Acta Informatica (2022) 59:521–556
https://doi.org/10.1007/s00236-021-00408-8

ORIGINAL ARTICLE

Interface Automata for Shared Memory

Ayleen Schinko[1] · Walter Vogler[1] · Johannes Gareis[2] · N. Tri Nguyen[2] · Gerald Lüttgen[2]

Received: 28 November 2019 / Accepted: 20 July 2021 / Published online: 30 August 2021

Abstract
Interface theories based on *Interface Automata* (IA) are formalisms for the component-based specification of concurrent systems. Extensions of their basic synchronization mechanism permit the modelling of data, but are studied in more complex settings involving modal transition systems or do not abstract from internal computation. In this article, we show how de Alfaro and Henzinger's original IA theory can be conservatively extended by shared memory data, without sacrificing simplicity or imposing restrictions. Our extension *IA for shared Memory* (IAM) decorates transitions with pre- and post-conditions over algebraic expressions on shared variables, which are taken into account by IA's notion of component compatibility. Simplicity is preserved as IAM can be embedded into IA and, thus, accurately lifts IA's compatibility concept to shared memory. We also provide a ground semantics for IAM that demonstrates that our abstract handling of data within IA's open systems view is faithful to the standard treatment of data in closed systems.

1 Introduction

Behavioural types [4,17] play an increasingly important role when developing and verifying software systems. For object-oriented software, behavioural types are specified as *contracts* [25], annotating methods and classes with pre- and post-conditions and invariants, respectively. For distributed software, *session types* are employed [10,19] to specify permitted and prohibited interactions. For concurrent software, component compatibility has been studied by *interface theories*, such as de Alfaro and Henzinger's *Interface Automata* (IA) [12,13]. Although IA is a well-studied theory, it does not *directly* reflect modern IT architectures that rely on networked, distributed *clusters* of multi-core/-processor computers. In these architec-

Research support provided by the DFG (German Research Foundation) under Grant Nos. LU 1748/3-2 and VO 615/12-2.

✉ Gerald Lüttgen
gerald.luettgen@uni-bamberg.de

Walter Vogler
walter.vogler@informatik.uni-augsburg.de

1 Institut für Informatik, University of Augsburg, Augsburg, Germany

2 Software Technologies Research Group, University of Bamberg, Bamberg, Germany

tures, communication between clusters is via message passing, and communication within a cluster is typically via shared variables. Holik et al. have studied the IA paradigm for message passing in [18], but the treatment of shared variables in the context of IA has so far been unsatisfactory.

This article is addressed to concurrency theoreticians and studies an IA-based setting where data are communicated via shared memory while activities, or *operations*, are triggered by synchronizing actions. In this introduction, we first summarize IA and related works on extensions of IA with data, both of which is elaborated in Sects. 2 and 3, respectively. This allows us to spell out our contributions and advances over the state-of-the-art in interface theories.

Background Interface Automata (IA) [12,13] model system components as labelled transition systems that distinguish a component's input and output actions. Parallel component composition assumes that a component may wait on inputs but never on outputs, implying that a component's output must be consumed immediately, or a communication error occurs. In case no system environment may restrict the components' behaviour so that all errors are avoided, the components are deemed to be *incompatible*.

To support refinement during software development, IA is equipped with a branching-time behavioural preorder called *alternating simulation*; it allows one to substitute an abstract component by a concrete one, provided the concrete component offers no fewer inputs and no more outputs than the abstract one. As this implies that outputs cannot be enforced in IA, researchers have frequently based IA-inspired interface theories on the more expressive *Modal Transition Systems* (MTS) and *modal refinement* [3,5,15,22,29]. As an aside, a linear-time semantics respecting communication errors as in IA was proposed by Dill [14] and more recently adapted for variations of IA in [6,8].

To enable the modelling of richer classes of concurrent systems and software, IA has been extended in various ways to capture data in addition to control [1,2,9,11,18,27]. Except for [18], these works focus on shared memory. The work of Mouelhi et al. [9,27] is closest to ours and adopts a paradigm well-known from structured programming. It considers an output of one IA to trigger an operation in another IA. The guaranteed data state resulting upon executing the operation on the input side is expected to meet the trigger's assumption; otherwise, an error occurs. While Mouelhi et al. stick to the IA-setting and express assumptions and guarantees on data states by decorating actions with pre- and post-conditions, respectively, none of the essential results that we treat in Sects. 3–7 are shown.

For an interface theory built upon the more expressive MTS setting, Bauer et al. [1,2] also introduce data via pre- and post-conditions with the idea of operation calls. But counter to the optimistic view on component compatibility adopted by IA, they restrict themselves to a *pessimistic* view, whereby components are already deemed to be incompatible if there *exists* a system environment that causes a communication error. De Alfaro et al.'s *Sociable Interfaces* [11] employs scoping and non-interference conditions to ensure that interfaces can participate in sophisticated n-to-n communication schemes on global variables. The semantics of sociable interfaces is not directly based on a behavioural preorder but uses a related phrasing in terms of game semantics. This semantics, however, does not abstract from internal computation, which is also a limitation for [1,2].

Contributions This article demonstrates that IA permits a *simple* and *faithful* extension where data are communicated via shared memory while activities are triggered by synchronizing actions. To do so, we develop *Interface Automata for Shared Memory* (IAM) which, similarly to [1,2,9,27], specify operation calls via pre- and post-conditions attached to transitions (see Sect. 3).

Naturally, pre-conditions act as transition guards. As in [9,27], the post-condition of an output-transition and that of a matching input-transition describe an assumption and guarantee, respectively, on the data state obtained after the synchronization. Thus, an output-transition may be understood as invoking an operation associated with the transition's action, whereas the matching input-transition describes how this operation manipulates the system's data state. One may think of the input-transition to actually perform the operation. This approach contrasts the one in [11], where the inputs contain only assumptions about the change of data values and the outputs determine which values are produced. We adopt the point-to-point, handshake communication model of IA that is inspired by the process algebra CCS [26]. A communication error occurs if either, as in IA, a matching input is not provided for an output, or if the new data state set by the input after synchronization does not obey the output's assumption.

We conservatively extend IA's concepts of parallel composition and alternating refinement to IAM, while preserving compositionality and abstracting from internal computation (see Sects. 3, 4). Then, we formally prove that IAM can be embedded into IA, which implies that, using IAM, one can reason finitely about component compatibility in IA, even when infinite data domains are involved (see Sect. 6); this is an important aspect not considered in related work. We also connect our abstract treatment of data via pre- and post-conditions, where data states are implicit, to the concrete treatment known from ground semantics for (data-)closed systems (see Sect. 7). Such ground semantics naturally encode data values into control states, leading to infinite presentations in the presence of infinite data domains. For establishing the connection, we define a reduction on IAM such that—under some restrictions—ground semantics *and* finiteness are preserved. The key result is that this ground semantics is bisimilar to the structure of the reduced IAM. Hence, we arrive at a finite presentation of the usually infinite ground semantics, which we consider as our most original and interesting result. The result shows that a cluster, as introduced above, externally behaves as an IA, i.e. as a message passing system, so that our ground semantics provides the user with an often finite description of the cluster's behaviour.

This article expands the published extended abstract [16] in multiple ways. First, a detailed introduction to IA and discussion of closely related work on extending IA with data is given here (see Sect. 2), in order to make the article accessible to an as broad audience of concurrency theoreticians as possible. Further, we extend the IAM setting with the possibility to refer to the previous data state in post-conditions by distinguishing between 'primed' and 'unprimed' variables. Thus, it is now possible to specify that a value is preserved or changed by some operation. The setting in [16] is equivalent to the current one if we refrain from unprimed variables in post-conditions. Additionally, this article technically justifies the pruning employed in parallel composition, includes all proofs of our results, and applies our theory to an illustrative example. The latter also testifies to the practical importance of primed variables (see Sect. 5.1). To testify that our theoretical research may eventually lead to tool-supported interface checkers employed in software engineering practice, we have built a first, prototypic, open-source IAM toolset that consists of a textual modelling language, implements interface composition, allows for simulating interfaces and debugging them for communication errors, automates refinement checking for simple data theories, and offers the interactive exploration of counterexamples in case a refinement check fails (see Sect. 5.2). Last but not least, we have improved the article's presentation in particular regarding, but not limited to, our ground semantics (see Sect. 7).

2 The IA setting

We start off with a brief technical introduction to the classical setting of *Interface Automata* (IA) by de Alfaro and Henzinger [13]. This makes it easier to understand our extension with data, and the concepts and results for IA are technically needed in Sects. 6 and 7 for our shared-memory extension IAM. An interface automaton specifies a system component as a labelled transition system over some input and output alphabets I and O, respectively.

Definition 1 (*Interface Automata*) An *Interface Automaton* (IA) is a tuple $(P, I, O, \rightarrow, p_0)$, where

- P is the set of *states* with $p_0 \in P$ being the *initial state*.
- $A =_{\text{df}} I \cup O$ is the *alphabet* consisting of disjoint sets I and O of *input* and *output actions*, respectively, not containing the distinguished *internal action* τ; (I, O) is the *signature* of the IA.
- $\rightarrow \subseteq P \times (A \cup \{\tau\}) \times P$ is the *transition relation*, and $p \xrightarrow{\alpha} q$ is written for $(p, \alpha, q) \in \rightarrow$. The IA is required to be *input-deterministic*, i.e. $p' = p''$ whenever $p \xrightarrow{a?} p'$ and $p \xrightarrow{a?} p''$ for some $p \in P$ and $a? \in I$.

We use $a, a?, a!, \alpha, \alpha?$ and $\alpha!$ as representatives of the sets $A, I, O, A \cup \{\tau\}, I \cup \{\tau\}$ and $O \cup \{\tau\}$ and simply write P for an IA $(P, I, O, \rightarrow, p_0)$, and similarly for IAM below. According to the above definition, a transition is labelled either with an input action $a?$, an output action $a!$ or the internal action τ. Here, ? and ! are just comments showing whether an action is an input or an output. The actual name of the action is a; thus, two components of a parallel composition can share an action a, which then necessarily is an input of one and an output of the other component (see below). The two IAs synchronize on such actions a, resulting in internal τ-transitions; the IAs interleave on all other actions. In general, an $a?$-labelled transition shows that the component is willing to synchronize on a. In contrast, an action $a!$ (like τ) is *locally controlled* (or *local* for short) and can always be performed disregarding the environment. Thus, $a!$ initiates a synchronization, and this requires an environment with input $a?$ to be ready to receive a with a corresponding transition. As an aside note that, in [13], there can be several internal actions and the refinement does not allow one to rename these. We avoid this issue by having a single internal action τ only.

Definition 2 (*Parallel Product*) IAs $(P, I_P, O_P, \rightarrow_P, p_0)$ and $(Q, I_Q, O_Q, \rightarrow_Q, q_0)$ are *composable* if $O_P \cap O_Q = \emptyset = I_P \cap I_Q$. The *product* $P \overline{\otimes} Q$ of such composable IAs is defined as $(P \times Q, I, O, \rightarrow, (p_0, q_0))$, where $I =_{\text{df}} (I_P \cup I_Q) \backslash (A_P \cap A_Q)$, $O =_{\text{df}} (O_P \cup O_Q) \backslash (A_P \cap A_Q)$, and the transition relation $\rightarrow$ is the least relation satisfying the following rules:

- $\langle p, q \rangle \xrightarrow{\alpha} \langle p', q \rangle$ if $p \xrightarrow{\alpha}_P p'$ and $\alpha \notin A_Q$;
- $\langle p, q \rangle \xrightarrow{\alpha} \langle p, q' \rangle$ if $q \xrightarrow{\alpha}_Q q'$ and $\alpha \notin A_P$;
- $\langle p, q \rangle \xrightarrow{\tau} \langle p', q' \rangle$ if $p \xrightarrow{a?}_P p'$, $q \xrightarrow{a!}_Q q'$ for some a;
- $\langle p, q \rangle \xrightarrow{\tau} \langle p', q' \rangle$ if $p \xrightarrow{a!}_P p'$, $q \xrightarrow{a?}_Q q'$ for some a.

Note that we often drop the index from a transition relation whenever it is clear from the context.

If IA P outputs an action shared with IA Q at a state p but Q cannot receive it in its current state q, a communication error occurs in state $\langle p, q \rangle$. The parallel composition prunes all illegal states, i.e. all error states $\langle p, q\rangle$ as well as those states from which an error state can be reached autonomously via local transitions; these are the states where the environment can no longer prevent the composition of P and Q from entering an error state. This consideration of errors makes the IA interface theory suitable for reasoning about *component compatibility* and provides a means of *error-awareness*.

Interface Automata apply an open systems view with optimistic compatibility, assuming that a composed system runs in a so-called helpful environment. Such an environment obeys the operation guideline as given by the parallel composition $P \parallel Q$ (see below) and avoids communication mismatches by controlling the composed component's input actions. Other theories such as [1,3] take a pessimistic view, where no assumptions about the environment are made: any occurrence of a reachable error state leads to an incompatibility of the components.

Definition 3 (*Parallel Composition*) Given a parallel product $P \overline{\otimes} Q$ of IAs P and Q, a state $\langle p, q\rangle$ is an *error state* if there is some $a \in A_P \cap A_Q$ s.t.

- $p \xrightarrow{a!} p'$ with $a \in I_Q$, but there is no transition $q \xrightarrow{a?} q'$;
- $q \xrightarrow{a!} q'$ with $a \in I_P$, but there is no transition $p \xrightarrow{a?} p'$.

The set $E \subseteq P \times Q$ of *illegal states* is the least set containing all error states and those states $\langle p, q\rangle$ for which there is a transition $\langle p, q\rangle \xrightarrow{\alpha!} \langle p', q'\rangle$ with $\langle p', q'\rangle \in E$. The *parallel composition* $P \parallel Q$ is then computed by *pruning* the illegal states, i.e. by removing all states in E and their incoming and outgoing transitions. If $\langle p, q\rangle \in P \parallel Q$, then states p and q are *compatible*, written $p \parallel q$; one also says that $p \parallel q$ is defined. Furthermore, P and Q are compatible if their initial states are compatible; otherwise, $P \parallel Q$ is undefined.

Proposition 4 (Associativity & Commutativity) *If P, Q and R are pairwise composable IAs, then $(P \parallel Q) \parallel R$ and $P \parallel (R \parallel Q)$ are either both undefined or $(P \parallel Q) \parallel R$ and $P \parallel (R \parallel Q)$ are isomorphic. Analogously, $\parallel$ is commutative.*

IA supports the compositional refinement of systems via a notion of *alternating simulation*: P refines Q if the input-transitions of Q can be simulated by P and, vice versa, the output-transitions of P can be simulated by Q. By simply following this pattern, new communication errors cannot be introduced during refinement. The pattern prevents the removal of specified inputs needed to synchronize with outputs of an environment as well as the addition of unspecified outputs, which would create new needs of synchronization. A weak transition $\overset{\varepsilon}{\Rightarrow}$ stands for a finite and possibly empty sequence of τ-transitions.

Definition 5 (*Alternating Simulation*) For IAs P, Q with the same signature, a relation $\mathcal{R} \subseteq P \times Q$ is an *alternating simulation* if the following conditions hold for all $p\mathcal{R}q$ and actions a:

(i) $q \xrightarrow{a?} q'$ implies $\exists p'.\ p \xrightarrow{a?} p'$ and $p'\mathcal{R}q'$
(ii) $p \xrightarrow{a!} p'$ implies $\exists q', q''.\ q'' \xrightarrow{a!} q'$ with $q \overset{\varepsilon}{\Rightarrow} q''$ and $p'\mathcal{R}q'$.
(iii) $p \xrightarrow{\tau} p'$ implies $\exists q'.\ q \overset{\varepsilon}{\Rightarrow} q'$ and $p'\mathcal{R}q'$.

We write $P \sqsubseteq_{IA} Q$ to say that P *IA-refines* Q, if there exists an alternating simulation $\mathcal{R}$ s.t. $p_0\mathcal{R}q_0$. Further, P is IA-equivalent to Q if P and Q IA-refine each other.

Intuitively, a state p refines q if it matches all inputs of q and, conversely, if q matches all outputs of p. Note that p may have additional inputs in which q cannot engage and which may be followed by arbitrary behaviour afterwards. It is not hard to show that the IA-refinement $\sqsubseteq_{IA}$ is a preorder and, as motivated above, a pre-congruence for parallel composition. It is also easy to see that, for the latter result, an input of q must immediately be matched by p, i.e. without preceding τs; neither trailing τs are allowed.

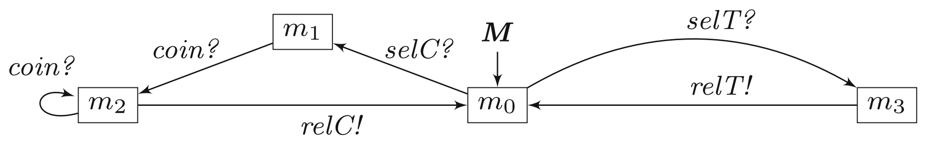


Fig. 1 Vending machine M with signature ($\{selC, selT, coin\}$, $\{relC, relT\}$)

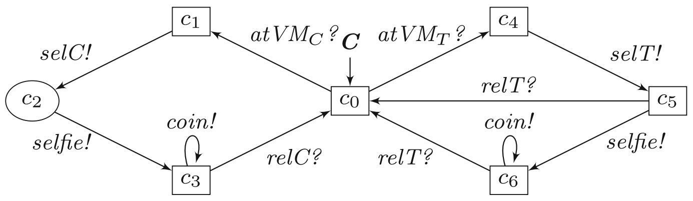


Fig. 2 Customer C with signature ($\{\text{atVM}_C, \text{atVM}_T, relC, relT\}$, $\{selC, selT, selfie, coin\}$)

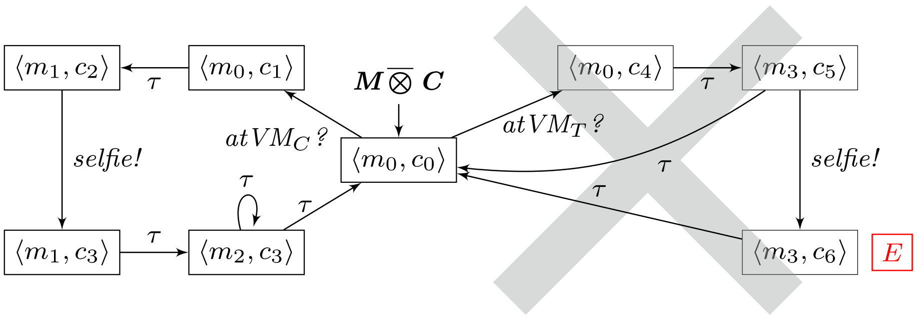


Fig. 3 Parallel product $M \overline{\otimes} C$ (full graph) and parallel composition $M \parallel C$ (without the crossed-out part)

Theorem 6 (Compositionality of IA) *Let P, Q, R be IAs and $P \sqsubseteq_{IA} Q$. Further assume that Q and R are composable. We have: (i) P and R are composable; (ii) $P \parallel R$ is defined if $Q \parallel R$ is, and then $P \parallel R \sqsubseteq_{IA} Q \parallel R$.*

We illustrate the concepts of IA with a simple vending machine example that consists of two components, the machine—offering coffee and tea—and the customer (see Figs. 1, 2).

The machine M waits for the customer C to select coffee or tea (inputs *selC?* and *selT?*). In the case of coffee, the machine accepts a positive number of coins (transitions labelled *coin?*), while tea is free of charge. Eventually, coffee or resp. tea is released (outputs *relC!* and *relT!*). If the customer approaches the vicinity of the machine and is thirsty for coffee or tea (inputs atVM_C? and atVM_T?), she or he presses the appropriate selection button (outputs *selC!* and *selT!*) and makes a selfie of her or him at the machine afterwards (output *selfie!*). In case of tea, a British customer would also accept the drink without taking the selfie. She or he then inserts some coins (output *coin!*) and waits for the drink to be dispensed (inputs *relC?* and *relT?*).

For the parallel composition of M and C, one determines the product shown in Fig. 3. Here, $\langle m_3, c_6\rangle$ is an error state, because m_3 does not accept the coin offered in c_6. Therefore, also states $\langle m_3, c_5\rangle$ and $\langle m_0, c_4\rangle$ are illegal, because they can locally reach $\langle m_3, c_6\rangle$. Thus, $M \parallel C$ is $M \overline{\otimes} C$ without the tea vending cycle in the right half of the figure.

