10.20378/irb-97637

BAMBERGER BEITRÄGE

ZUR WIRTSCHAFTSINFORMATIK UND ANGEWANDTEN INFORMATIK

ISSN 0937-3349

Nr. 107

Concurrency, Shared Variables, Compositionality: An Unlikely Triple

Eugene Yip Gerald Lüttgen

August 2024

FAKULTÄT WIRTSCHAFTSINFORMATIK UND ANGEWANDTE INFORMATIK

OTTO-FRIEDRICH-UNIVERSITÄT BAMBERG

Lehrstuhl Softwaretechnik & Programmiersprachen
Fakultät Wirtschaftsinformatik & Angewandte Informatik

Concurrency, Shared Variables, Compositionality: An Unlikely Triple

Research support provided by the DFG (German Research Foundation)
under grant nos. LU 1748/3-2 & VO 615/12-2 for the project
"Foundations of Heterogeneous Specifications Using State Machines and Temporal Logic"

Eugene Yip and Gerald Lüttgen
Software Technologies Research Group, University of Bamberg, Bamberg, Germany

August 23, 2024

Abstract: This paper reports our experiences with extending Interface Automaton (IA) with shared variables by lifting IA's intuitive notion of refinement and compositionality to shared variables. Although there are existing works that introduce shared variables to IA, they typically support a very restricted notion of sharing, e.g., the value of a shared variable is only defined for the duration of an atomic operation, with no ability to persist values for subsequent operations.

When attempting to formulate the semantics of shared variables that could persist their values across operations, we encountered numerous challenges when defining a notion of refinement that respected compositionality. We conjecture that, even for a basic notion of variable persistence, concurrent shared variable accesses between automata create a tight data dependency that prevents a compositional reasoning. We discuss the generality of this negative result in relation to other concurrency theories.

Contents

1 Introduction

Modern concurrent software systems range from multi-threaded programs on multi-core embedded processors, through to web services hosted on geographically distributed servers, that need to access a range of shared resources to fulfil their capabilities, e.g., program memory, files, peripherals, databases, or other web services. The operational behaviour for such a wide range of systems can be modelled using automata, such as *State Machines (SMs)*, running concurrently. SM-based modelling languages and tools include UML State Machines [1], MATLAB Stateflow [2], SCADE State Machines [3], Modelica State Machines [4], and KIELER SCCharts [5]. A SM captures a software component and its interaction with the environment, its computation steps triggered by actions, and its possible execution states. While SMs can communicate by synchronising their computation steps over *shared actions*, SMs can also be extended with *shared variables* to allow the communication of data among SMs. Shared variables can model shared resources at different levels of granularity, e.g., from program variables to peripherals and devices. It is the responsibility of the system designer to constrain shared variable accesses to ensure correct behaviour, e.g., that data is not corrupted by race conditions or improperly defined critical sections. Control and data dependencies among SMs, due to synchronisation and shared variables, hinder the ability of teams to work independently and incrementally on their own components, unless they adhere to a system-wide access protocol to avoid concurrency-related errors.

The *design-by-contract* methodology [6, 7] promotes the use of *contracts* as a way to formalise component behaviour, thus, aiding in the development of robust software systems [8]. Contract-based modelling languages and libraries include Eiffel [9], Ada [10], Dafny [11], and Kotlin [12]. A component's contract defines logical predicates that assert assumptions or guarantees about its state: a *precondition* defines the assumptions (or requirements) that need to be satisfied when calling the component, a *postcondtion* defines the guarantees that the caller can expect to receive, and an *invariant* defines what has to be satisfied during the component's execution. If any of these assertions fail, the component's execution is not guaranteed to be correct. Design-by-contract enables the modular design of concurrent systems because teams can develop against component contracts without needing to know how other components have been implemented. However, the authors [7] note that contracts need to be evaluated with global knowledge of the system in order to reason with shared states. Thus, run-time monitoring is typically required to handle contract violations.

As argued by *Interface Automata (IAs)* [13], the internal behaviour of a system component should be abstracted away into an *interface* that exposes only the information needed to determine whether it can operate properly with others, i.e., their *compatibility*. The interface, captured as a labelled transition system, serves as a protocol contract between the component and its environment: it describes the methods or services that the component can offer throughout its lifecycle, and it guarantees the correctness of its operation when its assumptions about the environment are satisfied. Unlike design-by-contract, IA formally defines an associative and commutative parallel operator for composing components, i.e., the same system behaviour is obtained no matter the order in which the components were composed; and a refinement relation for deciding whether a component conforms to the interface of another. IA guarantees that a component can always be replaced by a refinement without introducing new errors into a composed system, and that a composition of refined components will itself refine the composition of original components. Such a *compositionality* result enables the independent and incremental development and reuse of components. However, IA lacks support for shared variables, making it unsuitable in the design of practical software systems.

