined by expanding AF$'$ with (1) and instantiating the quantifier with σ'.

A.5 Proof of the Main Theorem

As mentioned above, the idea for the proof of the main theorem is to "infinitely repeat adding diagrams". The previous step has defined an operator Σ, which adds one diagram. Given this operator, we can easily define *finite* repetition as repeated application of the operator Σ: we start with some arbitrary trace σ of M, that satisfies $\sigma(0) \approx \sigma'(0)$. The existence of such a trace is guaranteed by the first condition[2] of Theorem 7. Adding k diagram now means applying Σ k times, so we set

$$(\sigma^{(\mathrm{k})},\mathrm{i_k},\mathrm{j_k}) := \Sigma^{\mathrm{k}}(\sigma,0,0) \tag{15}$$

Now the k^{th} trace $\sigma^{(\mathrm{k})}$ has been constructed to contain k commuting diagrams: $\sigma^{(\mathrm{k})}(\mathrm{i_m}) \approx \sigma'(\mathrm{j_m})$ for each $\mathrm{m} \leq \mathrm{k}$. All diagrams have positive size, and for $\mathrm{m} < \mathrm{k}$ traces $\sigma^{(\mathrm{m})}$ and $\sigma^{(\mathrm{k})}$ agree at all positions $\leq \mathrm{i_m}$. An inductive proof for these facts (over k) requires some generalization, which leads to the following lemma

$$\begin{array}{l} \quad \mathrm{trace}(\sigma) \wedge \mathrm{trace}(\sigma') \wedge \sigma(0) \approx \sigma'(0) \\ \rightarrow \forall\ \mathrm{m} \leq \mathrm{k}. \quad \mathrm{trace}(\sigma^{(\mathrm{m})}) \wedge \sigma^{(\mathrm{m})}(\mathrm{i_m}) \approx \sigma'(\mathrm{j_m}) \\ \qquad\qquad \wedge\ \mathrm{i_m} \geq \mathrm{m} \wedge \mathrm{i_m} \geq \mathrm{m} \wedge (\mathrm{m} < \mathrm{k} \rightarrow \mathrm{i_m} < \mathrm{i_k} \wedge \mathrm{j_m} < \mathrm{j_k}) \\ \qquad\qquad \wedge\ (\forall\ \mathrm{m_0} \leq \mathrm{i_m}.\ \sigma^{(\mathrm{m})}(\mathrm{m_0}) = \sigma^{(\mathrm{k})}(\mathrm{m_0})) \end{array} \tag{16}$$

The lemma assumes Σ to be defined as described above and uses (15) to abbreviate. Its proof is easy (although lengthy again), the main problem is to find a generalization that is both inductively provable and sufficient to prove the main theorem, which we can now state formally:

$$\begin{array}{l} \quad \sigma'(0) \in \mathrm{I}' \wedge \mathrm{trace}(\sigma') \\ \rightarrow \exists\ \sigma,\ (\mathrm{i_0},\ \mathrm{i_1},\ \ldots),\ (\mathrm{j_0},\ \mathrm{j_1},\ \ldots). \\ \qquad \sigma(0) \sim_I \sigma'(0) \wedge \mathrm{trace}(\sigma) \wedge (\forall\ \mathrm{k}.\ \sigma(\mathrm{i_k}) \sim \sigma'(\mathrm{j_k})) \\ \qquad \wedge\ (\forall\ \mathrm{m} < \mathrm{n}.\ \mathrm{i_m} < \mathrm{i_n} \wedge \mathrm{j_m} < \mathrm{j_n}) \\ \qquad \wedge\ ((\exists\ \mathrm{m}.\ \mathrm{final}(\sigma(\mathrm{m}))) \leftrightarrow (\exists\ \mathrm{n}.\ \mathrm{final}(\sigma'(\mathrm{n})))) \end{array} \tag{17}$$

The central idea needed to prove the theorem is, that the sequence of $\sigma^{(\mathrm{k})}(\mathrm{k})$ for each k (the "diagonal trace") is a trace of M, which contains an infinite number of commuting diagrams. This is indeed the case, since it agrees for every k with $\sigma^{(\mathrm{k})}$ until at least the positions $\mathrm{i_k}$, where the k^{th} diagram ends. The third verification condition of Theorem 7 is used to deduce $\sigma(\mathrm{i_k}) \sim \sigma'(\mathrm{j_k})$ from $\sigma(\mathrm{i_k}) \approx \sigma'(\mathrm{j_k})$ for each k. Otherwise the proof is completed by mapping the facts stated in (16) to the ones stated in the theorem by instantiating quantifiers suitably. The only exception is the proof of the final line of the main theorem, which ensures that a finite trace σ' is the refinement of a finite trace σ (and vice versa), which requires Lemmas 9 and 10 to find a corresponding final state.

[2] For the weaker condition (11) of Sect. 6.1 start with $(\sigma,\mathrm{i_0},\mathrm{j_0})$ such that $\sigma(\mathrm{i_0}) \approx \sigma'(\mathrm{j_0})$