An IA-refinement C' of C that accommodates the taste of Italian customers is shown in Fig. 4. Customer C' has an additional input *relC?* at state c_2, because he or she would also

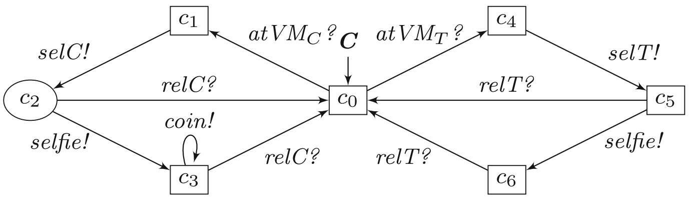


Fig. 4 Refined customer C'

accept coffee earlier. Moreover, C' has learned from M, seen as an 'operating guideline,' not to pay for tea. The product $M \overline{\otimes} C'$ is the same as $M \overline{\otimes} C$ except that $\langle m_3, c_6\rangle$ is no longer an error state. Hence, $M \parallel C' = M \overline{\otimes} C'$. Note that the additional input at c_2 has no effect on the composition.

3 The IAM setting

To capture shared memory in IA, our extension IAM additionally labels transitions by *pre-* and *post-conditions*, similar to [1,2,9,27]. These allow us to abstractly specify and reason about manipulations of *global variables* without modelling data states explicitly, i.e. as functions from variables to values. Formally, the shared memory is represented by a set V of variables, ranged over by x, y, etc., and all IAMs, as defined below, have the same set V of variables. Pre-conditions are predicates over V, i.e. first-order formulas with interpreted function and predicate symbols. Besides the propositional operators $\neg$, $\vee$, $\wedge$ and $\rightarrow$, we employ the usual arithmetic operators and predicate symbols such as $<$, $=$ and $\geq$. The universe of our predicates is denoted by $Pred(V)$, with representatives φ, $\overline{\varphi}$, φ_1, etc. In order to represent also post-values of variables, we introduce the set V' of fresh primed variables x' for each $x \in V$. While x represents the value of x before executing the respective transition, the value after is referenced by x'. We write φ' for φ where each x is replaced by x'. Post-conditions are ranged over by ψ, ψ_1, etc. and taken from the set $Pred(V, V')$ of predicates over $V \cup V'$.

A *data state* over V, often represented by σ or $\overline{\sigma}$, is a valuation $\sigma \colon V \rightarrow \mathbb{D}$ over a fixed, possibly infinite domain $\mathbb{D}$ of values. The set of all these data states is denoted by $[\![V]\!]$. To take a transition, now also its pre-condition φ must be satisfied by the current data state σ, and we write $\sigma \models \varphi$ in this case; further, we define $[\![\varphi]\!] =_{\text{df}} \{\sigma \in [\![V]\!] \mid \sigma \models \varphi\}$, and similarly for predicates over V'. The post-condition ψ must be satisfied by σ together with the new data state $\overline{\sigma}$ applied to the primed variables, and we write $\sigma, \overline{\sigma}' \models \psi$. Here, $\overline{\sigma} \colon V \rightarrow \mathbb{D}$ is a valuation like σ: because the post-condition refers to the next state with primed variables, we introduce $\overline{\sigma}' \colon V' \rightarrow \mathbb{D}$ defined by $\overline{\sigma}'(x') = \overline{\sigma}(x)$ for all $x \in V$.

Intuitively, a pre-condition acts as a guard for a transition. An output-transition labelled $(\varphi, a!, \psi)$ specifies that it is only executable when the system's data state satisfies pre-condition φ and expects that the system environment, upon synchronizing on a, leaves the system in a data state such that post-condition ψ is satisfied. Similarly, an input-transition labelled $(\varphi, a?, \psi)$ is executable in data states satisfying φ and guarantees that it implicitly manipulates variables only in ways such that the post-condition ψ is satisfied. This corresponds to an operation call: the caller performs an output-transition; this requires the callee to execute a respective input-transition in each data state satisfying φ, or otherwise a communication mismatch arises; the callee modifies the memory according to the post-condition of the input-transition and requires the caller to accept this. A violation of the caller's assumption

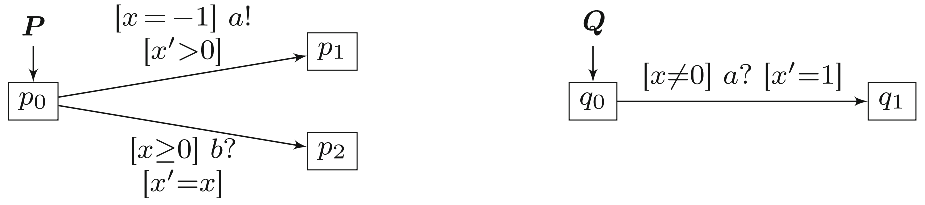


Fig. 5 Examples of communicating IAMs P and Q

gives rise to a (data-)error in the parallel composition (see Sect. 4). These mutual requirements are very different from the one-way requirement in an IA; thus, it is quite surprising that we can nevertheless embed IAM into IA below (see Sect. 6).

Interface Automata for Shared Memory We can now formally define the notion of IAM:

Definition 7 (*Interface Automata for Shared Memory*) An *Interface Automaton for Shared Memory* (IAM) is a tuple $(P, I, O, \rightarrow, p_0)$, where P, I, O, p_0 and the related notations are as for IA (cf. Definition 1) and $\rightarrow \subseteq P \times \mathit{Pred}(V) \times (A \cup \{\tau\}) \times \mathit{Pred}(V, V') \times P$ is the *transition relation*. The states in P are also called *control states*.

For a transition $(p, \varphi, \alpha, \psi, p') \in \rightarrow$, written $p \multimap(\varphi, \alpha, \psi)\rightarrowtail p'$ for simplicity, its pre-condition φ must be satisfiable, i.e. $[\![\varphi]\!] \neq \emptyset$ and denoted by φ *sat*. There is a stricter requirement for the post-condition: if one assigns values to the unprimed variables in ψ according to some data state σ satisfying φ, the resulting predicate $\psi(\sigma)$ must be satisfiable.[1] This requirement can also be written as $\varphi \Rrightarrow \exists X'.\, \psi$, where $\exists X'$ existentially quantifies over all primed variables in ψ and symbol $\Rrightarrow$ denotes logical entailment.

Moreover, the transition relation $\rightarrow$ is required to be *data-deterministic* for input actions, i.e. for all $p \in P$ and different transitions $p \multimap(\varphi_1, a?, \psi_1)\rightarrowtail p'$ and $p \multimap(\varphi_2, a?, \psi_2)\rightarrowtail p''$ with $a? \in I$, the conjunction $\varphi_1 \wedge \varphi_2$ is unsatisfiable.

In figures, pre- and post-conditions are set in square brackets. Note that the above definition coincides with the one of IA [13] in case only the tautology true, written *tt*, is allowed as pre- and post-condition. In particular, the normally finer notion of data-determinism is then the same as *input-determinism*, which is needed to achieve compositionality for parallel composition (see Theorem 6). Data-determinism is needed in our approach for the respective result (see Corollary 18).

Figure 5 shows two simple IAMs P and Q, where P can call operation a (output action $a!$) in initial state p_0, provided that variable x has value -1 (pre-condition $x = -1$). IAM P assumes that operation a adjusts the value of x to a positive one (post-condition $x' > 0$), and transitions to state p_1. Alternatively, if x is non-negative (pre-condition $x \geq 0$), P can serve a call to operation b in p_0 (input action $b?$), guarantees that the value of x remains unchanged (post-condition $x' = x$), and transitions to state p_2. Similarly, IAM Q offers to execute operation a in state q_0 (input action $a?$), provided that x has a non-zero value (pre-condition $x \neq 0$), and transitions to state q_1 while guaranteeing that the value of x is set to 1 (post-condition $x' = 1$). The resulting synchronization on action a in the parallel composition of P and Q can only take place if $x = -1$ and then updates the value of x to 1. The post-condition of Q's transition always meets the assumption of P's transition due to $x' = 1 \Rrightarrow x' > 0$. Parallel composition for IAM is fully introduced and discussed in Sect. 4.

[1] For example, given σ with $\sigma(y) = 1$ and the predicate $x' + y = 3$, the result is $x' + 1 = 3$. In other words, $\psi(\sigma)$ results from ψ by replacing any unprimed variable x with the syntactical element representing $\sigma(x)$.

Alternating simulation We now adapt alternating simulation from IA to IAM. First observe that a single IAM transition may represent many transitions when explicitly considering the underlying data state. This implies that the behaviour of a transition in one IAM might need to be matched by multiple transitions in another IAM, i.e. by a *family*—an indexed collection—of transitions. For example, the behaviour of a transition labelled with pre-condition $x \geq 0$ may only be covered in the other IAM by two transitions with pre-conditions $x = 0$ and $x > 0$, respectively, where the former transition matches the behaviour for data states in which x has value 0 and the latter in which x has a strictly positive value. Employing such families makes our approach more flexible and sophisticated than the one in [9,27]. In the following, families of transitions are indexed over some implicit index set with representative i. Such a family could even be infinite, e.g. when encoding local data states with infinite domains; the meaning of $\varphi \Rightarrow \bigvee_i \varphi_i$ is clear also in such a case.

A weak transition $\overset{\varepsilon}{\Rightarrow}$ stands for a finite and possibly empty sequence of τ-transitions labelled with arbitrary satisfiable pre- and post-conditions. This is because an environment component can always change a data state satisfying a post-condition to a data state satisfying the next pre-condition. This also explains why pre- and post-conditions are simply ignored in Item (iii) of the following definition.

Definition 8 (*Alternating Simulation*) For IAMs P and Q with the same signature, $\mathcal{R} \subseteq P \times Q$ is an *alternating simulation* if the following conditions hold for all $p\mathcal{R}q$ and φ, a, ψ:

(i) $q \prec\!(\varphi, a?, \psi)\!\succ q'$ implies that there exists a family of $p \prec\!(\varphi_i, a?, \psi_i)\!\succ p'_i$ with $\varphi \Rightarrow \bigvee_i \varphi_i$ and, for all i and all σ, $\sigma \models \varphi \wedge \varphi_i$ implies $\psi_i(\sigma) \Rightarrow \psi(\sigma)$ and $p'_i\mathcal{R}q'$.
(ii) $p \prec\!(\varphi, a!, \psi)\!\succ p'$ implies that there exists a family of $q_i \prec\!(\varphi_i, a!, \psi_i)\!\succ q'_i$ with $\varphi \Rightarrow \bigvee_i \varphi_i$ and, for every σ with $\sigma \models \varphi$, there is at least one transition $q_i \prec\!(\varphi_i, a!, \psi_i)\!\succ q'_i$ of the family satisfying $\sigma \models \varphi_i$, $q \overset{\varepsilon}{\Rightarrow} q_i$, $\psi_i(\sigma) \Rightarrow \psi(\sigma)$ and $p'\mathcal{R}q'_i$.
(iii) $p \prec\!(\varphi, \tau, \psi)\!\succ p'$ implies that there exists a q' with $q \overset{\varepsilon}{\Rightarrow} q'$ and $p'\mathcal{R}q'$.

We write $P \sqsubseteq_{IAM} Q$ and say that P *IAM-refines* Q, if there exists an alternating simulation $\mathcal{R}$ such that $p_0\mathcal{R}q_0$. Further, P is IAM-equivalent to Q if P and Q IAM-refine each other. Note that "for all σ with $\sigma \models \varphi \wedge \varphi_i$, $\psi_i(\sigma) \Rightarrow \psi(\sigma)$" is equivalent to $\varphi \wedge \varphi_i \wedge \psi_i \Rightarrow \psi$.

As for alternating simulation in IA and motivated in Sect. 2, inputs must be matched immediately for IAM, i.e. leading and trailing τs are not allowed. Additionally, Cond. (i) shows that pre-conditions are weakened while post-conditions are strengthened. The weakening of a pre-condition keeps in line with IA in that refinement may introduce additional inputs. The strengthening in the post-conditions ensures that refinement does not loosen guarantees, i.e. strengthening prevents the refined system from generating more data states than specified. Note that, as Cond. (i) covers input actions, the pre-conditions φ_i are pairwise disjoint due to data-determinism.

Outputs are matched the other way around, as specified in Cond. (ii), which is analogous to Cond. (i) except that, as in $\sqsubseteq_{IA}$, leading τs are permitted when matching output-transitions. The matching runs show that the same output can occur in the specification, at least in an environment causing the appropriate changes of data states; this time, the post-condition of the specification is stricter because it is a requirement. Cond. (iii) has been explained above.

Despite the fact that $\sqsubseteq_{IA}$ does not consider families when matching transitions, our alternating simulation $\sqsubseteq_{IAM}$ for IAM coincides with $\sqsubseteq_{IA}$ when considering only IAMs where all pre- and post-conditions are equivalent to *tt*. For such IAMs, it does not matter with which family member a transition is matched, any one family member would do.

An example illustrating alternating refinement is depicted in Fig. 6, where IAMs P, Q, R are defined over shared variable sets $V = \{x\}$ and $V' = \{x'\}$ with domain $\mathbb{R}$ such

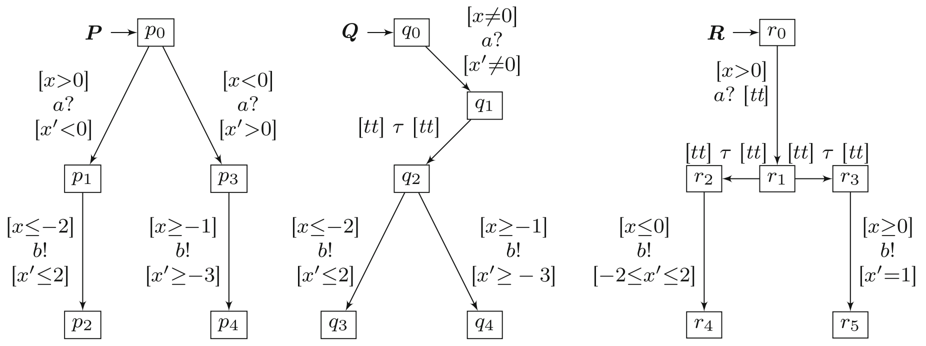


Fig. 6 Refinement $P \sqsubseteq_{IAM} Q \sqsubseteq_{IAM} R$

that $P \sqsubseteq_{IAM} Q \sqsubseteq_{IAM} R$. The input-transition $r_0 \relbar\!(x > 0, a?, tt)\!\!\rightarrowtail r_1$ of the most abstract IAM R is matched by transition $q_0 \relbar\!\big(x \neq 0, a?, x' \neq 0\big)\!\!\rightarrowtail q_1$ of Q, where the pre-condition is weakened and the post-condition is strengthened. In turn, this transition is matched in P by the transitions $p_0 \relbar\!\big(x > 0, a?, x' < 0\big)\!\!\rightarrowtail p_1$ and $p_0 \relbar\!\big(x < 0, a?, x' > 0\big)\!\!\rightarrowtail p_3$. The internal transition of q_1 is matched by r_1 staying put.

For output-transitions, leading τs are allowed when refining. Transition $q_2 \relbar\!\big(x \geq -1, b!, x' \geq -3\big)\!\!\rightarrowtail q_4$ of Q is matched in R by the family consisting of $r_2 \relbar\!\big(x \leq 0, b!, -2 \leq x' \leq 2\big)\!\!\rightarrowtail r_4$ and $r_3 \relbar\!\big(x \geq 0, b!, x' = 1\big)\!\!\rightarrowtail r_5$; observe that $r_1 \overset{\varepsilon}{\Rightarrow} r_2$ and $r_1 \overset{\varepsilon}{\Rightarrow} r_3$. Note that R has overlapping pre-conditions in r_1; unlike for inputs, data-determinism is not required for local actions in IAM.

Theorem 9 (Preorder) $\sqsubseteq_{IAM}$ *is a preorder.*

Proof Reflexivity immediately follows from the fact that the identity relation is an alternating simulation. For transitivity, we let $P \sqsubseteq_{IAM} Q$ and $Q \sqsubseteq_{IAM} R$ due to $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. We show that $\mathcal{R} =_{\text{df}} \{(p, r) \mid \exists q \in Q.\ p\mathcal{R}_1 q \text{ and } q\mathcal{R}_2 r\}$ is an alternating simulation, too, implying $P \sqsubseteq_{IAM} R$. Obviously, $p_0 \mathcal{R} r_0$. Let $p\mathcal{R}r$ due to q with $p\mathcal{R}_1 q$ and $q\mathcal{R}_2 r$, and distinguish Cases (i)–(iii) according to Definition 8. Because Cases (i) and (iii) are easier than Case (ii), we focus on the latter only. A transition $p \relbar\!(\varphi, a!, \psi)\!\!\rightarrowtail p'$ implies the existence of a family $q_i \relbar\!(\varphi_i, a!, \psi_i)\!\!\rightarrowtail q_i'$ with $\varphi \Rightarrow \bigvee_i \varphi_i$ and, for every σ with $\sigma \models \varphi$, there is at least one transition $q_i \relbar\!(\varphi_i, a!, \psi_i)\!\!\rightarrowtail q_i'$ of the family such that $\sigma \models \varphi_i$, $q \overset{\varepsilon}{\Rightarrow} q_i$, $\psi_i(\sigma) \Rightarrow \psi(\sigma)$ and $p'\mathcal{R}_1 q_i'$.

For every q_i, we provide a state r_i for which $r \overset{\varepsilon}{\Rightarrow} r_i$ and $q_i \mathcal{R}_2 r_i$. Consider the given sequence $q{=}q_i^1 \relbar\!\big(\varphi_i^1, \tau, \psi_i^1\big)\!\!\rightarrowtail q_i^2 \ldots q_i^n \relbar\!\big(\varphi_i^n, \tau, \psi_i^n\big)\!\!\rightarrowtail q_i^{n+1} = q_i$ for some $n \geq 0$. Starting from $r_i^1 =_{\text{df}} r$, we iteratively find $r_i^2, \ldots r_i^{n+1}$ by Definition 8 (iii) such that $r_i^k \overset{\varepsilon}{\Rightarrow} r_i^{k+1}$ and $q_i^{k+1} \mathcal{R}_2 r_i^{k+1}$ for $1 \leq k \leq n$. We conclude that $r \overset{\varepsilon}{\Rightarrow} r_i$ and $q_i \mathcal{R}_2 r_i$ for $r_i =_{\text{df}} r_i^{n+1}$.

For every transition of the family $q_i \relbar\!(\varphi_i, a!, \psi_i)\!\!\rightarrowtail q_i'$, there exists a family of transitions $r_{i,j} \relbar\!\big(\varphi_{i,j}, a!, \psi_{i,j}\big)\!\!\rightarrowtail r_{i,j}'$ with $\varphi_i \Rightarrow \bigvee_j \varphi_{i,j}$ and, for every σ with $\sigma \models \varphi_i$, there is at least one transition $r_{i,j} \relbar\!\big(\varphi_{i,j}, a!, \psi_{i,j}\big)\!\!\rightarrowtail r_{i,j}'$ of the family such that $\sigma \models \varphi_{i,j}$, $r_i \overset{\varepsilon}{\Rightarrow} r_{i,j}$, $\psi_{i,j}(\sigma) \Rightarrow \psi_i(\sigma)$ and $q_i' \mathcal{R}_2 r_{i,j}'$. Thus, we have a family $r_{i,j} \relbar\!\big(\varphi_{i,j}, a!, \psi_{i,j}\big)\!\!\rightarrowtail r_{i,j}'$ with $\varphi \Rightarrow \bigvee_i \varphi_i \Rightarrow \bigvee_i \bigvee_j \varphi_{i,j}$ and, for every σ with $\sigma \models \varphi$, there exists a transition $q_i \relbar\!(\varphi_i, a!, \psi_i)\!\!\rightarrowtail q_i'$ with $\sigma \models \varphi_i$ and a transition $r_{i,j} \relbar\!\big(\varphi_{i,j}, a!, \psi_{i,j}\big)\!\!\rightarrowtail r_{i,j}'$ of the family of transitions of R such that $\sigma \models \varphi_{i,j}$, $r \overset{\varepsilon}{\Rightarrow} r_i \overset{\varepsilon}{\Rightarrow} r_{i,j}$, $\psi_{i,j}(\sigma) \Rightarrow \psi_i(\sigma) \Rightarrow \psi(\sigma)$ and $p'\mathcal{R}r'_{i,j}$. This family matches $p \dashv\!(\varphi, a!, \psi)\!\mapsto p'$. □

Note that the decidability of alternating simulation for finite-state IAMs depends on the logic chosen for expressing pre- and post-conditions. Above we picked first-order logic, for which implication is known to be undecidable. We leave the question as to which logic is most suitable for future work.