Given IA's theoretical advantages for component development, many attempts have been made to extend IA and similar formalisms [14] to capture data or shared memory. However, all have various limitations that make them impractical for real systems. In IA, a method call or service invocation is modelled by an output transition that synchronises with an input transition that provides the method itself. *IA for Shared Memory (IAM)* [15] conservatively extends IA with shared variables and contracts on transitions without imposing restrictions on compositionality. Each transition is an atomic computation step. When a method is called, the output transition's precondition must guarantee the data state expected by input's precondition, while the input's postcondition must guarantee the data state expected by the output's postcondition. However, IAM components suffer from amnesia at each state because the environment can freely modify the variables to any value. Such nondeterminism prevents the modelling of stateful systems, where values need to be recalled in a later step. In [16], a component owns local variables and can read or modify them, but others can only read them. Thus, variables can only be shared by pairs of components. In [17], each action has fixed pre- and postconditions, output transitions cannot be refined, and no compositionality proof has been provided. Moreover, the states of [16, 17] also suffer from amnesia. In [18], variables can only be assigned values and not expressions involving variables, synchronous transitions must have the same pre- and postconditions, and there is no notion of refinement.

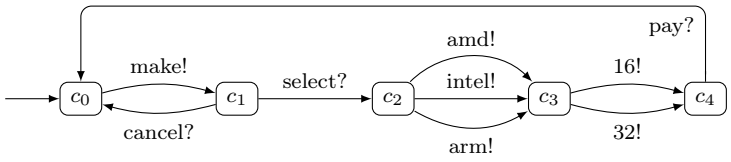
In this article, we report on our attempts to develop a practical IA theory for shared variables. Section 2 revisits IA in more detail, describes the latest shared memory extension to IA called *IA for Shared Memory (IAM)*, and reviews the main limitation of IAM, namely the data amnesia at each state. Section 3 attempts to address this limitation by proposing a stateful rendering of IAM that includes a constraint and variable ownership model for controlling concurrent shared variable accesses. We detail our main attempts to define a parallel operator and refinement relation that has the compositionality property needed for independent and incremental development. Even by simplifying IAM down to its essence, i.e., transitions with only preconditions on shared variables, no such compositionality result has been possible. We conjecture that shared variables introduce a tight control-flow dependency between concurrent components, which opposes the goals of independent development. This suggests that concurrent systems that require shared resources are fundamentally non-compositional and can only be developed modularly at best. Section 4 reflects on this conclusion by discussing its generality to software systems and to message passing IA formalisms that are compositional. Section 5 concludes this article with final thoughts.

2 Background

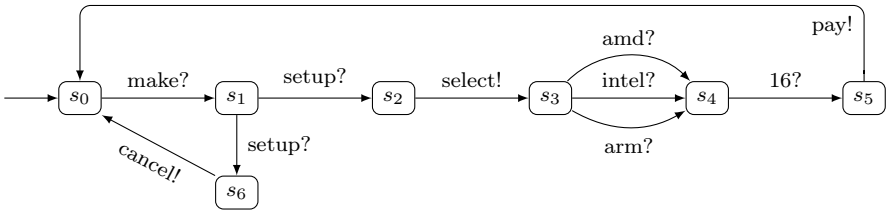
We briefly and informally recall *Interface Automaton (IA)* and the philosophy behind the independent and incremental development of software components in terms of IA's parallel composition operator and refinement relation. We then review *IA for Shared Memory (IAM)*, the latest extension of IA with shared memory, on which we conducted our work. We refer the reader to the original papers on IA [13] and IAM [15] for full details.

2.1 Interface Automata (IA)

An Interface Automaton (IA) specifies the input and output behaviour of a component as a labelled transition system. Each transition is labelled with an input, output, or internal action, denoted $a?$, $a!$, and τ, respectively. Two IA components can share an action a, where one offers the output $a!$ to the other's input $a?$. When two components synchronise



(a) IA *Customer* C, with $C = \{c_0, c_1, c_2, c_3, c_4\}$, $I_C = \{\text{cancel, select, pay}\}$, and $O_C = \{\text{make, amd, intel, arm, 16, 32}\}$.