Related work We close this section by discussing the differences of our IAM setting to the most closely related work, namely the publications of Mouelhi et al. [27], Bauer et al. [2] (which extends [1]) and de Alfaro et al. [11]:

Mouelhi et al. The paper [27] has served as our original motivation for IAM. We adopted Mouelhi et al.'s symbolic handling of data and their intuition of outputs triggering operations, with the exception that they fix a pre- and post-condition for each action, while we do so for each action instance. This results in larger expressiveness by being able to consider different caller states, as does our addition of primed variables. However, the notion of communication error—due to a missing input or because the pre- or post-conditions of caller and callee do not fit—are the same in IAM as in [27]. The latter should be checked statically for each action a, and in case of a misfit, each state with an a-output directly gives rise to an error. Otherwise, only the presence of input a has to be checked. Additionally, the variation of alternating refinement studied in [27] does not consider transition families when matching and, thus, is unnecessarily limited. Mouelhi et al. also do not prove the precongruence result wrt. parallel composition, and leave important terms such as compatibility open to interpretation.

Moreover, ideas such as extending the set of variables when refining are neither explained nor intuitive; we cannot see how then a pre-condition in the specification could imply one in the refinement. As an aside, note that Mouelhi et al. base their work on an early version of IA [12] and not [13], which uses a variation of alternating simulation but appropriately abstracts from internal actions; in particular, output transitions are matched with weak transitions.

Bauer et al. In [2], shared memory is considered for an IA-inspired variation of modal transition systems [21], which increases expressiveness wrt. IA by allowing one to specify outputs that are compulsory and cannot be refined away. Parallel composition is, however, different from IA in that Bauer et al. adopt the pessimistic view of compatibility, as explained in Sect. 1: if a communication error is possible, then parallel composition is undefined. This makes technicalities much easier, e.g. when proving a precongruence result. But it also disregards many meaningful compositions. Therefore, we advocate the optimistic compatibility of IA, where the description of a parallel composition can also be seen as an operating guideline that tells the system environment how to use the components such that errors are avoided.

As in IAM, Bauer et al. use pre- and post-conditions for modelling operation calls and their effect on shared-memory data, and adopt the same notion of communication error. Moreover, the memory available to a component is subdivided in [2] into local variables and external variables. A local variable can be read and written by the component, and it can only be read by another component provided the variable is declared to be visible. An external variable is a local visible variable of another component. Additionally, each operation has a list of parameters with values set by the caller. In case of a synchronization, the operation and the shared variables are internalized but operations keep their names. Hence, such a variable only allows a one-way information flow between only two components; this is not what is usually called shared memory. Moreover, refinement does not abstract from internal computation: each transition, even if is invisible, has to be matched by a single step with the same operation, which is quite restrictive in practice.

A further study of Bauer et al. concerns the comparison of systems via inclusion of their sets of implementations. Such a refinement is often called *thorough* and shown to be implied by the refinement preorder adopted in [2]. Here, a state of an implementation consists of a control state and a data state for the local variables. This is not unlike our ground semantics (see Sect. 7), but the implementation is still open since the values of external variables are not fixed. Instead, such a valuation is part of each transition label.

In addition, there is no discussion in [2] that one transition can end in some concrete data state for the external variables, while the next transition might be performed from a different data state. Surprisingly, there is also no requirement that pre- and post-conditions must be satisfiable: if a data state σ satisfies some pre-condition but there is no σ' satisfying the new data state, then there might be no implementation for such a specification.

De Alfaro et al. Sociable interfaces [11] also employ automata, called *modules*, but in contrast to IA, de Alfaro et al. allow a module to use the same action both as input *and* as output, consider a *multicast* parallel operator, and do *not* abstract from internal actions for refinement. Each module has local and global variables, where local variables describe the module's state. Further, for each action a, a module has a set $W(a)$ of variables that it can modify, which interestingly might contain none of the local variables; in the latter case, a cannot change the state, which is a bit peculiar. In a parallel composition, $W(a)$ always contains all local variables, and an output a modifies the values of $W(a)$, while the input requires that these are acceptable. Hence, sociable interfaces are *not* meant to model operation calls.

Sociable interfaces are equipped with a semantics in terms of a two-player game, where one player controls input actions and the other controls output actions. Moreover, the module states are partitioned into the two sets *iSet* (called φ^I in [11]) and *oSet*, with the aim of the input (output) player being that the game does not leave *iSet* (*oSet*). In the product of two modules, the good states are defined, in standard terms, as the non-error states. On the basis of the good states, the old *iSet*s and the winning states for input, the *iSet* of the product is modified and this gives the parallel composition. However, the relation of this construction to standard pruning is not clarified in any way in [11], and this is not obvious to us.

To conclude, we point out that it might well be useful to add local variables to IAM in the future, similar to [2,11]. But, for the time being, this would only obscure our issue, namely to study an IA-like setting where data is communicated by shared memory while operations are triggered by synchronizing actions. From the perspective of the present article, local variables are just syntactic sugar: if an interface has local variables x and y, we can choose states of the form s_{xy}, where the index gives the current variable values. For this to work, it is important that our IAM theory treats infinite state systems.

4 Parallel composition

This section extends the parallel operator of IA, as outlined above, to IAM.[2] Two IAMs P and Q can be composed, if each action that is in the alphabets of both is an output action in one IAM and an input action in the other. When the overall system composed of P and Q is in data state σ and P is in control state p with $p \,{-}\!\!(\varphi_p, a!, \psi_p)\!\!\rightarrow p'$ such that $\sigma \models \varphi_p$, then P can perform action $a!$. If Q is in state q and $a \in I_Q$, a transition $q \,{-}\!\!(\varphi_q, a?, \psi_q)\!\!\rightarrow q'$ such that $\sigma \models \varphi_q$ is required, i.e. Q has to perform the operation invoked: such a transition provides a new data state $\overline{\sigma}$ with $\sigma, \overline{\sigma}' \models \psi_q$. If each such $\overline{\sigma}$ meets the expectation of P, i.e. $\sigma, \overline{\sigma}' \models \psi_p$ (and $\psi_q(\sigma) \Rightarrow \psi_p(\sigma)$ in Definition 1(P!?)), the data state changes from σ to $\overline{\sigma}$ and the components

[2] Recall that all IAMs have the same set V of variables.

jointly move from $\langle p, q\rangle$ to $\langle p', q'\rangle$. Otherwise, we have a communication mismatch, and $\langle p, q\rangle$ is an error state. This also holds if there is no transition $q \xrightarrow{(\varphi_q, a?, \psi_q)} q'$ with $\sigma \models \varphi_q$. As for IA, all error states are pruned in the parallel composition of P and Q; the same applies for those states from which reaching an error cannot be prevented by any system environment.

Definition 10 (*Parallel Product*) IAMs $(P, I_P, O_P, \rightarrow_P, p_0)$, $(Q, I_Q, O_Q, \rightarrow_Q, q_0)$ are *composable* if $O_P \cap O_Q = \emptyset = I_P \cap I_Q$. The *product* $P \otimes Q$ of composable IAMs is defined as $(P \times Q, I, O, \rightarrow, \langle p_0, q_0\rangle)$, where $I =_{\text{df}} (I_P \cup I_Q)\backslash(A_P \cap A_Q)$, $O =_{\text{df}} (O_P \cup O_Q)\backslash(A_P \cap A_Q)$, and the transition relation $\rightarrow$ is the least relation satisfying the following rules:

(Pl) $\langle p, q\rangle \xrightarrow{(\varphi_p, \alpha, \psi_p)} \langle p', q\rangle$ if $p \xrightarrow{(\varphi_p, \alpha, \psi_p)}_P p'$ and $\alpha \notin A_Q$;
(Pr) $\langle p, q\rangle \xrightarrow{(\varphi_q, \alpha, \psi_q)} \langle p, q'\rangle$ if $q \xrightarrow{(\varphi_q, \alpha, \psi_q)}_Q q'$ and $\alpha \notin A_P$;
(P!?) $\langle p, q\rangle \xrightarrow{(\varphi_p \wedge \varphi_q, \tau, \psi_q)} \langle p', q'\rangle$ if $\varphi_p \wedge \varphi_q$ *sat*, $p \xrightarrow{(\varphi_p, a!, \psi_p)}_P p'$, $q \xrightarrow{(\varphi_q, a?, \psi_q)}_Q q'$ and $\psi_q(\sigma) \Rightarrow \psi_p(\sigma)$ for all σ with $\sigma \models \varphi_p \wedge \varphi_q$;
(P?!) $\langle p, q\rangle \xrightarrow{(\varphi_p \wedge \varphi_q, \tau, \psi_p)} \langle p', q'\rangle$ if $\varphi_p \wedge \varphi_q$ *sat*, $p \xrightarrow{(\varphi_p, a?, \psi_p)}_P p'$, $q \xrightarrow{(\varphi_q, a!, \psi_q)}_Q q'$ and $\psi_p(\sigma) \Rightarrow \psi_q(\sigma)$ for all σ with $\sigma \models \varphi_p \wedge \varphi_q$.

The asynchronous interleaving of autonomous transitions of P and Q is modelled by Rules (Pl) and (Pr). Rules (P!?) and (P?!) govern synchronization between P and Q. The pre-condition of the synchronized transition is the conjunction of the pre-conditions of the two involved transitions, as both must be fulfilled in the system's current data state for P and Q to engage in their transitions. To model the invocation of an operation, the input-transition may alter the data state; accordingly, the synchronized transition is labelled with the post-condition ψ_q. This requires that the post-condition of the output-transition is respected, leading to the implication $\psi_q(\sigma) \Rightarrow \psi_p(\sigma)$ as explained above. The text "$\psi_q(\sigma) \Rightarrow \psi_p(\sigma)$ for all σ with $\sigma \models \varphi_p \wedge \varphi_q$" could be replaced by $\varphi_p \wedge \varphi_q \wedge \psi_q \Rightarrow \psi_p$.

Here, as already announced above, we deviate from the IA-approach—as do the authors of [9,27]—in order to model operation calls properly as follows. A synchronization has two phases: it is triggered by the caller's output in the first phase, which requires a matching input; in the second phase, the new data state is produced on the input side (i.e. by the callee), requiring the outputting component (i.e. the caller) to accept it. Violation of one of the requirements leads to an error.

Definition 11 (*Parallel Composition*) Given a parallel product $P \otimes Q$ of IAMs P and Q, a state $\langle p, q\rangle$ is an *error state* if at least one of the following conditions is satisfied:

(E!?φ) $p \xrightarrow{(\varphi_p, a!, \psi_p)} p'$ with $a? \in I_Q$, and there exists σ s.t. $\sigma \models \varphi_p$ but $\sigma \not\models \varphi_q$ for all transitions $q \xrightarrow{(\varphi_q, a?, \psi_q)} q'$;
(E?!φ) $q \xrightarrow{(\varphi_q, a!, \psi_q)} q'$ with $a? \in I_P$, and there exists σ s.t. $\sigma \models \varphi_q$ but $\sigma \not\models \varphi_p$ for all transitions $p \xrightarrow{(\varphi_p, a?, \psi_p)} p'$;
(E!?ψ) $p \xrightarrow{(\varphi_p, a!, \psi_p)} p'$ and there exists a transition $q \xrightarrow{(\varphi_q, a?, \psi_q)} q'$ and σ s.t. $\sigma \models \varphi_p \wedge \varphi_q$ and $\psi_q(\sigma) \not\Rightarrow \psi_p(\sigma)$, i.e. $[\![\psi_q(\sigma)]\!] \not\subseteq [\![\psi_p(\sigma)]\!]$;
(E?!ψ) $q \xrightarrow{(\varphi_q, a!, \psi_q)} q'$ and there exists a transition $p \xrightarrow{(\varphi_p, a?, \psi_p)} p'$ and σ s.t. $\sigma \models \varphi_p \wedge \varphi_q$ and $\psi_p(\sigma) \not\Rightarrow \psi_q(\sigma)$, i.e. $[\![\psi_p(\sigma)]\!] \not\subseteq [\![\psi_q(\sigma)]\!]$.

The set $E \subseteq P \times Q$ of *illegal states* is the least set containing all error states and the states $\langle p, q\rangle$ satisfying $\langle p, q\rangle \xrightarrow{(\varphi, \alpha!, \psi)} \langle p', q'\rangle$ with $\langle p', q'\rangle \in E$; this is also called *Rule* (Epb).

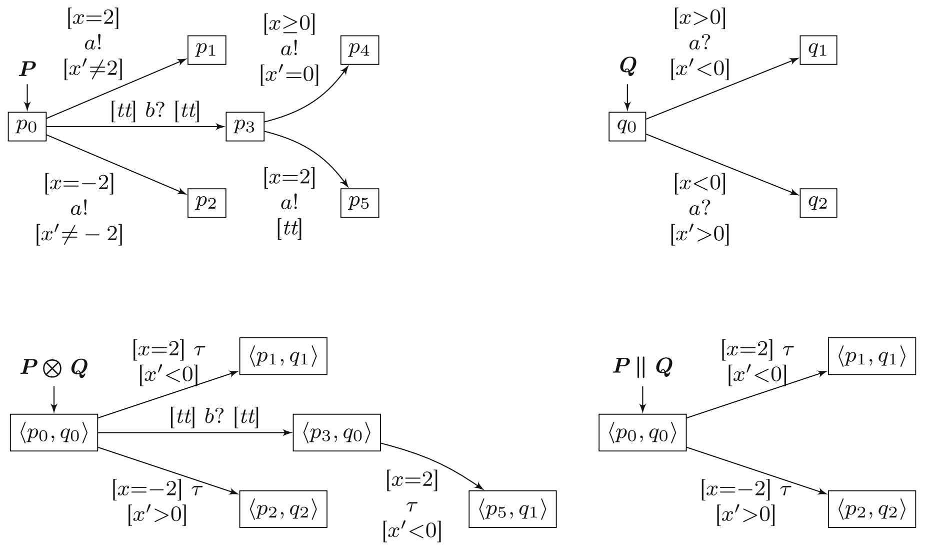


Fig. 7 Parallel composition of IAMs P, Q, where $I_P = \{b\}$, $O_P = I_Q = \{a\}$ and $O_Q = \emptyset$

The parallel composition $P \parallel Q$ is obtained by *pruning* the illegal states. We also refer to the set of illegal states of $P \otimes Q$ as its *E-set*. The notions of state and component compatibility are as in IA.

Rule (E!?φ) describes a situation where p performs $a!$ in some data state σ where q has no $a?$-transition enabled by σ. If (E!?ψ) applies, q has such an enabled $a?$-transition and it is unique by data-determinism; but this transition can update the shared memory in a way that violates the assumption of the $a!$-transition.

Note that the above definition coincides with the corresponding one for IA, if we consider IAMs where only *tt* occurs as pre- and post-condition. The latter guarantees that the side conditions of the parallel product, i.e. $\varphi_p \wedge \varphi_q$ *sat* and $\psi_q(\sigma) \Rightarrow \psi_p(\sigma)$ for all σ with $\sigma \models \varphi_p \wedge \varphi_q$ are satisfied trivially. Rule (E!?φ) states that q has no $a?$-transition, giving rise to an error as for IA, and Rule (E!?ψ) is never applicable.

Figure 7 shows an example of a parallel product $P \otimes Q$ and the effect of pruning. IAMs P, Q synchronize on a where, e.g. $p_0 \multimap\!\!\big(x = 2, a!, x' \neq 2\big)\!\!\rightarrow p_1$ can only synchronize with the upper transition $q_0 \multimap\!\!\big(x > 0, a?, x' < 0\big)\!\!\rightarrow q_1$ of Q: data state σ with $\sigma(x) = 2$ satisfies both pre-conditions, and $x' < 0 \Rightarrow x' \neq 2$ for the post-conditions. Note that the pre-condition $x < 0$ of the lower $a?$-transition from q_0 contradicts $x = 2$. The synchronization results in transition $\langle p_0, q_0\rangle \multimap\!\!\big(x = 2, \tau, x' < 0\big)\!\!\rightarrow \langle p_1, q_1\rangle$ in $P \otimes Q$. IAM P can perform its $b?$-transition independently. Transition $p_3 \multimap\!\!\big(x \geq 0, a!, x' = 0\big)\!\!\rightarrow p_4$ cannot be matched by an $a?$-transition of q_0 for $x = 0$, so (E!?φ) applies to $\langle p_3, q_0\rangle$. For $x > 0$, the upper transition from q_0 has a fitting pre-condition, but the new value of x could be -1; this contradicts $x' = 0$, so (E!?ψ) applies, too. Hence, $\langle p_3, q_0\rangle$ is an error state so that transitions $\langle p_0, q_0\rangle \multimap\!\!(tt, b?, tt)\!\!\rightarrow \langle p_3, q_0\rangle$ and $\langle p_3, q_0\rangle \multimap\!\!\big(x = 2, \tau, x' < 0\big)\!\!\rightarrow \langle p_5, q_1\rangle$ in $P \otimes Q$ are pruned and not part of $P \parallel Q$.

It is important to observe that, in contrast to IA, an error may arise in a parallel composition due to an output enabled in some data state, although a τ-transition in which the output participates may still exist in the product due to some different data state. For example, an output $a!$ might be offered when variable x has value 1 or 2, while the communication partner only has input $a?$ if x has value 1. This results in an error due to (E!?φ), but the product still has a synchronizing τ-transition with pre-condition $x = 1$.

Clearly, one expects parallel composition to be associative and commutative. While the latter is obvious, the former is not trivial in IA-like approaches. For our IAM-setting, the associativity proof is given in Sect. 6.

Proposition 12 (Associativity & Commutativity) *If P, Q and R are pairwise composable IAMs, then $(P \parallel Q) \parallel R$ and $P \parallel (R \parallel Q)$ are undefined or $(P \parallel Q) \parallel R$ and $P \parallel (R \parallel Q)$ are isomorphic. Analogously, parallel composition is commutative.*

Preorder $\sqsubseteq_{IAM}$ is compositional wrt. parallel composition, which follows directly from the properties of our embedding of IAM to IA in the next section (see Corollary 18). Before showing this embedding, we wish to technically justify the notion of pruning which realizes IAM's optimistic notion of compatibility that it inherits from IA. We follow the approach adopted by us for Modal Interface Automata (MIA) in [5], which is based on so-called *legal*, or helpful, *environments* as first introduced for IA in [13].

Definition 13 (*Legal Environment*) A *legal environment* for IAMs P, Q is an IAM U such that

(i) U is composable with $P \otimes Q$;
(ii) The reachable states of $(P \otimes Q) \otimes U$ contain no error states, where $((r_1, r_2), r_3)$ is an error state, if some (r_i, r_j) with $i \neq j$ is an error state according to Definition 11.

Intuitively, U prevents $P \otimes Q$ from reaching an error state, so that one can think of a legal environment as modelling a user with whom each desired system assembled from concurrently running IAMs is composed at the end. Technically, our definition of legal environment is more general than the one of de Alfaro and Henzinger in [13], because we only demand composability for the signature. In the IA setting, de Alfaro and Henzinger showed that two IAs are compatible if and only if there is a legal environment for them. As in MIA, we can give a better justification for pruning:

Proposition 14 (Pruning Justification) *Let P, Q be IAMs.*

(i) *P and Q are compatible if and only if there exists a legal environment for them.*
(ii) *$(P \otimes Q) \otimes U = (P \parallel Q) \otimes U$, for any legal environment U for P and Q.*

Hence, pruning only removes behaviour from $P \otimes Q$ that is never reached in any legal environment, i.e. when the composition is used properly.