(b) IA *Store* S, with $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$, $I_S = \{\text{make, setup, amd, intel, arm, 16, 32}\}$, and $O_S = \{\text{cancel, select, pay}\}$.

Figure 1: Example IA components *Customer (C)* and *Store (S)*.

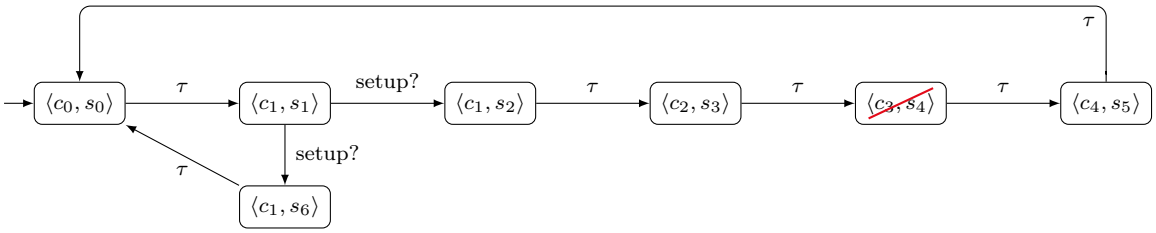
on a shared action, internal τ-transitions are created. Otherwise, components interleave on all other actions. A component always waits patiently on an input transition to receive a matching output, but can always take an output or internal τ-transition immediately—they are *autonomous* transitions because they can be taken without waiting. If a component cannot immediately match with an output, a *communication error* occurs.

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

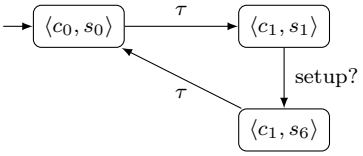
- P is the set of *states* with $p_0 \in P$ being the *initial state*;
- $A =_{\text{df}} I \cup O$ is the *alphabet* of disjoint sets of *input* I and *output* O *actions*, without the distinguished *internal action* τ; and
- $\rightarrow \subseteq P \times (A \cup \{\tau\}) \times P$ is the *transition relation*, and a transition $(p, \alpha, p') \in \rightarrow$ is written intuitively as $p \xrightarrow{\alpha} p'$. The IA is required to be *input-deterministic*: for $p \in P$ and $a \in A$, if $p \xrightarrow{a?} p'$ and $p \xrightarrow{a?} p''$, then $p' = p''$.

We simply refer to an IA $(P, I, O, \rightarrow, p_0)$ as P.

Example 1. Figure 1(a) is an IA that models a customer who wishes to configure and purchase a computer at a store. They begin by requesting the *make* service and transition to state c_1. They wait for the store to decide whether they can go ahead and *select* the computer parts or to *cancel* the transaction because the service area is busy. If the component selection can proceed, the customer picks a branded CPU (AMD, Intel, or ARM), the amount of RAM (16GB or 32GB), and wait for the store to ask them to *pay*. Figure 1(b) is an IA that models the transaction from the store's perspective. The store begins by waiting for a customer request to *make* a computer, and then waits for their service area to be *set up*. If the service area is busy, the store requests the customer to *cancel* the transaction. Otherwise, the service area is available and the store requests the customer to *select* their computer parts. After the store has processed the desired CPU and RAM, they request the customer to *pay*. Note how input action *32* is specified in the interface of S, but is never offered by the IA itself.



(a) Parallel product $C \otimes S$, with $C \otimes S = \{\langle c_0, s_0\rangle, \langle c_1, s_1\rangle, \langle c_1, s_2\rangle, \langle c_2, s_3\rangle, \langle c_3, s_4\rangle, \langle c_4, s_5\rangle, \langle c_1, s_6\rangle\}$, $I_{C\otimes S} = \{\text{setup}\}$, and $O_{C\otimes S} = \emptyset$. State $\langle c_3, s_4\rangle$ is crossed out because it has a communication error.



(b) Parallel composition $C \parallel S$.

Figure 2: Parallel product and composition of the IA components *Customer (C)* and *Store (S)* from Figure 1.

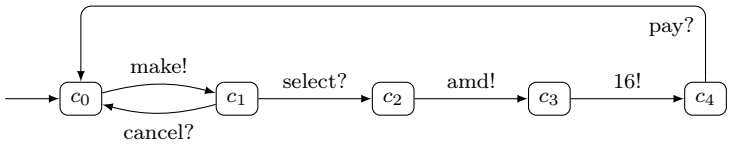
IA takes the optimistic view that a component's environment is helpful in avoiding communication errors, i.e., the environment does not offer an output if the component is not in a state with a matching input transition. Two IAs P and Q are *composable* if $O_P \cap O_Q = \emptyset$ and $I_P \cap I_Q = \emptyset$. The *parallel composition* of two such IA components, which defines their parallel behaviour (semantics), is computed in two phases:

- their product automaton is computed to determine their combined state space, where both components take transitions together on shared actions and interleave on all other actions; and
- states with communication errors, and states that can autonomously reach the error states, are pruned from the product automaton.

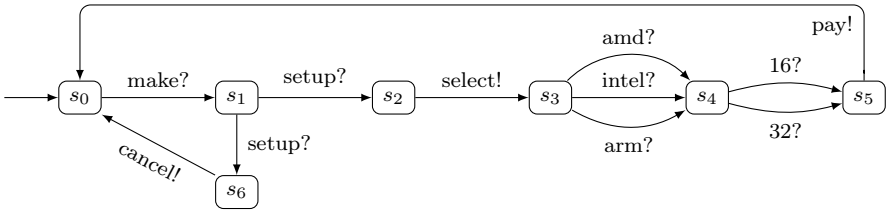
The resulting automaton, containing only "good behaviours", is the parallel composition, and has inputs $I =_{\text{df}} (I_P \cup I_Q) \setminus (A_P \cap A_Q)$ and outputs $O =_{\text{df}} (O_P \cup O_Q) \setminus (A_P \cap A_Q)$.

Example 2. Figure 2(a) is the product IA of the *Customer (C)* and *Store (S)* from Figure 1. By inspecting the inputs and outputs of C and S, we see that they synchronise on the following actions: *cancel*, *select*, *pay*, *make*, *amd*, *intel*, *arm*, *16*, and *32*. Only the output action *setup* is asynchronous. The initial state of the parallel product is $\langle c_0, s_0\rangle$, where both C and S synchronise on *make* and transition to $\langle c_1, s_1\rangle$. At this state, C must wait for S to make a *select* request. S can take a nondeterministic transition to s_2 or s_6, i.e., the product states $\langle c_1, s_2\rangle$ and $\langle c_1, s_6\rangle$. For all remaining states, C and S take synchronised transitions. Note how at state $\langle c_3, s_4\rangle$, C has output transitions with actions *16* and *32*, but S is can only handle action *16*. Thus, a communication error occurs at state $\langle c_3, s_4\rangle$ for output action *32*; indicated in Figure 2(a) by $\langle c_3, s_4\rangle$ being crossed out. To compute the parallel composition in Figure 2(b), we prune away the error state $\langle c_3, s_4\rangle$ and all the states that can autonomously reach it, i.e., $\langle c_1, s_2\rangle$ and $\langle c_2, s_3\rangle$ because of the internal τ-transitions. We can conclude that, when the store does not have a computer part, the only sensible behaviour is to turn the customer away.

When developing a component, it is desirable to *refine* an existing component specification (or abstraction) into an implementation that is more concrete and guarantees a monotonic



(a) Refined *Customer C'*.



(b) Refined *Store S'*.

Figure 3: Refinement of the IA components from Figure 1.

decrease in communication errors. This implies that the implementation becomes more deterministic and does not introduce new communication errors. IA formalises the refinement of specification Q by implementation P using the notion of *alternating simulation*:

- whenever specification Q can take an input transition on $a?$, implementation P can as well;
- whenever implementation P can take an output transition on $a!$, specification Q can take zero or more internal τ-transitions followed by an output transition on $a!$; and
- whenever implementation P can take an internal τ-transition, specification Q can take zero or more as well.

We say that P IA-refines Q, written $P \sqsubseteq_{IA} Q$, if such a relation exists between P and Q. The contra-variant definition of refinement is motivated by the desire to reduce the occurrence of communication errors: an implementation should be ready to receive more outputs, and should reduce the likelihood of producing an output that another component is not ready to receive.

Example 3. Figure 3(a) refines Figure 1(a), i.e., $C' \sqsubseteq_{IA} C$, because output transitions have been removed from states c_2 and c_3 while maintaining the existing input transitions. Figure 3(b) refines Figure 1(b), i.e., $S' \sqsubseteq_{IA} S$, because state s_4 now offers the input *32*, whose absence caused the communication error in Figure 2(a). The components C' and S would work together without any communication error and their parallel commposition would look like Figure 2(a) (ignoring that state $\langle c_3, s_4 \rangle$ has been crossed out). Likewise, the component pairs C and S', and C' and S' would each work together without any communication error.