Proof (i) "$\Rightarrow$": If P and Q are compatible, then $P \otimes Q$ has no locally reachable error states. Composing it with an IAM U that has only one state, no transitions, no inputs, and all inputs of $P \otimes Q$ as outputs, yields essentially only those states that are locally reachable from (p_0, q_0). Thus, U is a legal environment for P and Q.
"$\Leftarrow$": Assume towards a contradiction that P and Q are incompatible, i.e. $P \otimes Q$ has a locally reachable error state (p, q). Then, for any IAM U, the IAM $(P \otimes Q) \otimes U$ either has a locally reachable error state $((p, q), u)$, or there is a first output transition on the path to (p, q) that U prevents by not providing the corresponding input transition. This also results in a locally reachable error state in $(P \otimes Q) \otimes U$. Either way, U is no legal environment.

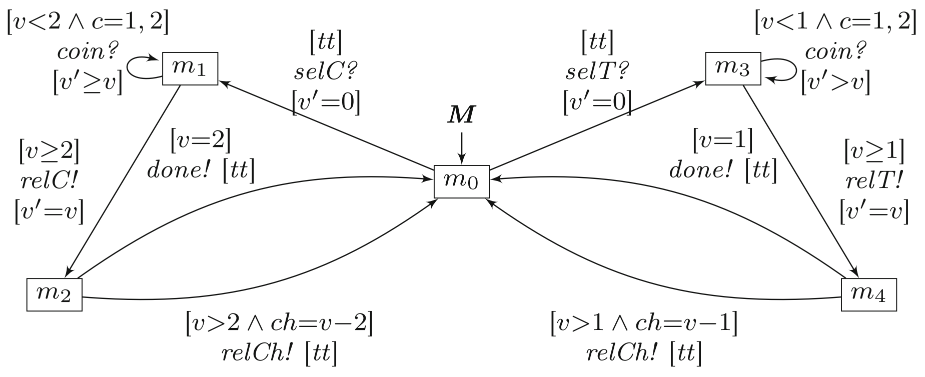

M
m_0
m_1
m_2
m_3
m_4
$[v{<}2 \wedge c{=}1,2]$ coin? $[v'{\geq}v]$
$[tt]$ selC? $[v'{=}0]$
$[tt]$ selT? $[v'{=}0]$
$[v{<}1 \wedge c{=}1,2]$ coin? $[v'{>}v]$
$[v{\geq}2]$ relC! $[v'{=}v]$
$[v{=}2]$ done! $[tt]$
$[v{=}1]$ done! $[tt]$
$[v{\geq}1]$ relT! $[v'{=}v]$
$[v{>}2 \wedge ch{=}v{-}2]$ relCh! $[tt]$
$[v{>}1 \wedge ch{=}v{-}1]$ relCh! $[tt]$

Fig. 8 Specification *M* of the vending machine

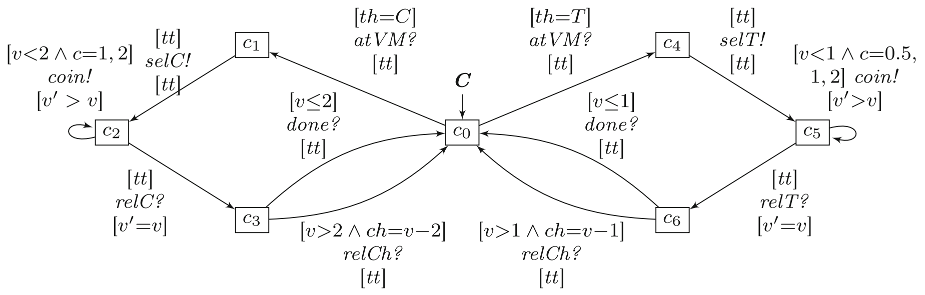

C
c_0
c_1
c_2
c_3
c_4
c_5
c_6
$[v{<}2 \wedge c{=}1,2]$ coin! $[v' > v]$
$[tt]$ selC! $[tt]$
$[th{=}C]$ atVM? $[tt]$
$[th{=}T]$ atVM? $[tt]$
$[tt]$ selT! $[tt]$
$[v{<}1 \wedge c{=}0.5, 1,2]$ coin! $[v'{>}v]$
$[v{\leq}2]$ done? $[tt]$
$[v{\leq}1]$ done? $[tt]$
$[tt]$ relC? $[v'{=}v]$
$[tt]$ relT? $[v'{=}v]$
$[v{>}2 \wedge ch{=}v{-}2]$ relCh? $[tt]$
$[v{>}1 \wedge ch{=}v{-}1]$ relCh? $[tt]$

Fig. 9 Specification *C* of the vending machine's customer

(ii) The claim can only fail if there is a state $(p, q) \in P \otimes Q$ that is illegal due to error (p', q') but exists in $(P \otimes Q) \otimes U$ in the form of some $((p, q), u)$. As just argued, the path from (p, q) to (p', q') still exists in $(P \otimes Q) \otimes U$, or is prevented by U at some stage leading to another error state. Either way, U is not a legal environment.

□

The correctness of pruning has also been investigated in the context of a linear-time semantics for IA in [6], where a problem with the original version of IA [12] in the presence of input-nondeterminism is pointed out and fixed.

5 Example

This section first illustrates the handling of data in IAM using a variation of the vending machine example of Sect. 2. It then briefly introduces an initial, prototypic toolset-based around our IAM theory, which supports modelling and simulating IAMs, debugging communication errors, as well as checking and reasoning about model refinements.

5.1 The vending machine example, expanded with shared memory

The variation of our vending machine example (see Figs. 8, 9) considers, in particular, the payment operation *coin* and the change giving operation *relCh*, the latter of which is supple-

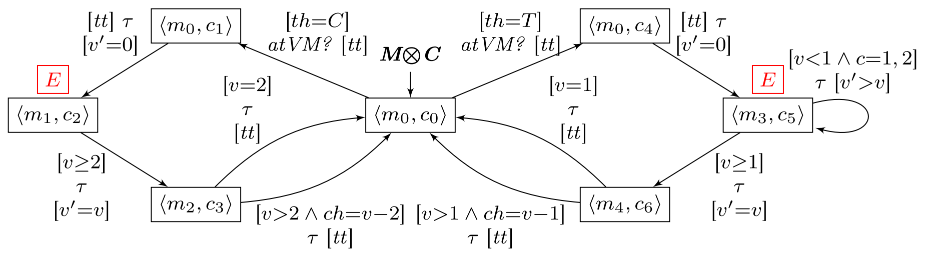


Fig. 10 Parallel product $M \otimes C$

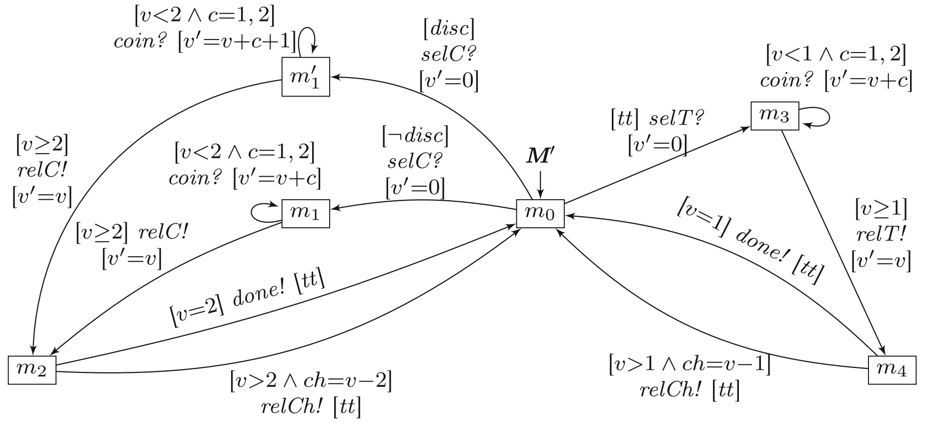

M'
m_0
m_1'
m_1
m_2
m_3
m_4
$[v<2 \wedge c=1,2]\ coin?\ [v'=v+c+1]$
$[disc]\ selC?\ [v'=0]$
$[v<1 \wedge c=1,2]\ coin?\ [v'=v+c]$
$[v\geq 2]\ relC!\ [v'=v]$
$[v<2 \wedge c=1,2]\ coin?\ [v'=v+c]$
$[\neg disc]\ selC?\ [v'=0]$
$[tt]\ selT?\ [v'=0]$
$[v\geq 2]\ relC!\ [v'=v]$
$[v=1]\ done!\ [tt]$
$[v\geq 1]\ relT!\ [v'=v]$
$[v=2]\ done!\ [tt]$
$[v>2 \wedge ch=v-2]\ relCh!\ [tt]$
$[v>1 \wedge ch=v-1]\ relCh!\ [tt]$

Fig. 11 IAM-refinement M' of vending machine M

mented with action *done* in case no change is given. Variable *th* stores whether customer C is thirsty for coffee (value C) or tea (value T); hence, this information is no longer encoded in action *atVM*. Variable v stores the payment that the customer has made so far. Upon receiving a selection *selC* or *selT*, machine M sets this value to 0. Variable v is updated in operation *coin*; for coffee (tea), the machine never lowers (always raises) the value, while the customer always expects an increase. Variable c holds the value of the inserted coin, which can be 50¢, 1€ or 2€. Note that, implicitly, c is set by the customer when selecting a coin. The drink is dispensed (operation *relC*) when v reaches the resp. price of 2€ for coffee and 1€ for tea. Finally, the change computed in *ch* is returned in operation *relCh*.

The product $M \otimes C$ is shown in Fig. 10. We discuss some of the transitions. In the τ-transition from $\langle m_0, c_4\rangle$, actions *selT!* and *selT?* synchronize and M sets v to 0, while C does not care about the value. The τ-transition from $\langle m_4, c_6\rangle$ originating from the synchronization on *done* is triggered by M if $v = 1$; customer C would also accept a lesser value. The pre-condition of the τ-loop at $\langle m_3, c_5\rangle$ is the conjunction of the pre-conditions of the underlying *coin*-loops. But *coin!* is also triggered when the customer has chosen a 50¢ coin for the cheaper drink. Because the machine does not accept such a coin, $\langle m_3, c_5\rangle$ is an error state according to Rule (E?!φ). The *coin*-transitions at m_1 and c_2 do not give rise to a τ-loop because M might keep the value of v, violating the assumption $v' > v$ of C. Also $\langle m_1, c_2\rangle$ is an error state since $v' \geq v$ does not imply $v' > v$. Consequently, $M \parallel C$ just consists of $\langle m_0, c_0\rangle$. While M and C are formally compatible, this is only the case if the environment ensures that the customer is never thirsty in the vicinity of the machine.

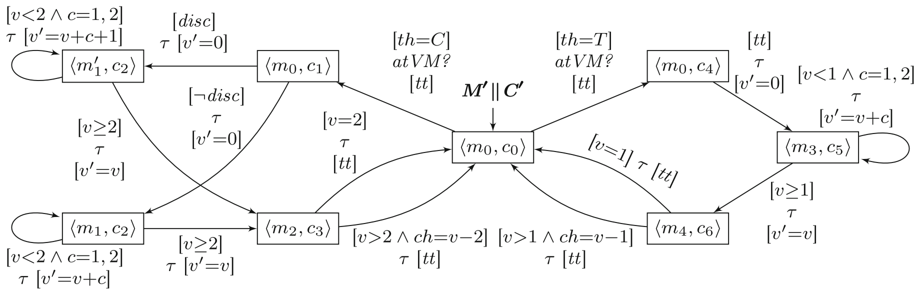

[v<2 ∧ c=1, 2] τ [v'=v+c+1]
[$disc$] τ [v'=0]
⟨m'_1, c_2⟩
⟨m_0, c_1⟩
[th=C] atVM? [tt]
[th=T] atVM? [tt]
⟨m_0, c_4⟩
[tt] τ [v'=0]
[v<1 ∧ c=1, 2] τ [v'=v+c]
M′ || C′
[¬$disc$] τ [v'=0]
[v≥2] τ [v'=v]
[v=2] τ [tt]
⟨m_0, c_0⟩
[v=1] τ [tt]
⟨m_3, c_5⟩
[v≥1] τ [v'=v]
⟨m_1, c_2⟩
⟨m_2, c_3⟩
⟨m_4, c_6⟩
[v<2 ∧ c=1, 2] τ [v'=v+c]
[v≥2] τ [v'=v]
[v>2 ∧ ch=v−2] τ [tt]
[v>1 ∧ ch=v−1] τ [tt]

Fig. 12 Parallel product and composition of M' and C', satisfying $M' \otimes C' = M' \parallel C'$

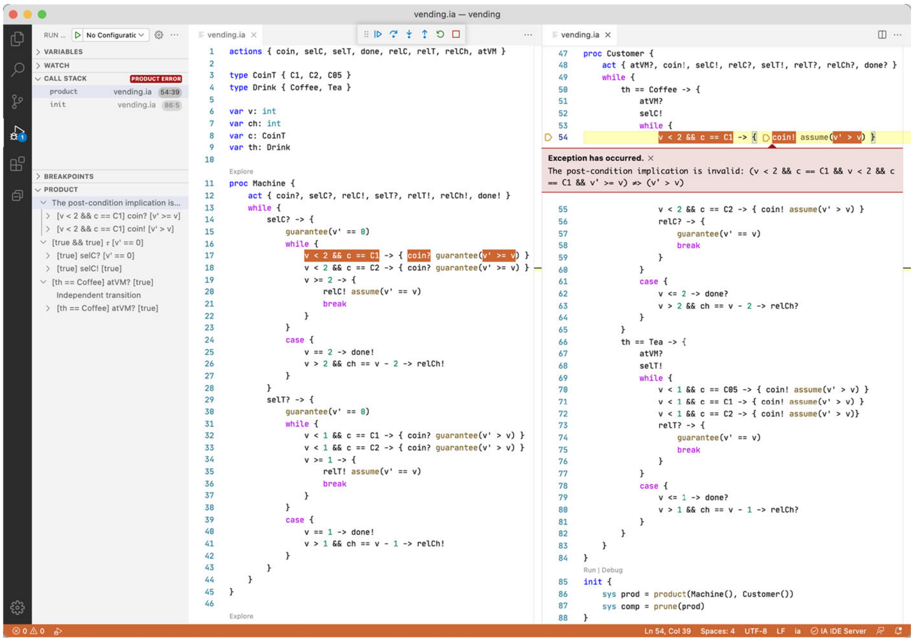
To improve the situation, we IAM-refine M and C to M' and C', respectively. Refinement C' (not shown) is obtained from C by deleting the 0.5-option in the offending pre-condition. Analogously to the IA-example, the customer has learned that 50¢ coins are never accepted. On the machine side, refinement M' (see Fig. 11) computes the correct payment $v' = v + c$ instead of $v' > v$ at m_3. A similar change is made at m_1, but here we have a new feature, showing the splitting of the input *selC?* at m_0. The refined machine M' is configurable to give a discount if variable *disc* is true. In this case, payment v is incremented by one additional Euro for each coin, which actually happens exactly once. Observe that data-determinism is maintained since $disc \wedge \neg disc$ unsat, while Definition 8(i) is satisfied because $disc \vee \neg disc$ implies *tt*. With these changes, we obtain $M' \otimes C' = M' \parallel C'$, as displayed in Fig. 12. Because $M' \sqsubseteq_{IAM} M$ and $C' \sqsubseteq_{IAM} C$, we also have $M' \parallel C' \sqsubseteq_{IAM} M \parallel C$ and, indeed, $\langle m_0, c_0\rangle$ only has additional inputs in $M' \parallel C'$. Note that the behaviour after a new input can be arbitrary, because inputs of the refinement do not have to be matched according to Definition 8.

5.2 A prototypic IAM toolset

In order to demonstrate that tool support for the IAM interface theory is conceivable, we have developed an initial, prototypical toolset, to which we refer as *IAM Toolset*. The toolset is developed using various open-source technologies described below, is itself open source and available for download and further development via GitHub [28] and comprises approx. 12,000 lines of code. It runs on Windows, Linux and macOS with installed JDK11. We have used the toolset to check small examples that we had developed by pencil and paper, such as the vending machine example above. Note that we do neither claim that the IAM Toolset is fit for practical use by software engineers as is, nor that it scales to real-world examples of concurrent systems.

Textual language To specify interfaces, we have developed a textual syntax for IAM, have used the popular *Xtext* framework (https://www.eclipse.org/Xtext/) to realize this domain-specific modelling language, and written an extension for Microsoft's *Visual Studio Code* (VS Code) editor (https://code.visualstudio.com). This has led to a state-of-the-art IDE for IAM, with a modern editor that supports features such as syntax highlighting and code completion, and a compiler—actually, more of an interpreter—for transforming the textual language into IAM automata.

The textual language allows one to specify the underlying universe of global actions (keyword `actions`), the shared global variables (keyword `var`) that are typed using built-



vending.ia — vending

RUN ... No Configuratic

VARIABLES
WATCH
CALL STACK PRODUCT ERROR
```
product    vending.ia 54:39
init       vending.ia 86:5
```
BREAKPOINTS
PRODUCT
```
The post-condition implication is...
  [v < 2 && c == C1] coin? [v' >= v]
  [v < 2 && c == C1] coin! [v' > v]
[true && true] τ [v' == 0]
  [true] selC? [v' == 0]
  [true] selC! [true]
[th == Coffee] atVM? [true]
  Independent transition
  [th == Coffee] atVM? [true]
```

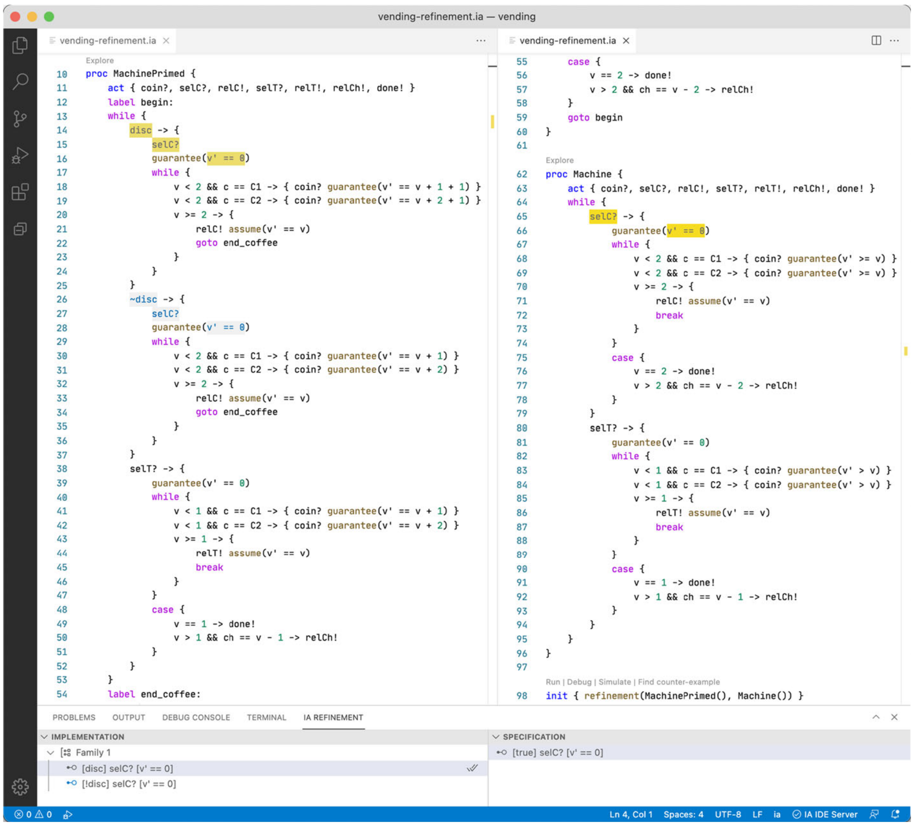
vending.ia
```
actions { coin, selC, selT, done, relC, relT, relCh, atVM }

type CoinT { C1, C2, C05 }
type Drink { Coffee, Tea }

var v: int
var ch: int
var c: CoinT
var th: Drink

Explore
proc Machine {
    act { coin?, selC?, relC!, selT?, relT!, relCh!, done! }
    while {
        selC? -> {
            guarantee(v' == 0)
            while {
                v < 2 && c == C1 -> { coin? guarantee(v' >= v) }
                v < 2 && c == C2 -> { coin? guarantee(v' >= v) }
                v >= 2 -> {
                    relC! assume(v' == v)
                    break
                }
            }
            case {
                v == 2 -> done!
                v > 2 && ch == v - 2 -> relCh!
            }
        }
        selT? -> {
            guarantee(v' == 0)
            while {
                v < 1 && c == C1 -> { coin? guarantee(v' > v) }
                v < 1 && c == C2 -> { coin? guarantee(v' > v) }
                v >= 1 -> {
                    relT! assume(v' == v)
                    break
                }
            }
            case {
                v == 1 -> done!
                v > 1 && ch == v - 1 -> relCh!
            }
        }
    }
}

Explore
```

vending.ia
```
proc Customer {
    act { atVM?, coin!, selC!, relC?, selT!, relT?, relCh?, done? }
    while {
        th == Coffee -> {
            atVM?
            selC!
            while {
                v < 2 && c == C1 -> { coin! assume(v' > v) }
```
Exception has occurred.
```
The post-condition implication is invalid: (v < 2 && c == C1 && v < 2 && c
== C1 && v' >= v) ≠> (v' > v)
```
```
                v < 2 && c == C2 -> { coin! assume(v' > v) }
                relC? -> {
                    guarantee(v' == v)
                    break
                }
            }
            case {
                v <= 2 -> done?
                v > 2 && ch == v - 2 -> relCh?
            }
        }
        th == Tea -> {
            atVM?
            selT!
            while {
                v < 1 && c == C05 -> { coin! assume(v' > v) }
                v < 1 && c == C1 -> { coin! assume(v' > v) }
                v < 1 && c == C2 -> { coin! assume(v' > v)}
                relT? -> {
                    guarantee(v' == v)
                    break
                }
            }
            case {
                v <= 1 -> done?
                v > 1 && ch == v - 1 -> relCh?
            }
        }
    }
}
Run | Debug
init {
    sys prod = product(Machine(), Customer())
    sys comp = prune(prod)
}
```
Ln 54, Col 39 Spaces: 4 UTF-8 LF ia IA IDE Server

Fig. 13 Debugging the parallel product of vending machine M and customer C

in primitive types such as `int` and `bool` or custom-defined enumeration types (keyword `type`), and the templates of interface processes (keyword `proc`). It also has an initialization section (keyword `init`), where processes can be instantiated from templates (keyword `sys`), composed in parallel (keywords `product` and `composition` for the parallel product and composition, respectively) and checked for refinement (keyword `refinement`). In fact, there is also an explicit pruning operator (keyword `prune`) such that `composition` is simply `product` followed by `prune`. While Xtext assist us in realizing our language interpreter's front-end (approx. 1100 lines of code), the backend and, thus, all IAM operators are implemented solely in the *Kotlin* programming language (approx. 9400 lines of code). The various commands for instantiation, composition and refinement can also be executed directly from VS Code (approx. 1350 lines of code in *TypeScript*) or the command line. For checking static semantics constraints, we frequently call the *Z3* SMT solver (https://github.com/Z3Prover/z3), which we also use in the context of the product operator and IAM refinement for checking logic implications that involve arithmetic expressions over shared variables.