The parallel composition and refinement definitions together allow IA components to be developed independently and incrementally, and this property is formally called *compositionality*.

Theorem 1 (IA Compositionality [13])**.** *Let P, Q, and R be IAs, $P \sqsubseteq_{IA} Q$, and Q and R be composable. We have: (1) P and R are composable; (2) $P \parallel R$ is defined if $Q \parallel R$ is, and so $P \parallel R \sqsubseteq_{IA} Q \parallel R$.*

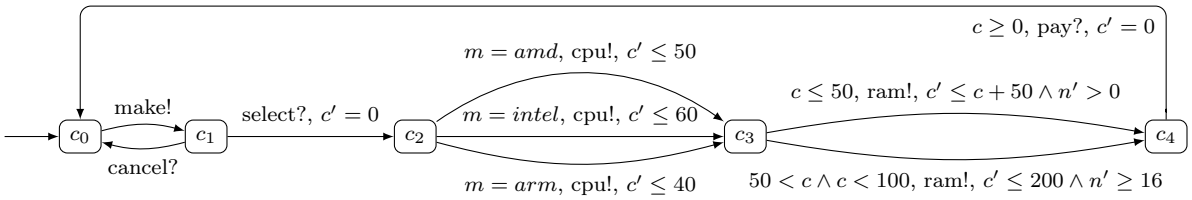
2.2 Interface Automata for Shared Memory (IAM)

An IA for Shared Memory (IAM) conservatively extends IA with shared variables in the sense that an IAM can be translated into an equivalent IA; the extensions are simply syntactic sugar. Hence, IAM inherits all the properties of IA, including compositionality. IAM extends IA with a set of global variables V and pre- and postconditions on transitions as logical predicates, written φ and ψ, respectively, which allow the abstract specification of data states. For example, let $V =_{\text{df}} \{x\}$ with x being an integer, then the precondition $\varphi =_{\text{df}} (0 < x) \wedge (x < 3)$ specifies the data state $\{x \in \{1, 2\}\}$. A postcondition can also use primed versions of variables to reference the values they will have when a transition completes. For example, the postcondition $\psi =_{\text{df}} (x' < x)$ specifies that x will have a smaller value when the transition completes. We write $\text{Pred}(V)$ to represent the universe of predicates over the variables in V.

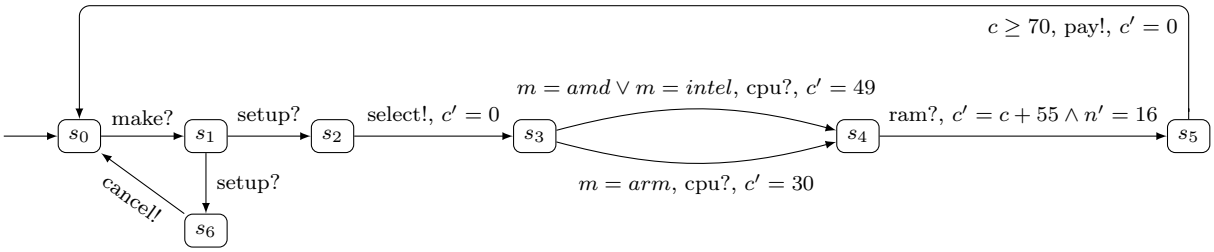
Definition 2 (IA for Shared Memory)**.** An *IA for Shared Memory (IAM)* is a tuple $(P, I, O, \rightarrow, p_0)$, where P, I, O, p_0 is the same as for IA (Definition 1), and $\rightarrow \subseteq P \times \text{Pred}(V) \times (A \cup \{\tau\}) \times \text{Pred}(V, V') \times P$ is the *transition relation* and $(p, \varphi, \alpha, \psi, p') \in \rightarrow$ is written intuitively as $p \xrightarrow{\varphi,\alpha,\psi} p'$. The states in P are also called *control states*. The IAM is required to be *data-deterministic* for input actions: for $p \in P$ and $a? \in I$, if $p \xrightarrow{\varphi_1,a?,\psi_1} p'$ and $p \xrightarrow{\varphi_2,a?,\psi_2} p''$ are different transitions, then $\varphi_1 \wedge \varphi_2$ is unsatisfiable.