Figure 13 shows, in the rightmost two columns, snippets of the machine and customer specifications of our vending machine example of the previous section, now cast in the syntax of the IAM Toolset. The process template `Machine` first defines its input and output alphabet and then defines interface behaviour via a guarded-command language enriched with control structures such as `while` loops (with optional `break`s), `case` statements, sequential composition (via newlines or semicolons), or `goto` (jump to a `label`). Guards are on the left of symbol `->` and constitute the pre-condition of the first action following symbol `->`. Post-conditions immediately follow an action and are labelled with keyword `assume` for post-conditions of output actions and `guarantee` for post-conditions of input actions; the rationale is that a post-condition of an output action $a!$ abstractly specifies how



vending-refinement.ia — vending

```
Explore
proc MachinePrimed {
    act { coin?, selC?, relC!, selT?, relT!, relCh!, done! }
    label begin:
    while {
        disc -> {
            selC?
            guarantee(v' == 0)
            while {
                v < 2 && c == C1 -> { coin? guarantee(v' == v + 1 + 1) }
                v < 2 && c == C2 -> { coin? guarantee(v' == v + 2 + 1) }
                v >= 2 -> {
                    relC! assume(v' == v)
                    goto end_coffee
                }
            }
        }
        ~disc -> {
            selC?
            guarantee(v' == 0)
            while {
                v < 2 && c == C1 -> { coin? guarantee(v' == v + 1) }
                v < 2 && c == C2 -> { coin? guarantee(v' == v + 2) }
                v >= 2 -> {
                    relC! assume(v' == v)
                    goto end_coffee
                }
            }
        }
        selT? -> {
            guarantee(v' == 0)
            while {
                v < 1 && c == C1 -> { coin? guarantee(v' == v + 1) }
                v < 1 && c == C2 -> { coin? guarantee(v' == v + 2) }
                v >= 1 -> {
                    relT! assume(v' == v)
                    break
                }
            }
            case {
                v == 1 -> done!
                v > 1 && ch == v - 1 -> relCh!
            }
        }
    }
    label end_coffee:
    case {
        v == 2 -> done!
        v > 2 && ch == v - 2 -> relCh!
    }
    goto begin
}

Explore
proc Machine {
    act { coin?, selC?, relC!, selT?, relT!, relCh!, done! }
    while {
        selC? -> {
            guarantee(v' == 0)
            while {
                v < 2 && c == C1 -> { coin? guarantee(v' >= v) }
                v < 2 && c == C2 -> { coin? guarantee(v' >= v) }
                v >= 2 -> {
                    relC! assume(v' == v)
                    break
                }
            }
            case {
                v == 2 -> done!
                v > 2 && ch == v - 2 -> relCh!
            }
        }
        selT? -> {
            guarantee(v' == 0)
            while {
                v < 1 && c == C1 -> { coin? guarantee(v' > v) }
                v < 1 && c == C2 -> { coin? guarantee(v' > v) }
                v >= 1 -> {
                    relT! assume(v' == v)
                    break
                }
            }
            case {
                v == 1 -> done!
                v > 1 && ch == v - 1 -> relCh!
            }
        }
    }
}

Run | Debug | Simulate | Find counter-example
init { refinement(MachinePrimed(), Machine()) }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL IA REFINEMENT

IMPLEMENTATION
- Family 1
  - [disc] selC? [v' == 0]
  - [!disc] selC? [v' == 0]

SPECIFICATION
- [true] selC? [v' == 0]

Ln 4, Col 1 Spaces: 4 UTF-8 LF ia IA IDE Server

Fig. 14 Interactively exploring the refinement between machines M' and M

the called operation *a* may modify the global variables, while a post-condition of an input *a*? abstractly specifies how the global variables are modified when operation *a* is executed. Missing pre- and post-conditions around an action are simply assumed to be *true*.

Debugging The left column in Fig. 13 reveals (see the section labelled *Call Stack*) that this screenshot has been taken after instantiating machine and customer and computing their product, which revealed a communication error between the interfaces. This error relates to the highlighted lines 17 and 54, i.e. on the synchronization on action `coin`. The detailed error message in the red box explains that the assumption is not met by the guarantee, and the section labelled *Product* in the left column provides information via which series of actions the error state is reached from the products initial state. As an aside, we wish to note here that the IAM toolset has standard debugging features, including the setting of breakpoints, that allow engineers the interactive tracing through interface processes, parallel products and parallel compositions.

Refinement checking IAM's notion of refinement, alternating simulation, has a convenient characterization in terms of a two-player game [13], which we exploit so that users can interactively explore in the IAM Toolset whether two processes are related under refinement. A snapshot of such an exploration is shown in Fig. 14 where, in a first move of the game, the initial `selC?` input action of the specification side (right-hand side), i.e. of `Machine`, was

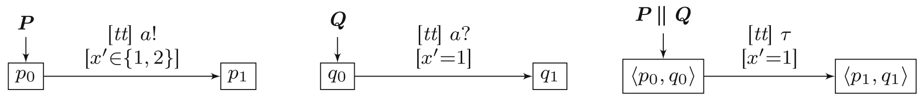


Fig. 15 IAMs P, Q and their parallel composition $P \parallel Q$

chosen. The toolset now highlights the available `selC?` families on the implementation side (left-hand side), i.e. of `MachinePrimed`; in this case, there is a single family consisting of two `selC?` transitions, one with pre-condition `disc` and one with its negation. In general, our tool displays every family (i) consisting of members that have pre- and post-conditions that match the requirements of alternating simulation (cf. Definition 8) and (ii) not including a member already covered by others. The engineer now, in the matching move, picks the only available family and continues the game by investigating the two transitions of the family.

The IAM Toolset also prototypes an automated refinement checker which reduces the refinement problem between two IAMs to solving a Boolean equation system; note that all IAMs specified in the toolset's textual language are finite-state. The reduction is rather straightforward and follows the lines proposed by Mateescu and Oudot in [24] who proposed such a reduction for bisimulation [26]. We then employ the *mCRL2 toolset* [7] (https://www.mcrl2.org) for solving the resulting Boolean equation system. In case a refinement does not hold, the IAM Toolset synthesizes an attacking strategy so that an engineer might play a game against the computer to gain understanding as to why refinement between the given two processes is violated.

6 Embedding IAM into IA

This section shows that, while IAM adds shared memory to the IA theory, it does not alter IA's concept of component compatibility and refinement. IAM is rather an intuitive and possibly finite abstraction for reasoning about possibly infinite data, which maintains the simplicity that has made IA popular. In the following, we develop a behaviour-preserving translation from IAM to IA such that our parallel composition and refinement match exactly those of IA.

While our formal semantics of IAM is based on alternating simulation, we wish to foster intuition here by considering an IAM P to have runs of the form $(\sigma_0, p_0) \xrightarrow{\alpha_0} (\overline{\sigma}_0, p_1)(\sigma_1, p_1) \xrightarrow{\alpha_1} (\overline{\sigma}_1, p_2) \ldots (\sigma_{n-1}, p_{n-1}) \xrightarrow{\alpha_{n-1}} (\overline{\sigma}_{n-1}, p_n)$, for transitions $p_i \overset{(\varphi_i, \alpha_i, \psi_i)}{\longrightarrow} p_{i+1}$ with $\sigma_i \models \varphi_i$ and $\sigma_i, \overline{\sigma}'_i \models \psi_i$, for all $0 \leq i < n$. Note that $\overline{\sigma}_i$ and σ_{i+1} might be different, because interface theories consider *open systems* and P's environment can always interleave and arbitrarily modify the data state; we study and define *closed systems* in Sect. 7.[3]

For our translation, one might consider to integrate data states into actions, so that the above run gives rise to $\alpha_0(\sigma_0, \overline{\sigma}'_0) \ldots \alpha_{n-1}(\sigma_{n-1}, \overline{\sigma}'_{n-1})$ as a sequence of actions, i.e. each $\alpha_i(\sigma_i, \overline{\sigma}'_i)$ is taken to be an (atomic) action. This idea is similar to, e.g. the translation of full CCS into basic CCS in [26].

However, this translation does not work in general. Consider IAMs P, Q of Fig. 15, where $V = \{x\}$, $V' = \{x'\}$ and $\mathbb{D} = \{1, 2\}$. By the above idea, P would have tran-

[3] As an aside, also note that a formal treatment of linear-time semantics for IA can be found in [6,8,30,31]; lifting such semantics to IAM is left to future work.

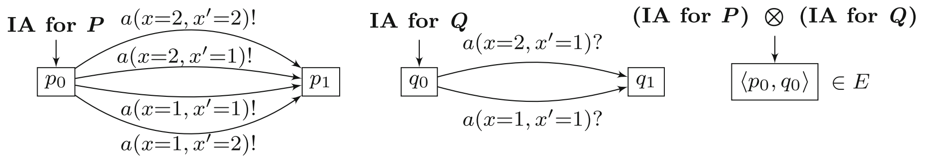


Fig. 16 Intuitive, direct but inadequate translation of P and Q into IA

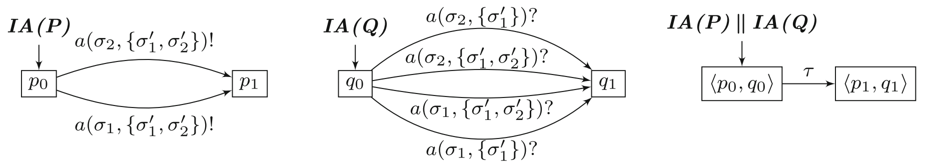

$IA(P)$
$a(\sigma_2, \{\sigma_1', \sigma_2'\})!$
$a(\sigma_1, \{\sigma_1', \sigma_2'\})!$
p_0
p_1
$IA(Q)$
$a(\sigma_2, \{\sigma_1'\})?$
$a(\sigma_2, \{\sigma_1', \sigma_2'\})?$
$a(\sigma_1, \{\sigma_1', \sigma_2'\})?$
$a(\sigma_1, \{\sigma_1'\})?$
q_0
q_1
$IA(P) \parallel IA(Q)$
$\langle p_0, q_0\rangle$
τ
$\langle p_1, q_1\rangle$

Fig. 17 Correct translation of P and Q into IA

sition $p_0 \xrightarrow{a(x=1,x'=2)!} p_1$ in its corresponding IA, as shown in Fig. 16. The best match $q_0 \xrightarrow{a(x=1,x'=1)?} q_1$ in the IA for Q is not good enough; thus, $\langle p_0, q_0\rangle \in E$, i.e. the IAs of P and Q are incompatible. But there is no error according to Rule (E!?ψ) in the operational reality: Q decides about the new data state when answering the request of P. In other words, P would take the transition $p_0 \xrightarrow{a(x=1,x'=1)!} p_1$.

As a remedy, we translate an output-transition $p \multimap\!(\varphi, a!, \psi)\!\rightarrowtail p'$ of IAM P to multiple IA transitions $\{p \xrightarrow{a(\sigma,M)!} p' \mid \sigma \models \varphi,\ [\![\psi(\sigma)]\!] \subseteq M \subseteq [\![V']\!]\}$, and analogously for input-transitions. Pre- and post-conditions of τ-transitions are simply suppressed. The reasons for this choice are as follows. The first idea would be to use $p \xrightarrow{a(\sigma,M)!} p'$ with $M = [\![\psi(\sigma)]\!]$, because it is really *the set* of acceptable new data states that counts. To get a matching input-transition, the same translation should be applied there. But such an input-transition could have a smaller set according to its post-condition as in the example. To get the same input action as the output, we allow all $M \supseteq [\![\psi(\sigma)]\!]$ for *inputs*; such an M can be understood as the guarantee that the new data state will be in M. This way, the IAs do not produce false errors, and the real errors still occur in the translation if the output with $M = [\![\psi(\sigma)]\!]$ cannot be matched.

Another issue is that, if we refine some $p \multimap\!(\varphi, a!, \psi)\!\rightarrowtail p'$, we could weaken the assumption, i.e. the corresponding $a(\sigma, M)!$ would be refined by $a(\sigma, M')!$ with $M \subsetneq M'$. To make this possible, we also include supersets on the *output* side. This does not disturb the correct handling of errors: if there is a matching input for $a(\sigma, M)!$, there is also one for $a(\sigma, M')!$. Observe that taking supersets for inputs actually also allows for correct refinements, because we have the same entailment $\psi_i(\sigma) \Rightarrow \psi(\sigma)$ in Definition 8(i) and (ii).

To reflect on this translation scheme for our example IAMs P, Q, consider their translations $IA(P)$, $IA(Q)$ in Fig. 17, where σ_i stands for $\sigma(x) = i$. Due to pre-condition *tt*, the transition of P is split into two transitions in $IA(P)$, one each for the possible values of x in the given domain. Because $\psi(\sigma_1) = \psi(\sigma_2) = (x' \in \{1, 2\})$, the only choice for M is $\{\sigma_1', \sigma_2'\}$. Similarly, the translation scheme splits Q's transition into two; each of these is again split into two transitions, because $\psi(\sigma_1) = \psi(\sigma_2) = (x' = 1)$ allows for choosing $M = \{\sigma_1'\}$ and also $M = \{\sigma_1', \sigma_2'\}$. This way, the resulting $IA(P)$ and $IA(Q)$ do not produce false errors on action $a(\sigma_1, \{\sigma_1', \sigma_2'\})$ when being composed. In other words, our translation ensures that the weaker post-condition of P is taken care of by allowing the according weakening of Q's post-condition, while maintaining the semantics of Q by also including transitions for Q's original post-condition.

In such a scenario, more generally speaking, $a(\sigma, M)!$ is present for σ satisfying the pre-conditions of both P and Q whenever $[\![\psi_P(\sigma)]\!] \subseteq M$, and $a(\sigma, M')?$ whenever $[\![\psi_Q(\sigma)]\!] \subseteq M'$, where ψ_P and ψ_Q refer to the according post-conditions in P and Q, resp. If no error arises for σ, i.e. if $[\![\psi_Q(\sigma)]\!] \subseteq [\![\psi_P(\sigma)]\!]$, each $a(\sigma, M)!$ is matched by $a(\sigma, M)?$. Otherwise, $a(\sigma, [\![\psi_P(\sigma)]\!])!$ does not have a match and the error arises in the translation as well.

We now formalize our translation and then work towards proving the correctness of the induced embedding of IAM into IA.

Definition 15 (*Embedding IAM into IA*) IAM P is translated to the IA $\mathit{IA}(P) =_{\text{df}} (P, I', O', \rightarrow', p_0)$, where $I' =_{\text{df}} \{a(\sigma, M) \mid a \in I, \sigma \in [\![V]\!] \text{ and } M \subseteq [\![V']\!]\}$, $O' =_{\text{df}} \{a(\sigma, M) \mid a \in O, \sigma \in [\![V]\!] \text{ and } M \subseteq [\![V']\!]\}$, and $\rightarrow'$ is obtained from $\rightarrow$ of P according to the following rules:

- $p \,\text{—}\!(\varphi, a, \psi)\!\!\rightarrowtail p'$ implies $p \xrightarrow{a(\sigma,M)}{}' p'$, for $\sigma \models \varphi$ and $M \supseteq [\![\psi(\sigma)]\!]$;[4]
- $p \,\text{—}\!(\varphi, \tau, \psi)\!\!\rightarrowtail p'$ implies $p \xrightarrow{\tau}{}' p'$.

This embedding satisfies two important properties: it is a homomorphism for parallel composition, as indicated in Figs. 15 and 17, and monotonic wrt. alternating refinement.

Theorem 16 (Homomorphism) *IAMs P, Q are composable (compatible, resp.) iff $\mathit{IA}(P)$ and $\mathit{IA}(Q)$ are. Then, $\mathit{IA}(P \parallel Q)$ and $\mathit{IA}(P) \parallel \mathit{IA}(Q)$ are identical.*

Proof Composability only depends on the signature, and inputs (outputs, resp.) of an IAM R give rise to inputs (outputs, resp.) of $\mathit{IA}(R)$, so the first claim follows. Obviously, $P \otimes Q$ and $\mathit{IA}(P) \,\overline{\otimes}\, \mathit{IA}(Q)$ have the same state sets. The initial states also coincide. We proceed as follows:

1. We show that $P \otimes Q$ has the same *error states* as $\mathit{IA}(P) \,\overline{\otimes}\, \mathit{IA}(Q)$:

   - *$\langle p, q\rangle$ is an error state in $P \otimes Q$ due to Definition 11* ($E!?\varphi$) *or* ($E?!\varphi$). W.l.o.g., P has a transition $p \,\text{—}\!\big(\varphi_p, a!, \psi_p\big)\!\!\rightarrowtail p'$, and some σ with $\sigma \models \varphi_p$ satisfies $\sigma \not\models \varphi_q$ for all transitions $q \,\text{—}\!\big(\varphi_q, a?, \psi_q\big)\!\!\rightarrowtail q'$. The p-transition gives rise to some $p \xrightarrow{a(\sigma,M)!} p'$ in $\mathit{IA}(P)$ but, due to the condition $\sigma \not\models \varphi_q$, state q has no input-transition with any $a(\sigma, X)?$. As a result, $\langle p, q\rangle$ is also an error state in $\mathit{IA}(P) \,\overline{\otimes}\, \mathit{IA}(Q)$.
   - *$\langle p, q\rangle$ is an error state in $P \otimes Q$ due to Definition 11* ($E!?\psi$) *or* ($E?!\psi$). W.l.o.g., P has a transition $p \,\text{—}\!\big(\varphi_p, a!, \psi_p\big)\!\!\rightarrowtail p'$ and, for some transition $q \,\text{—}\!\big(\varphi_q, a?, \psi_q\big)\!\!\rightarrowtail q'$ and data state σ, $\sigma \models \varphi_p \wedge \varphi_q$ and $\psi_q(\sigma) \not\Rightarrow \psi_p(\sigma)$. The latter means $[\![\psi_q(\sigma)]\!] \nsubseteq [\![\psi_p(\sigma)]\!]$. Now, $\mathit{IA}(P)$ contains the transition $p \xrightarrow{a(\sigma,[\![\psi_p(\sigma)]\!])!} p'$. Due to data-determinism, the $a(\sigma, M)?$-transitions of q result from transition $q \,\text{—}\!\big(\varphi_q, a?, \psi_q\big)\!\!\rightarrowtail q'$, i.e. $M \supseteq [\![\psi_q(\sigma)]\!]$ and M cannot be $[\![\psi_p(\sigma)]\!]$. Again, $\langle p, q\rangle$ is also an error in $\mathit{IA}(P) \,\overline{\otimes}\, \mathit{IA}(Q)$.
   - *$\langle p, q\rangle$ is an error state in $\mathit{IA}(P) \,\overline{\otimes}\, \mathit{IA}(Q)$.* W.l.o.g., $\mathit{IA}(P)$ has a transition $p \xrightarrow{a(\sigma,M)!} p'$ that $\mathit{IA}(Q)$ cannot match in state q. Thus, P has a transition $p \,\text{—}\!\big(\varphi_p, a!, \psi_p\big)\!\!\rightarrowtail p'$ with $\sigma \models \varphi_p$ and $[\![\psi_p(\sigma)]\!] \subseteq M$. By the definition of our translation, p can also perform $a(\sigma, [\![\psi_p(\sigma)]\!])$, while q cannot:

[4] Note that infinite branching may occur in case the domain of variables is infinite and, thus, I' and O' are infinite. This may be unusual in interface theories, but the embedding exercise is rather a theoretical one aimed at demonstrating that the IAM setting is not more expressive than the IA setting in principle.

- If $IA(Q)$ does not contain any $a(\sigma, X)?$-transition at q, then there is no $q \multimap(\varphi_q, a?, \psi_q)\rightarrowtail q'$ with $\sigma \models \varphi_q$ in Q. Thus, $\langle p, q\rangle$ is an error in $P \otimes Q$ due to Definition 11 ($E!?\varphi$).
- Otherwise, there exists only one transition $q \multimap(\varphi_q, a?, \psi_q)\rightarrowtail q'$ with $\sigma \models \varphi_q$ in Q, due to data-determinism. Because q does not have the action $a(\sigma, [\![\psi_p(\sigma)]\!])$, we get $[\![\psi_p(\sigma)]\!] \not\supseteq [\![\psi_q(\sigma)]\!]$. Again, $\langle p, q\rangle$ is an error in $P \otimes Q$ due to Definition 11 ($E!?\psi$).

2. We show that $IA(P \otimes Q)$ is isomorphic to $IA(P) \overline{\otimes} IA(Q)$, except for outgoing transitions of error states, and consider synchronized transitions first:
   - If $p \multimap(\varphi_p, a!, \psi_p)\rightarrowtail p'$ and $q \multimap(\varphi_q, a?, \psi_q)\rightarrowtail q'$, $\varphi_p \wedge \varphi_q$ sat and $\psi_q(\sigma) \Rightarrow \psi_p(\sigma)$ holds for all σ with $\sigma \models \varphi_p \wedge \varphi_q$, then the transitions synchronize to $\langle p, q\rangle \multimap(\varphi_p \wedge \varphi_q, \tau, \psi_q)\rightarrowtail \langle p', q'\rangle$ in $P \otimes Q$. We get $\langle p, q\rangle \xrightarrow{\tau} \langle p', q'\rangle$ on the left. Choose σ with $\sigma \models \varphi_p \wedge \varphi_q$ and $M \supseteq [\![\psi_p(\sigma)]\!] \supseteq [\![\psi_q(\sigma)]\!]$. Then, we have $p \xrightarrow{a(\sigma,M)!} p'$ and $q \xrightarrow{a(\sigma,M)?} q'$, resulting in $\langle p, q\rangle \xrightarrow{\tau} \langle p', q'\rangle$ on the right. Note that all transitions $p \xrightarrow{a(\sigma,X)!} p'$ are dealt with in this way, but possibly not all transitions $q \xrightarrow{a(\sigma,X)?} q'$ stemming from the q-transition in Q; these are inputs and do not arise on the right.
   - If $\langle p, q\rangle \xrightarrow{\tau} \langle p', q'\rangle$ arises due to $p \xrightarrow{a(\sigma,M)!} p'$ and $q \xrightarrow{a(\sigma,M)?} q'$ on the right, then there are $p \multimap(\varphi_p, a!, \psi_p)\rightarrowtail p'$ and $q \multimap(\varphi_q, a?, \psi_q)\rightarrowtail q'$ with $\sigma \models \varphi_p \wedge \varphi_q$. In case $\psi_q(\sigma) \not\Rightarrow \psi_p(\sigma)$, state $\langle p, q\rangle$ is an error state in $P \otimes Q$. Otherwise, $\langle p, q\rangle \multimap(\varphi_p \wedge \varphi_q, \tau, \psi_q)\rightarrowtail \langle p', q'\rangle$ in $P \otimes Q$, and we get $\langle p, q\rangle \xrightarrow{\tau} \langle p', q'\rangle$ on the left, for all σ such that $\sigma \models \varphi_p \wedge \varphi_q$.

   Unsynchronized transitions of P are "copied" into $P \otimes Q$ and replaced by a bundle of transitions on the left, and replaced by the same bundle that is copied into the product on the right. The same holds for unsynchronized transitions of Q.
3. Due to Item (1), $P \otimes Q$ and $IA(P) \overline{\otimes} IA(Q)$ have the same error states. Each illegal state has a path of local actions reaching an error state at the end only. Hence, we can conclude from Item (2) that both systems have the same illegal states, and $IA(P) \parallel IA(Q)$ is defined iff $P \parallel Q$ is. Applying Item (2) again, we get that $IA(P \parallel Q)$ is isomorphic to $IA(P) \parallel IA(Q)$.

□

Theorem 17 (Monotonicity) *$P \sqsubseteq_{IAM} Q$ iff $IA(P) \sqsubseteq_{IA} IA(Q)$, for all IAMs P, Q.*

Proof "⇒": Assume $P \sqsubseteq_{IAM} Q$ due to the alternating simulation $\mathcal{R}$. We show that $\mathcal{R}$ is also an alternating simulation for $IA(P)$ and $IA(Q)$. First, $p_0\mathcal{R}q_0$ holds for the IAMs and, thus, also for the IAs. Let $p\mathcal{R}q$ and consider the following cases:

- $q \xrightarrow{a(\overline{\sigma},M)?} q'$ *due to* $q \multimap(\varphi, a?, \psi)\rightarrowtail q'$. Then, there is a family of transitions $p \multimap(\varphi_i, a?, \psi_i)\rightarrowtail p_i$ with $\varphi \Rightarrow \bigvee_i \varphi_i$ such that, for all i and σ with $\sigma \models \varphi \wedge \varphi_i$, we have $\psi_i(\sigma) \Rightarrow \psi(\sigma)$ and $p_i\mathcal{R}q'$. Further, $\overline{\sigma} \models \varphi$ and $[\![\psi(\overline{\sigma})]\!] \subseteq M$ due to our translation. Hence, $\overline{\sigma} \models \bigvee_i \varphi_i$ and there is a unique i with $\overline{\sigma} \models \varphi_i$. Because $[\![\psi_i(\overline{\sigma})]\!] \subseteq [\![\psi(\overline{\sigma})]\!] \subseteq M$, we get $p \xrightarrow{a(\overline{\sigma},M)?} p_i$ in $IA(P)$ and $p_i\mathcal{R}q'$.
- $p \xrightarrow{a(\sigma,M)!} p'$ *due to* $p \multimap(\varphi, a!, \psi)\rightarrowtail p'$, $\sigma \models \varphi$ and $[\![\psi(\sigma)]\!] \subseteq M$. Then, there is a suitable family of $q_i \multimap(\varphi_i, a!, \psi_i)\rightarrowtail q_i'$ with $\varphi \Rightarrow \bigvee_i \varphi_i$, and at least one of

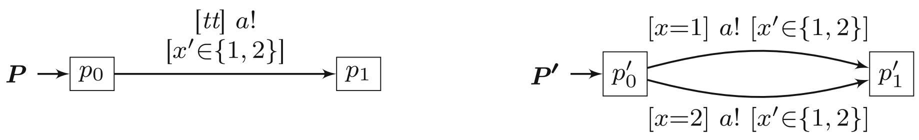


Fig. 18 Example showing $P \neq P'$ but $IA(P) = IA(P')$, for IAM P of Fig. 15

these transitions $q_i \multimap(\varphi_i, a!, \psi_i)\rightarrowtail q'_i$ satisfies $\sigma \models \varphi_i$, $q \overset{\varepsilon}{\Rightarrow} q_i$, $\psi_i(\sigma) \Rightarrow \psi(\sigma)$ and $p'\mathcal{R}q'_i$. Now, $q \overset{\varepsilon}{\Rightarrow} q_i$ also in $IA(Q)$, $q_i \xrightarrow{a(\sigma,M)!} q'_i$ due to $[\![\psi_i(\sigma)]\!] \subseteq [\![\psi(\sigma)]\!] \subseteq M$, and $p'\mathcal{R}q'_i$.

– $p \overset{\tau}{\rightarrow} p'$. This obviously leads to $q \overset{\varepsilon}{\Rightarrow} q'$ in $IA(Q)$, for some q' with $p'\mathcal{R}q'$.

"$\Leftarrow$": Assume $IA(P) \sqsubseteq_{IA} IA(Q)$ due to the alternating simulation $\mathcal{R}$. We claim that $\mathcal{R}$ is also an alternating simulation for P and Q. Note that $p_0\mathcal{R}q_0$ holds for the IAs and, thus, also for the IAMs. Let $p\mathcal{R}q$ and distinguish the following cases:

– $q \multimap(\varphi, a?, \psi)\rightarrowtail q'$. Consider some σ with $\sigma \models \varphi$. So, $q \xrightarrow{a(\sigma,[\![\psi(\sigma)]\!])?} q'$ and there is some $p \xrightarrow{a(\sigma,[\![\psi(\sigma)]\!])?} p'_\sigma$ with $p'_\sigma\mathcal{R}q'$. The latter is due to a unique transition $p \multimap(\varphi_\sigma, a?, \psi_\sigma)\rightarrowtail p'_\sigma$ with $\sigma \models \varphi_\sigma$, which also satisfies $[\![\psi_\sigma(\sigma)]\!] \subseteq [\![\psi(\sigma)]\!]$.[5] The family of all these transitions satisfies $\varphi \Rightarrow \bigvee_\sigma \varphi_\sigma$ and, for all σ with $\sigma \models \varphi$, we have $\sigma \models \varphi \wedge \varphi_\sigma$ and $\psi_\sigma(\sigma) \Rightarrow \psi(\sigma)$.

– $p \multimap(\varphi, a!, \psi)\rightarrowtail p'$. Consider some σ with $\sigma \models \varphi$. So, $p \xrightarrow{a(\sigma,[\![\psi(\sigma)]\!])!} p'$ and there is some q_σ with $q \overset{\varepsilon}{\Rightarrow} q_\sigma$ and $q_\sigma \xrightarrow{a(\sigma,[\![\psi(\sigma)]\!])!} q'_\sigma$ such that $p'\mathcal{R}q'_\sigma$. The latter is due to some $q_\sigma \multimap(\varphi_\sigma, a!, \psi_\sigma)\rightarrowtail q'_\sigma$ with $\sigma \models \varphi_\sigma$ and $[\![\psi_\sigma(\sigma)]\!] \subseteq [\![\psi(\sigma)]\!]$. The family of all these transitions in Q satisfies $\varphi \Rightarrow \bigvee_\sigma \varphi_\sigma$ and, for each σ with $\sigma \models \varphi$, the resp. transition satisfies $\sigma \models \varphi_\sigma$, $q \overset{\varepsilon}{\Rightarrow} q_\sigma$ and $\psi_\sigma(\sigma) \Rightarrow \psi(\sigma)$.

– $p \multimap(\varphi, \tau, \psi)\rightarrowtail p'$. We get the transition $p \overset{\tau}{\rightarrow} p'$. Due to $\mathcal{R}$, we obtain some q' with $q \overset{\varepsilon}{\Rightarrow} q'$ in $IA(Q)$ and $p'\mathcal{R}q'$. Hence, $q \overset{\varepsilon}{\Rightarrow} q'$ also holds in Q.

□

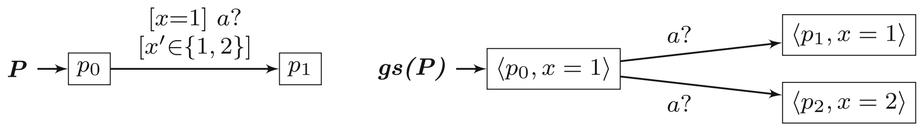
Using the last two theorems and Proposition 4, we can now prove the associativity of parallel composition as stated in Proposition 12:

Proof $IA((P \parallel Q) \parallel R) = (IA(P) \parallel IA(Q)) \parallel IA(R)$ is IAM-equivalent to $IA(P) \parallel (IA(Q) \parallel IA(R)) = IA(P \parallel (Q \parallel R))$ □

The embedding of IAM into IA is not injective due to at least two possible reasons, although these are somewhat pathological. The first reason is that pre- and post-conditions of τ-transitions in an IAM are omitted in the translation. Hence, $IA(P)$ cannot distinguish between τ-transitions with the same source and target but different pre- or post-conditions in P. The second reason is that two different IAMs can have the same translation because of visible parallel transitions. IAM P' in Fig. 18 is not equal to P, but $IA(P')$ is the same automaton as $IA(P)$ shown in Fig. 17. Observe that, in general, two IAMs that are mapped to the same IA are at least IAM-equivalent due to Theorem 17.

The compositionality of parallel composition for IAM now follows from the corresponding property for IA [13] (see Theorem 6) and the above theorems:

[5] The notation φ_σ used in this proof does not stand for a univalent predicate as introduced in the next section.



$[x{=}1]\ a?$
$[x'{\in}\{1,2\}]$
P
p_0
p_1
$gs(P)$
$\langle p_0, x=1\rangle$
$a?$
$\langle p_1, x=1\rangle$
$a?$
$\langle p_2, x=2\rangle$

Fig. 19 Example showing that $gs(P)$ can be a quasi-IA

Corollary 18 (Compositionality) *Let P, Q, R be IAMs and $P \sqsubseteq_{IAM} Q$. Further, assume that Q and R are composable. We have: (i) P and R are composable; (ii) $P \parallel R$ is defined if $Q \parallel R$ is, and then $P \parallel R \sqsubseteq_{IAM} Q \parallel R$.*

Proof By Theorems 16 and 17, the assumptions hold for $IA(P)$, $IA(Q)$ and $IA(R)$, and also the assumption in (ii) carries over. For these, the result is known (see Theorem 6). Hence, also P and R are composable (compatible, resp.) by Theorem 16. Further, $IA(P) \parallel IA(R) \sqsubseteq_{IA} IA(Q) \parallel IA(R)$ implies $IA(P \parallel R) \sqsubseteq_{IA} IA(Q \parallel R)$ by Theorem 16. Now, $P \parallel R \sqsubseteq_{IAM} Q \parallel R$ follows by Theorem 17. □

7 Ground semantics: data-closed IAM

Communication via shared memory works efficiently only for a limited number of components that are locally close to each other. In practical applications, one has a group of components that communicate among each other via shared memory and with their environment via, e.g. message passing [18]. The group is modelled as a parallel composition of IAMs with the understanding that the shared memory is local to this group. Technically, the composition is itself an IAM that is data-closed as defined below. Informally, we speak of a *cluster* to keep in mind that it represents a group closely connected by common data. To sketch the use of our framework for designing shared-memory systems, we assume that the communication with the system environment is just by synchronizing on common actions, i.e. the environment is an IA and sees the cluster as an IA that is originally specified by some IA *Spec*.

As the first step for implementing *Spec*, it has to be translated to an IAM *Spec'*, and we demonstrate how this can be done towards the end of this section. *Spec'* is developed with the IAM design-process into a final cluster P. Now, the system environment needs an IA as a behavioural description of P. As the main result, we show how to represent P by an IA that refines *Spec*; under some restrictions, the construction even gives a finite IA if P is finite. Due to the symbolic handling of possibly infinite data, this is not obvious. After Proposition 30, when having obtained all formal results, we describe the design process from *Spec* to the representation of P in greater detail. Until then, we concentrate on the representation of P.

In the previous sections, we have assumed that a control state p of P can be entered while being in some data state σ, and then left because of a data state $\overline{\sigma}$ different from σ. In a *data-closed* system, such a change cannot happen since the shared memory is inaccessible from the outside. To describe the behaviour of such a system, we present a ground semantics where p can only be left via a transition whose pre-condition is satisfied by data state σ. In particular, pre- and post-conditions are now also relevant for τ-transitions.

More formally, states of the ground semantics are pairs $\langle p, \sigma\rangle$, where p is a control state and σ a data state. A transition of p with pre-condition φ is only kept for $\langle p, \sigma\rangle$ if $\sigma \models \varphi$. Note that, in case of infinite data, there are usually infinitely many states, even if P is finite.

The ground semantics of a data-closed IAM P, denoted by $gs(P)$, describes the behaviour of P as seen from its environment, given some *initial data state* σ_0. Such an initial state would not make much sense for an IAM since it could be changed immediately by the environment. Automaton $gs(P)$ is essentially an IA without pre- and post-conditions, but it might violate input-determinism if several data states satisfy the post-condition of an input-transition. Figure 19 shows an example with $V = \{x\}, \mathbb{D} = \{1, 2\}$ and an initial data state where $x = 1$. In this section, we call such IA-like systems *quasi-IAs*. Similarly, a system that is an IAM except for data-determinism is called a *quasi-IAM*.

Definition 19 (*Ground Semantics*) A *data-closed IAM P* is a tuple $(P, I, O, \rightarrow_P, p_0, \sigma_0)$, where $(P, I, O, \rightarrow_P, p_0)$ is an IAM and σ_0 is the *initial data state*. The *ground semantics* of such a P is the quasi-IA $gs(P) =_{\mathrm{df}} (P \times [\![V]\!], I, O, \rightarrow_{gs(P)}, \langle p_0, \sigma_0\rangle)$ such that $\langle p, \sigma\rangle \xrightarrow{\alpha}_{gs(P)} \langle p', \overline{\sigma}\rangle$ if there is a transition $p \,\text{–}\!(\varphi, \alpha, \psi)\!\!\rightarrow_P p'$ with $\sigma \models \varphi$ and $\sigma, \overline{\sigma}' \models \psi$.

7.1 Reduction of a data-closed IAM

To fit the development of a shared-memory cluster into the IA-approach, we first define a quasi-IA that properly reflects the behaviour of the final data-closed IAM P wrt. the ground semantics, but is finite whenever P is (cf. Theorem 25). In turn, this quasi-IA is based on a quasi-IAM to which we refer as the *reduction* of P; it has *the same* ground semantics as P. Recall that state $\langle p, \sigma\rangle$ of the ground semantics inherits those transitions of p whose pre-conditions are satisfied by σ. If another data state $\overline{\sigma}$ satisfies the same pre-conditions as σ, then $\langle p, \overline{\sigma}\rangle$ inherits the same transitions as $\langle p, \sigma\rangle$. The main idea is to merge such states. Each merged state is essentially characterized by a conjunction of pre-conditions, written φ_J in the following definition.

Definition 20 (*Reduction*) Given a data-closed IAM $(P, I, O, \rightarrow_P, p_0, \sigma_0)$, assume that each $p \in P$ has outgoing transitions indexed over some K_p with pre-conditions φ_j, $j \in K_p$. For each $J \subseteq K_p$, let φ_J be $\bigwedge_{j\in J} \varphi_j \wedge \bigwedge_{j\notin J} \neg\varphi_j$. Set J is called a *p-index set* if φ_J is satisfiable.

The *reduction* of P is the data-closed quasi-IAM $(red(P), I, O, \rightarrow_{red(P)}, \langle p_0, \varphi_0\rangle, \sigma_0)$ with the following properties:

– The set $red(P) \subseteq P \times Pred(V)$ of states consists of all pairs $\langle p, \varphi_J\rangle$ with J being a p-index set.
– If $p \,\text{–}\!\big(\varphi_j, \alpha, \psi\big)\!\!\rightarrow_P p'$ and there are p- and p'-index sets J and J', resp., with $j \in J$ and $\varphi_J \wedge \psi \wedge \varphi'_{J'}$ *sat*, then $\langle p, \varphi_J\rangle \,\text{–}\!\big(\varphi_J, \alpha, \varphi_J \wedge \psi \wedge \varphi'_{J'}\big)\!\!\rightarrow_{red(P)} \langle p', \varphi_{J'}\rangle$.
– Predicate φ_0 in the initial state $\langle p_0, \varphi_0\rangle$ is φ_{J_0}, where J_0 is the p_0-index set $\{j \mid \sigma_0 \models \varphi_j\}$.

State $\langle p, \varphi_J\rangle$ of $red(P)$ represents all $\langle p, \sigma\rangle$ in $gs(P)$ where $\sigma \models \varphi_J$. Given p and σ, there is a unique p-index set J with $\sigma \models \varphi_J$; for this J, data state σ satisfies φ_j iff $j \in J$. A state $\langle p, \varphi_\emptyset\rangle$ corresponds to the data states that do not satisfy any pre-condition attached to p; so all such states of $red(P)$ have no outgoing transitions and could be identified with a new state *dead*. Observe that there are finitely many formulas φ_J for a finite P; thus:

Lemma 21 (Finiteness) *For any finite, data-closed IAM P, the quasi-IAM $red(P)$ is finite.*

Given the transition from p in Definition 20, consider some σ with $\sigma \models \varphi_J$. Because $j \in J$, we have $\sigma \models \varphi_j$, $\psi(\sigma)$ is satisfiable and, for each feasible new data state $\overline{\sigma}$ at p', one identifies the proper p'-index set J'. Then, $\overline{\sigma}'$ (which is applied to V') satisfies $\psi(\sigma)$

and $\varphi'_{J'}$ (obtained from the resp. pre-conditions, replacing each x by x' first). Together, $\sigma, \overline{\sigma}' \models \varphi_J \wedge \psi \wedge \varphi'_{J'}$. This explains the details of the transition of $red(P)$ in the above definition. It also shows that each transition from p gives rise to a transition from $\langle p, \varphi_J \rangle$ for each suitable J; we refer to this in the proof of Theorem 29. Examples of reductions can be found in Figs. 20 and 21.

Now, we show that $gs(red(P))$ is essentially the same as $gs(P)$. This means that $red(P)$ and P have the same behaviour as seen from the environment. We call a state $\langle\langle p, \varphi_J \rangle, \sigma\rangle$ of $gs(red(P))$ *data consistent* if $\sigma \models \varphi_J$. In other words, φ_J is an invariant for data consistent states $\langle\langle p, \varphi_J \rangle, \sigma\rangle$. All reachable states in $gs(red(P))$ are data consistent: the initial state is data consistent, because φ_0 is defined such that $\sigma_0 \models \varphi_0$. Furthermore, if σ and $\overline{\sigma}'$ satisfy the post-condition of a transition leading to $\langle p', \varphi_{J'} \rangle$ in $red(P)$, then $\overline{\sigma} \models \varphi_{J'}$, i.e. the resp. state $\langle\langle p', \varphi_{J'} \rangle, \overline{\sigma}\rangle$ of $gs(red(P))$ is data consistent.

Similarly, we remark that only data consistent states of $gs(red(P))$ have outgoing transitions. All transitions in $gs(red(P))$ are based on transitions of $red(P)$ and the data states have to fulfill the pre-conditions of the outgoing transitions; these pre-conditions coincide with the φ_J of the state of $red(P)$.

Theorem 22 (Data consistency) *Let P be a data-closed IAM. Each reachable state of $gs(red(P))$ is data consistent. The quasi-IA induced by data consistent states is isomorphic to $gs(P)$ due to the isomorphism ι: $\langle\langle p, \varphi_J \rangle, \sigma\rangle \mapsto \langle p, \sigma\rangle$; note that $\sigma \models \varphi_J$. The reachable parts of $gs(red(P))$ and $gs(P)$ are isomorphic due to the resp. restriction of ι.*

Proof After the above observations, it only remains to show that ι is an isomorphism. Because the initial state of $gs(red(P))$ is data consistent and mapped to $\langle p_0, \sigma_0\rangle$, this implies the last claim. Let $\iota(\langle\langle p, \varphi_J \rangle, \sigma\rangle) = \langle p, \sigma\rangle$:

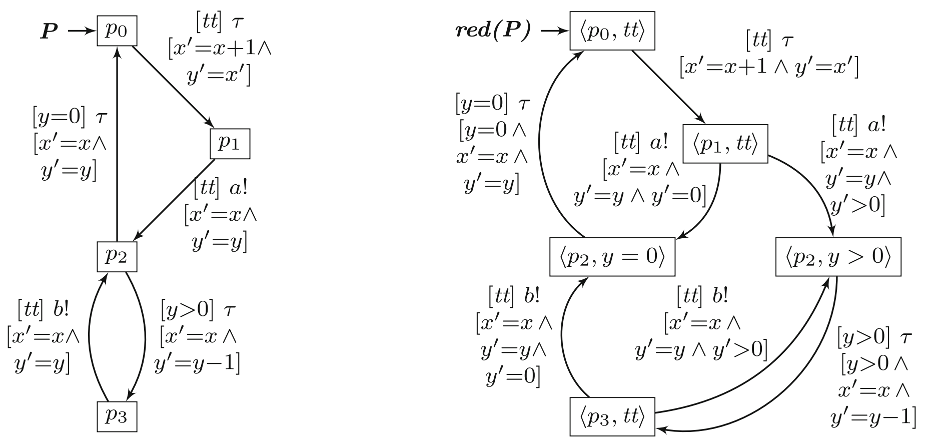
- Let $\langle\langle p, \varphi_J \rangle, \sigma\rangle \xrightarrow{\alpha}_{gs(red(P))} \langle\langle p', \varphi_{J'} \rangle, \overline{\sigma}\rangle$ be a transition from $\langle\langle p, \varphi_J \rangle, \sigma\rangle$. This exists due to transition $\langle p, \varphi_J \rangle \xrightarrow{(\varphi_J, \alpha, \varphi_J \wedge \psi \wedge \varphi'_{J'})}_{red(P)} \langle p', \varphi_{J'} \rangle$ with $\sigma, \overline{\sigma}' \models \varphi_J \wedge \psi \wedge \varphi'_{J'}$. In turn, this has arisen from a transition $p \xrightarrow{(\varphi_j, \alpha, \psi)}_P p'$ with $j \in J$ in P. Hence, $\sigma \models \varphi_j$ and $\sigma, \overline{\sigma}' \models \psi$, implying that there is the transition $\langle p, \sigma\rangle \xrightarrow{\alpha}_{gs(P)} \langle p', \overline{\sigma}\rangle$ in $gs(P)$.
- Let $\langle p, \sigma\rangle \xrightarrow{\alpha}_{gs(P)} \langle p', \overline{\sigma}\rangle$ be a transition from $\langle p, \sigma\rangle$, which has resulted from some transition $p \xrightarrow{(\varphi_i, \alpha, \psi)}_P p'$ with $\sigma \models \varphi_i$ and $\sigma, \overline{\sigma}' \models \psi$. This is the ith outgoing transition of p, and since σ also fulfills φ_J, we must have $i \in J$ as observed above. Also as observed, there is a unique p'-index set J' with $\overline{\sigma} \models \varphi_{J'}$. Thus, σ and $\overline{\sigma}'$ show $\varphi_J \wedge \psi \wedge \varphi'_{J'}$ sat, and $red(P)$ has the transition $\langle p, \varphi_J \rangle \xrightarrow{(\varphi_J, \alpha, \varphi_J \wedge \psi \wedge \varphi'_{J'})}_{red(P)} \langle p', \varphi_{J'} \rangle$. Hence, $\langle\langle p, \varphi_J \rangle, \sigma\rangle \xrightarrow{\alpha}_{gs(red(P))} \langle\langle p', \varphi_{J'} \rangle, \overline{\sigma}\rangle$ in $gs(red(P))$ and $\iota(\langle\langle p', \varphi_{J'} \rangle, \overline{\sigma}\rangle) = \langle p', \overline{\sigma}\rangle$.

□

We now define a simple translation of an IAM Q to a quasi-IA $ia(Q)$. It turns out that $ia(red(P))$ has essentially the same behaviour as $gs(P)$, if P's post-conditions do not contain any unprimed variables.

Definition 23 (*Quasi-IA Function*) Quasi-IA $ia(Q)$ results from IAM Q by deleting all pre- and post-conditions.

The quasi-IA $ia(Q)$ is only IA-like as we do not necessarily get an input-deterministic system. Input-determinism is implied by data-determinism in combination with another property (see below). Obviously, $ia(Q)$ is finite if Q is. The notion of similar behaviour, as used here, is (strong) bisimilarity. On IAs, this is much stronger than mutual IA-refinement.



P
p_0
p_1
p_2
p_3
$[tt]\ \tau\ [x'=x+1\wedge y'=x']$
$[y=0]\ \tau\ [x'=x\wedge y'=y]$
$[tt]\ a!\ [x'=x\wedge y'=y]$
$[tt]\ b!\ [x'=x\wedge y'=y]$
$[y>0]\ \tau\ [x'=x\wedge y'=y-1]$
$red(P)$
$\langle p_0, tt\rangle$
$\langle p_1, tt\rangle$
$\langle p_2, y=0\rangle$
$\langle p_2, y>0\rangle$
$\langle p_3, tt\rangle$
$[tt]\ \tau\ [x'=x+1\wedge y'=x']$
$[y=0]\ \tau\ [y=0\wedge x'=x\wedge y'=y]$
$[tt]\ a!\ [x'=x\wedge y'=y\wedge y'=0]$
$[tt]\ a!\ [x'=x\wedge y'=y\wedge y'>0]$
$[tt]\ b!\ [x'=x\wedge y'=y\wedge y'=0]$
$[tt]\ b!\ [x'=x\wedge y'=y\wedge y'>0]$
$[y>0]\ \tau\ [y>0\wedge x'=x\wedge y'=y-1]$

Fig. 20 Example showing that $ia(red(P))$ and $gs(P)$ are not bisimilar in general, where $\sigma_0(x) = \sigma_0(y) = 0$

Definition 24 (*Bisimulation*) A *bisimulation* for quasi-IAs P, Q is a relation $\mathcal{R} \subseteq P \times Q$ if the following conditions hold for all $p\mathcal{R}q$ and α:

(i) $p \xrightarrow{\alpha} p'$ implies $\exists q'.\ q \xrightarrow{\alpha} q'$ and $p'\mathcal{R}q'$;
(ii) $q \xrightarrow{\alpha} q'$ implies $\exists p'.\ p \xrightarrow{\alpha} p'$ and $p'\mathcal{R}q'$.

States p and q are called *bisimilar* if $p\mathcal{R}q$ for a bisimulation $\mathcal{R}$. Further, P and Q are bisimilar if the initial states p_0 and q_0 are bisimilar.

If P has no unprimed variables in its post-conditions, one obtains directly from Definitions 19 and 20 that data consistent states $\langle\langle p, \varphi_J\rangle, \sigma\rangle$ and $\langle\langle p, \varphi_J\rangle, \overline{\sigma}\rangle$ in $gs(red(P))$ have the same transitions, i.e. actions and target states coincide. Relating all such pairs results in a bisimulation $\mathcal{R}$ on $gs(red(P))$ that is an equivalence. It is well known that merging such equivalence classes gives a bisimilar system [20]. If we denote each class by the resp. $\langle p, \varphi_J\rangle$, this system *is* $ia(red(P))$. With Theorem 22 and the above observation that $ia(Q)$ is finite if Q is, we have:

Theorem 25 *Let P be a data-closed IAM. If P is finite, then $ia(red(P))$ is finite, too. If P has no unprimed variables in its post-conditions, then $ia(red(P))$ and $gs(red(P))$, as well as $ia(red(P))$ and $gs(P)$, are bisimilar. They are IA-equivalent, if they are IAs.*

Remark 26 The example in Fig. 20 shows that the restriction wrt. unprimed variables cannot be dropped for this result; here, we do not show conjuncts *tt* in the post-conditions of *red*(P). The ground semantics of data-closed IAM P, with σ_0 mapping x and y to 0, has the sequence $ab^1ab^2ab^3\ldots$ of outputs as visible behaviour. On the resp. unique run, τ-transitions and visible transitions alternate. The states connected by τ are bisimilar; each pair is characterized by the number of bs before the next a and the number of bs after the next but before the next but one a. Thus, no other states are bisimilar, and no finite quasi-IA is bisimilar to $gs(P)$. Although the ground semantics of the reduction *red*(P) is infinite, *red*(P) is finite, and so is $ia(red(P))$. Hence, $ia(red(P))$ and $gs(P)$ cannot be bisimilar.

Even worse, when we consider only quasi-IAs where outputs a and b are the only visible actions, then the quasi-IAs are actually IAs. Taking all states as final states, we can regard them as 'possibly infinite finite automata,' which are language equivalent if they are equivalent wrt. $\sqsubseteq_{IA}$. The language of $gs(P)$ consists of all finite prefixes of $ab^1ab^2ab^3\ldots$; this language is not regular, so no finite IA is equivalent to the IA $gs(P)$ wrt. $\sqsubseteq_{IA}$.

The restriction regarding unprimed variables is clearly uncomfortable; we plan to study how to weaken it in future work. □

7.2 Fitting the reduction to the IA-approach

As indicated above, we want to use a data-closed IAM P in a design process developing an IA. For this, we would like *red*(P) to be an IAM and *ia*(P) and *ia*$($*red*$(P))$ to be IAs. This only holds under some restrictions. We first characterize those *red*(P) that violate data-determinism.

Definition 27 (Transition ambiguity) Input-transition $p \multimap(\varphi, a?, \psi)\mapsto p'$ of data-closed IAM P is *ambiguous* if, for some p-index set J containing the index of this transition, $\varphi_J \wedge \psi \wedge \varphi'_{J'}$ is satisfiable for more than one p'-index set J'.

A predicate that is satisfied exactly by one data state σ is called *univalent* and denoted by φ_σ; w.l.o.g. we assume that a univalent post-condition is written without unprimed variables.

Concerning the input-transitions of some state p in a data-closed IAM P, we observe the following: if two transitions from p numbered i and j concern the same input $a?$, the pre-conditions φ_i and φ_j contradict each other due to data-determinism. Thus, a p-index set J can contain at most one index of an $a?$-transition for a fixed $a?$, i.e. there is at most one $p \multimap(\varphi, a?, \psi)\mapsto_P p'$ that gives rise to $a?$-transitions from $\langle p, \varphi_J\rangle$. Still, *red*$(P)$ might only be a quasi-IAM: if a were an input in the example shown in Fig. 20, *red*(P) would otherwise look the same and would violate data-determinism in state $\langle p_1, tt\rangle$.

Because every p-index set can contain at most one index of an $a?$-transition for a fixed $a?$, a violation of data-determinism at some $\langle p, \varphi_J\rangle$ can only arise from a single input-transition at p, which is then ambiguous. Conversely, if p has an outgoing ambiguous $a?$-transition as in the definition, *red*(P) has more than one outgoing $a?$-transition from $\langle p, \varphi_J\rangle$ by the definition of reduction. Hence, *red*(P) is not data-deterministic, because every outgoing transition of $\langle p, \varphi_J\rangle$ has the pre-condition φ_J. The latter also shows that data-determinism at each $\langle p, \varphi_J\rangle$ implies input-determinism and that *ia*$($*red*$(P))$ is an IA.

We summarize, noting also that a transition with a univalent post-condition is unambiguous, because the $\varphi_{J'}$ at some state p' contradict each other:

Proposition 28 *Let P be a data-closed IAM.*

(i) *P has no ambiguous input-transitions iff red(P) is an IAM.*
(ii) *If red(P) is an IAM, then ia$($red$(P))$ is an IA.*
(iii) *If all input post-conditions in P are univalent, then P has no ambiguous input-transitions.*

Obviously, *ia*(P) is input-deterministic, i.e. an IA, if every state of P has at most one outgoing $a?$-transition for each input a. Due to data-determinism, this is guaranteed if all pre-conditions of input-transitions are equivalent to true. The latter condition also allows us to relate a data-closed IAM P and *red*(P) on the IA-level:

Theorem 29 (Reduction and IA-refinement) *Let P be a data-closed IAM without ambiguous input-transitions, where all pre-conditions of input-transitions are equivalent to true. Then, ia$($red$(P))$ and ia(P) are IAs and, moreover, ia$($red$(P)) \sqsubseteq_{IA}$ ia(P).*

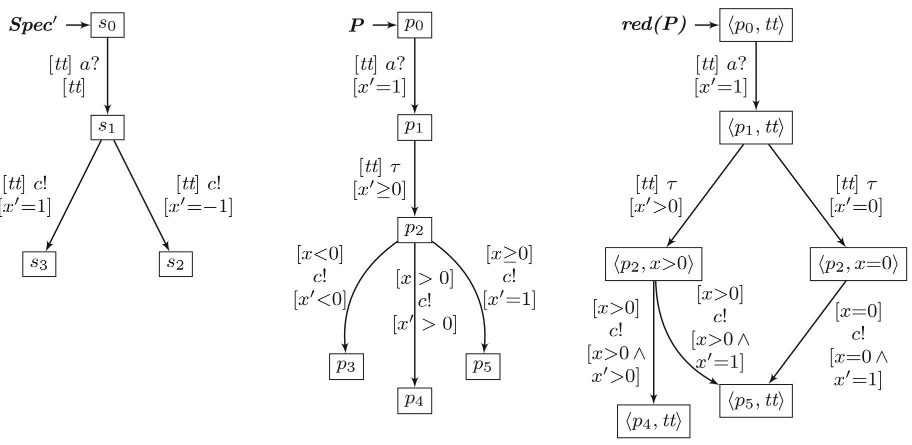


Fig. 21 Example illustrating IAM's application in the IA-context

Proof As noted above and by Proposition 28(i) and (ii), $ia(red(P))$ and $ia(P)$ are IAs. We show that $\mathcal{R} =_{\mathrm{df}} \{(\langle p, \varphi_J\rangle, p) \mid p \in P\}$ is an alternating simulation for $ia(red(P))$ and $ia(P)$. Clearly, $\langle p_0, \varphi_0\rangle \mathcal{R} p_0$ holds for the initial states. Let $\langle p, \varphi_J\rangle \mathcal{R} p$ and consider the following cases:

- $p \xrightarrow{a?}_{ia(P)} p'$. For some φ and ψ, the transition $p \,{-\!\!(}\varphi, a?, \psi{)\!\!\rightarrow}_P\, p'$ is a transition of P, and φ is equivalent to *tt* by assumption, implying that the index of the transition is in each p-index set. As argued after Definition 20, there is some p'-index set J' such that $\langle p, \varphi_J\rangle \,{-\!\!\big(}\varphi_J, a?, \varphi_J \wedge \psi \wedge \varphi'_{J'}{\big)\!\!\rightarrow}_{red(P)}\, \langle p', \varphi_{J'}\rangle$. (Actually, J' is unique by unambiguity.) This implies $\langle p, \varphi_J\rangle \xrightarrow{a?}_{ia(red(P))} \langle p', \varphi_{J'}\rangle$ with $\langle p', \varphi_{J'}\rangle \mathcal{R} p'$.
- $\langle p, \varphi_J\rangle \xrightarrow{\alpha!}_{ia(red(P))} \langle p', \varphi_{J'}\rangle$. The IAM $red(P)$ has a corresponding transition $\langle p, \varphi_J\rangle \,{-\!\!\big(}\varphi_J, \alpha!, \varphi_J \wedge \psi \wedge \varphi'_{J'}{\big)\!\!\rightarrow}_{red(P)}\, \langle p', \varphi_{J'}\rangle$ for some ψ. Hence, P has some transition $p \,{-\!\!\big(}\varphi_j, \alpha!, \psi{\big)\!\!\rightarrow}_P\, p'$ with $j \in J$, implying $p \xrightarrow{\alpha!}_{ia(P)} p'$ with $\langle p', \varphi_{J'}\rangle \mathcal{R} p'$.

□

The following, final result shows how IAM-refinement fits together with IA-refinement:

Proposition 30 (IAM-refinement fits IA-refinement) *Let $P \sqsubseteq_{IAM} Q$ due to an alternating simulation $\mathcal{R}$. Then, $\mathcal{R}$ also satisfies the conditions of an alternating simulation for $ia(P)$ and $ia(Q)$.*

Proof Let $p\mathcal{R}q$. The case of τ-transitions is obvious, because we ignore their conditions in the IAMs anyway. An input-transition of q is matched by a family of input-transitions from p. Each of these results in a transition in $ia(P)$ that matches the resp. transition in $ia(Q)$. The case of outputs is similar. □

We now describe in detail how one can implement a given IA specification *Spec* as a shared memory cluster; explanations and the correctness proof are given afterwards. The inputs and outputs of *Spec* are the external actions for the IAM-design. In the first step, we translate *Spec* into an IAM *Spec′* by decorating each transition with *tt* as pre- and post-condition, except for the post-conditions of output-transitions. As first design choices, we select an initial data state σ_0 and, for each output-transition, a single expected σ, i.e. the post-condition is the univalent predicate φ'_σ. A possible resulting IAM *Spec*$'$ is given in Fig. 21.

To obtain a working implementation, one typically refines *Spec*$'$ by a parallel composition. While developing this cluster in the IAM-framework, the overall system keeps the external actions, and we *require* that the pre-condition of a transition with an external input is always *tt*. Introducing new components introduces new actions; there are no restrictions for the resp. transitions, and they are hidden after synchronization. Refining the new components, which are not data-closed, demands IAM-refinement in order to have a precongruence.

We require the final cluster P to not have any unprimed variables in its post-conditions. Furthermore, it must not have any ambiguous transitions, e.g. due to all post-conditions of external input-transitions being univalent (cf. Definition 27). Then, $ia(red(P))$ IA-refines *Spec*; as a faithful behavioural description of the data-closed IAM P, $ia(red(P))$ can be given to the user of the shared memory cluster as an operating guideline. Preferably, P is finite so that $ia(red(P))$ is finite as well, even if the common ground semantics of P and $red(P)$ were infinite.

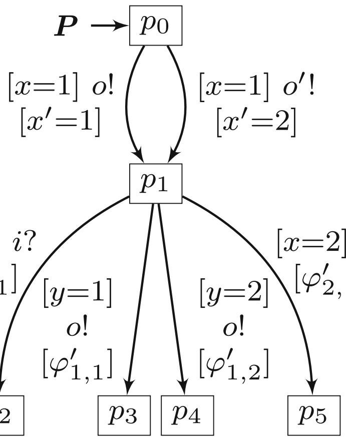
This concludes the description of the design process. In order to demonstrate the technical steps, we give a small example and, as we aim to keep things simple, we do not show any components here. Instead, Fig. 21 depicts a realization P of *Spec*$'$, which has to be understood as the parallel composition of the final cluster. *Spec*$'$, P and $red(P)$ have the external inputs and outputs as signature. Most interestingly, state p_2 in P has transitions with overlapping pre-conditions. This leads to three satisfiable conditions φ_J, which are equivalent to $x < 0$, $x > 0$ and $x = 0$. As $x < 0$ is always invalid after the τ-transition in P, state p_2 results in two reachable states in $red(P)$; the post-conditions of the two τ-transitions are suitably adapted. The left-hand state inherits the two c-transitions where $x > 0$ is allowed, and the right-hand state inherits only the c-transition where $x = 0$ is allowed; the pre-conditions are adapted.

We close this section by motivating the initial choices for *Spec*$'$ and proving the *correctness* of our method. Predicate *tt* is chosen as pre-condition for output-transitions, because this is the most general choice. These pre-conditions can be made stricter during refinement. Dually, the post-conditions can be made broader, so we start from a minimal post-condition. The conditions of τ-transitions do only matter for the final cluster. Post-conditions *tt* for input-transitions are again most general. The pre-conditions of input-transitions being *tt* is only necessary for the final P, where we have to apply Theorem 29, but it seems conceptually easiest to require this throughout. This condition guarantees that the environment, which does not have access to the shared memory, can really rely on the operation calls being accepted as promised in the operating guideline. Obviously, $ia(Spec')$ is *Spec*.

As an aside, we note that *Spec*$'$ is isomorphic to $red(Spec')$; simply replace each p by $\langle p, tt\rangle$. Furthermore, *Spec*$'$ has no post-conditions with unprimed variables. Hence, the behaviour of the data-closed IAM *Spec*$'$ properly reflects $ia(red(Spec'))$ by Theorem 25, and the latter is isomorphic to *Spec*. Therefore, the choice of *Spec*$'$ is intuitively well motivated.

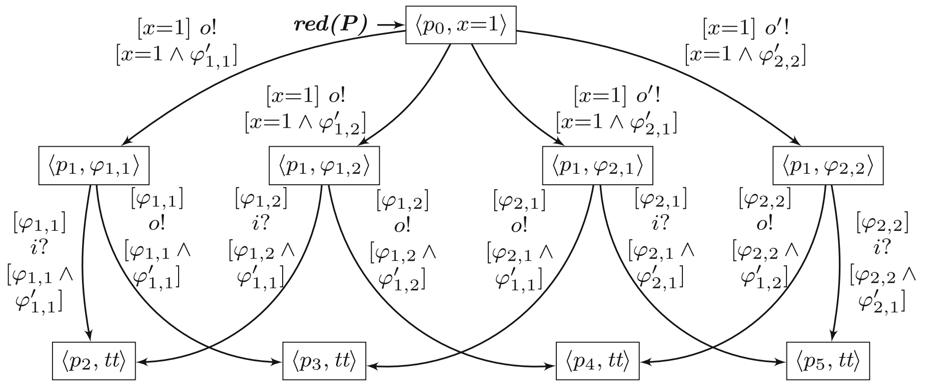
The final cluster P is obtained from *Spec*$'$ by IAM-refinement. By Theorem 29, $ia(P)$ and $ia(red(P))$ are IAs and $ia(red(P)) \sqsubseteq_{IA} ia(P)$. From $P \sqsubseteq_{IAM} Spec'$ and Proposition 30, we derive $ia(P) \sqsubseteq_{IA} ia(Spec')$, where the latter is *Spec*. Together, we have $ia(red(P)) \sqsubseteq_{IA} Spec$. So, $ia(red(P))$ is an IA-refinement of *Spec*, and it faithfully shows the behaviour of P by Theorem 25. Finally, $ia(red(P))$ is finite by Lemma 21, if P is.

Remark 31 One might think that these considerations could be simplified. The idea would be to conclude $red(P) \sqsubseteq_{IAM} Spec'$ from $red(P) \sqsubseteq_{IAM} P$. The IAM P in Fig. 22 and $red(P)$ in Fig. 23 show that the latter can fail in various ways. In this example, we have $\mathbb{D} = \{1, 2\}$,



P
p_0
$[x{=}1]\ o!$ $[x'{=}1]$
$[x{=}1]\ o'!$ $[x'{=}2]$
p_1
$[x{=}1]\ i?$ $[\varphi'_{1,1}]$
$[y{=}1]$ $o!$ $[\varphi'_{1,1}]$
$[y{=}2]$ $o!$ $[\varphi'_{1,2}]$
$[x{=}2]\ i?$ $[\varphi'_{2,1}]$
p_2
p_3
p_4
p_5

Fig. 22 IAM P with $red(P) \not\sqsubseteq_{IAM} P$



$red(P)$
$\langle p_0, x{=}1\rangle$
$[x{=}1]\ o!$ $[x{=}1 \wedge \varphi'_{1,1}]$
$[x{=}1]\ o!$ $[x{=}1 \wedge \varphi'_{1,2}]$
$[x{=}1]\ o'!$ $[x{=}1 \wedge \varphi'_{2,1}]$
$[x{=}1]\ o'!$ $[x{=}1 \wedge \varphi'_{2,2}]$
$\langle p_1, \varphi_{1,1}\rangle$
$\langle p_1, \varphi_{1,2}\rangle$
$\langle p_1, \varphi_{2,1}\rangle$
$\langle p_1, \varphi_{2,2}\rangle$
$[\varphi_{1,1}]$ $i?$ $[\varphi_{1,1} \wedge \varphi'_{1,1}]$
$[\varphi_{1,1}]$ $o!$ $[\varphi_{1,1} \wedge \varphi'_{1,1}]$
$[\varphi_{1,2}]$ $i?$ $[\varphi_{1,2} \wedge \varphi'_{1,1}]$
$[\varphi_{1,2}]$ $o!$ $[\varphi_{1,2} \wedge \varphi'_{1,2}]$
$[\varphi_{2,1}]$ $o!$ $[\varphi_{2,1} \wedge \varphi'_{1,1}]$
$[\varphi_{2,1}]$ $i?$ $[\varphi_{2,1} \wedge \varphi'_{2,1}]$
$[\varphi_{2,2}]$ $o!$ $[\varphi_{2,2} \wedge \varphi'_{1,2}]$
$[\varphi_{2,2}]$ $i?$ $[\varphi_{2,2} \wedge \varphi'_{2,1}]$
$\langle p_2, tt\rangle$
$\langle p_3, tt\rangle$
$\langle p_4, tt\rangle$
$\langle p_5, tt\rangle$

Fig. 23 IAM $red(P)$ for IAM P of Fig. 22

$V = \{x, y\}$ and σ_0 maps x and y to 1. We write $\varphi_{i,j}$ instead of $x = i \wedge y = j$ and, therefore, $\varphi'_{i,j}$ instead of $x' = i \ \wedge\ y' = j$, for $i, j \in \mathbb{D}$. In some of the post-conditions in *red*(P), we have omitted some redundant conjuncts like *tt*. We assume that each of p_2, p_3, p_4 and p_5 has a characteristic behaviour that is omitted in the figures, e.g. each has an input-transition not occurring elsewhere. This way, no alternating simulation can relate any $\langle p_i, \varphi_J\rangle \in red(P)$ to some $p_j \in P$ with $i, j \in \{2, 3, 4, 5\}$ and $i \neq j$.

Suppose that $red(P) \sqsubseteq_{IAM} P$ due to some alternating simulation that relates $\langle p_0, x = 1\rangle$ to p_0. Due to the o-transition from $\langle p_0, x = 1\rangle$ to $\langle p_1, \varphi_{1,1}\rangle$, the simulation also relates $\langle p_1, \varphi_{1,1}\rangle$ to p_1. Now, the first problem is that $p_1 \mathrel{-\!\!\left(x = 2, i?, \varphi'_{2,1}\right)\!\!\rightarrow_P} p_5$ must be matched by an i-transition from $\langle p_1, \varphi_{1,1}\rangle$ to $\langle p_5, tt\rangle$. But there is no such transition, because the common pre-condition $\varphi_{1,1}$ of the transitions from $\langle p_1, \varphi_{1,1}\rangle$ contradicts $x = 2$. More generally speaking, the index of the i-transition above is not in the p_1-index set giving rise to $\varphi_{1,1}$. Such a contradiction cannot occur if all pre-conditions of input-transitions in P are tautologies, which implies input-determinism.

The second problem is similar. Transition $p_1 \mathrel{-\!\!\left(x = 1, i?, \varphi'_{1,1}\right)\!\!\rightarrow_P} p_2$ must be matched by the only i-transition from $\langle p_1, \varphi_{1,1}\rangle$. However, due to transition $p_1 \mathrel{-\!\!\left(y = 1, o!, \varphi'_{1,1}\right)\!\!\rightarrow_P} p_3$

and the construction of $red(P)$, we have the pre-condition $\varphi_{1,1}$ in the latter IAM, and this is not implied by $x = 1$. This effect can also occur if all pre-conditions of input-transitions in P are tautologies.

Consider the transition $\langle p_0, x = 1\rangle \xrightarrow{\left(x = 1, o!, x = 1 \wedge \varphi'_{1,1}\right)} \langle p_1, \varphi_{1,1}\rangle$ in $red(P)$ to see the last problem. This must be matched by the only o-transition from p_0. By the construction of $red(P)$, the post-condition of the given transition has an additional conjunct that makes the post-condition stricter than $x' = 1$. This also contradicts the definition of alternating simulation. □

Finally, observe that also $ia(P) \sqsubseteq_{IA}$ *Spec*, but $ia(P)$ should not be used in place of $ia(red(P))$. It might have outputs that cannot ever occur in the data-closed P, i.e. outputs for which the environment does not have to care.

8 Conclusions and future work

Interface Automata (IA) [12,13] have laid the foundation for reasoning about the compatibility of concurrent system components and been extended in the literature in various ways by shared variables for modelling data manipulating operations [1,2,9,11,27]. However, all these works consider a more complex setting than IA [1,2], do not relate their proposed semantics to an intuitive ground semantics [1,2,9,11,27], or do not abstract from internal computation [1,2,9,11,27]. Thus, it has remained an open question whether IA permits a simple and faithful extension to shared memory.

This article answered this question positively. Our interface theory IAM is a conservative extension of IA by shared memory. Similar to [1,2,9,27], we decorated action-labelled transitions with pre- and post-conditions constraining data states. A pre-condition acts as the transition's guard, and a post-condition of an output (input) transition specifies an assumption (guarantee) on the data state reached when executing the transition. We extended IA's concepts of compatibility, refinement and parallel composition to this setting, and provided a translation from IAM to IA. The latter shows that IAM accurately lifts IA's concepts to shared memory. In this sense, IAM keeps the simplicity that has made IA increasingly popular in the formal methods community. But the clear advantage is that IAM attains finiteness even when reasoning about infinite data domains. For our IAM theory, we implemented an initial, prototypic toolset [28] using open-source technologies and applied it to a simple example, thereby demonstrating that practical tool support is conceivable.

To prove that IAM treats shared variables as intuitively expected and that a cluster of IAMs can be abstracted to an IA, we provided a ground semantics for data-closed IAMs. This makes the data states, which are implicit in an IAM, explicit, and is not unlike the implementation semantics of [2]. Most importantly, our presentation of the usually infinite ground semantics is often finite. Data-closed IAMs may occur as a cluster of IAMs that communicate among each other by shared memory, but via message passing with the system environment. Thus, the cluster looks externally similar to an IA. We sketched an approach, where under some restrictions, an IA as a specification is transformed into an IAM, which in turn can be implemented by a cluster according to IAM-refinement. Then, the ground semantics of the cluster IA-refines the specification, and it can be turned into a bisimilar finite IA obtained from a reduction of data-closed IAMs.

Future work We propose to extend IAM with operators enhancing its practicality, namely a conjunction operator that is needed when a component must satisfy several interfaces, as well as operators for action scoping and variable scoping. The former is not as easy as for IA [23], where two interfaces always have a common refinement since contradictions cannot occur. In IAM, contradictions may arise due to transitions with conflicting post-conditions. Action scoping can be realized by pruning inputs and hiding outputs [8]. Variable scoping requires the introduction of an access control to IAM's global variables [1,2,11].

Further, we plan to attach data invariants to IAM states. The idea is that, while a component is in a state, a compatible system environment can only alter data in ways respecting the invariant. This restricts the environments with which a component can be composed, thus making shared variables more meaningful wrt. an open systems view. Variations like this bring our theory closer to practical application; we then have to adapt our cluster result accordingly, while replacing IA by message-passing systems.

Last, but not least, we wish to investigate the extent to which our prototypic IAM toolset scales to realistic systems and whether it can cope with the variety of concurrent systems developed in engineering practice. Certainly, scalability also depends on the logic chosen for expressing pre- and post-conditions, which effects the computational complexity for deciding alternating simulation.

Acknowledgements We thank the anonymous reviewers and also the reviewers of the extended abstract of this article [16] for their helpful comments and suggestions.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Bauer, S.S., Hennicker, R., Bidoit, M.: A modal interface theory with data constraints. In: SBMF, volume 6527 of LNCS, pp. 80–95. Springer, Berlin (2010)
2. Bauer, S.S., Hennicker, R., Wirsing, M.: Interface theories for concurrency and data. Theoret. Comput. Sci. **412**(28), 3101–3121 (2011)
3. Bauer, S.S., Mayer, P., Schroeder, A., Hennicker, R.: On weak modal compatibility, refinement, and the MIO Workbench. In: TACAS, volume 6015 of LNCS, pp. 175–189. Springer, Berlin (2010)
4. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T.A., Larsen, K.G.: Contracts for system design. Found. Trends Electr. Des. Autom. **12**(2–3), 124–400 (2018)
5. Bujtor, F., Fendrich, S., Lüttgen, G., Vogler, W.: Nondeterministic modal interfaces. Theoret. Comput. Sci. **642**(C), 24–53 (2016)
6. Bujtor, F., Vogler, W.: Error-pruning in interface automata. Theoret. Comput. Sci. **597**, 18–39 (2015)
7. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems–improvements in expressivity and usability. In: TACAS, volume 11428 of LNCS, pp. 21–39. Springer, Berlin (2019)
8. Chilton, C., Jonsson, B., Kwiatkowska, M.: An algebraic theory of interface automata. Theoret. Comput. Sci. **549**, 146–174 (2014)
9. Chouali, S., Mountassir, H., Mouelhi, S.: An I/O automata-based approach to verify component compatibility: application to the CyCab car. ENTCS **238**(6), 3–13 (2010)
10. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017)

11. de Alfaro, L., da Silva, L.D., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable interfaces. In: FroCoS, volume 3717 of LNCS, pp. 81–105. Springer, Berlin (2005)
12. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/FSE, pp. 109–120. ACM (2001)
13. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: ETSIS, volume 195 of NAII, pp. 83–104. Springer, Berlin (2005)
14. Dill, D.L.: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. MIT Press, Cambridge (1989)
15. Fendrich, S., Lüttgen, G.: A generalised theory of interface automata, component compatibility and error. Acta Inf. **56**(4), 298–319 (2019)
16. Gareis, J., Lüttgen, G., Schinko, A., Vogler, W.: Interface automata for shared memory. In: Models, Mindsets, Meta: The What, the How, and the Why Not?—Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday, volume 11200 of LNCS, pp. 151–166. Springer, Berlin (2018)
17. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3), 16:1-16:58 (2012)
18. Holik, L., Isberner, M., Jonsson, B.: Mediator synthesis in a component algebra with data. In: Correct System Design, volume 9360 of LNCS, pp. 238–259. Springer, Berlin (2015)
19. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. SIGPLAN Not. **43**(1), 273–284 (2008)
20. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Inf. Comput. **86**(1), 43–68 (1990)
21. Larsen, K.G.: Modal specifications. In: Automatic Verification Methods for Finite State Systems, volume 407 of LNCS, pp. 232–246. Springer, Berlin (1989)
22. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: ESOP, volume 4421 of LNCS, pp. 64–79. Springer, Berlin (2007)
23. Lüttgen, G., Vogler, W.: Modal interface automata. Log. Methods Comput. Sci. **9**(3:4), 1–28 (2013)
24. Mateescu, R., Oudot, E.: Improved on-the-fly equivalence checking using boolean equation systems. In: SPIN, volume 5156 of LNCS, pp. 196–213. Springer, Berlin (2008)
25. Meyer, B.: Applying "Design by Contract". IEEE Comput. **25**(10), 40–51 (1992)
26. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
27. Mouelhi, S., Chouali, S., Mountassir, H.: Refinement of interface automata strengthened by action semantics. ENTCS **253**(1), 111–126 (2009)
28. Nguyen, N.T., Lüttgen, G. The IAM Toolset, 2021. Available at GitHub: https://github.com/uniba-swt/ia-toolset
29. Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: A modal interface theory for component-based design. Fundam. Inform. **108**(1–2), 119–149 (2011)
30. Schinko, A., Vogler, W.: Fault-free refinements for interface automata. Sci. Ann. Comp. Sci. **28**, 289–337 (2018)
31. Vogler, W., Lüttgen, G.: A linear-time branching-time perspective on interface automata. Acta Inform. **57**(3), 513–550 (2020)