An input transition models a method or service that a component offers, while an output transition models a method call or service request. The input's precondition specifies the data state expected (or required) by the method, which the output's precondition must guarantee. The output's postcondition specifies the data state that it expects the method to return, which the input's postcondition must guarantee. As a shorthand, when a pre- or postcondition is trivially true, written tt, we omit it from the transition notation, e.g., $p \xrightarrow{tt,a?,\psi} p''$ is $p \xrightarrow{a?,\psi} p''$, and $p \xrightarrow{tt,a?,tt} p''$ is $p \xrightarrow{a?} p''$. An IAM component experiences a communication error when it is at a state with no input transition that can match an output, i.e., there is no input transition with a precondition that the output's precondition can satisfy, and no postcondition that implies the output's postcondition.

Example 4. Figures 4(a) and 4(b) are IAM interpretations of the *Customer (C)* and *Store (S)* components, respectively, from Figure 1. The IAMs use the global variables m and n to represent a CPU brand and RAM size, respectively. These variables could as well have represented the price, model number, description, or even a photo of the desired computer part. The global variable c is the cost (in \$) of the computer being configured. For IAM C, the *select?* transition from state c_1 to c_2 has a postcondition for the customer's insistance that the cost starts from \$0. Then, the *cpu!* transitions each have a precondition for the required CPU brand and a postcondition for the expected cost. The following *ram!* transitions each have a precondition on the current cost to determine the expected RAM size and updated cost. Lastly, the *pay?* transition has a precondition to require that the cost is non-negative and that the expected cost after paying is \$0. For IAM S, the pre- and postconditions are complementary to those of IAM C. The *cpu?* transition with precondition $m = amd \vee m = intel$ handles CPU requests for both the AMD and Intel brands, with a postcondition guaranteeing a price of \$49. The *ram?* transition guarantees that the RAM size is 16GB at an additional cost of \$55. Lastly, the *pay!* transition guarantees the cost is at least \$70 and then expects it to be \$0 after the has customer paid.



(a) IAM *Customer* C, with $C = \{c_0, c_1, c_2, c_3, c_4\}$, $I_C = \{\text{cancel, select, pay}\}$, and $O_C = \{\text{make, cpu, ram}\}$.



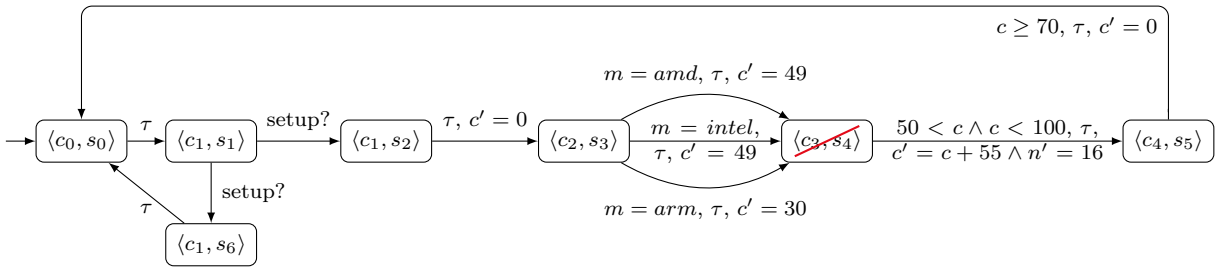
(b) IAM *Store* S, with $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$, $I_S = \{\text{make, setup, cpu, ram}\}$, and $O_S = \{\text{cancel, select, pay}\}$.

Figure 4: Example IAM components *Customer (C)* and *Store (S)*, based on the IA components from Figure 1. The set of global variables V is $\{m \in \{amd, intel, arm\}, n \in \mathbb{Z}, c \in \mathbb{Z}\}$.

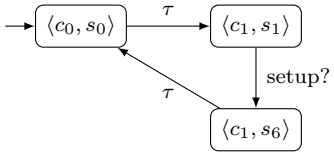
The parallel composition of two IAMs, P and Q, is more involved than for IA:

- their product automaton is computed, where both components synchronise on shared actions and interleave on all other actions. For synchronous transitions, their pre- and postconditions are taken into account: given an output transition $p \xrightarrow{\varphi_p, a!, \psi_p}_P p'$ and input transition $q \xrightarrow{\varphi_q, a?, \psi_q}_Q q'$, if $\varphi_P \wedge \varphi_Q$ is satisfiable and $\psi_Q \Rightarrow \psi_P$ under the data state that satisfies $\varphi_P \wedge \varphi_Q$, then the synchronisation results in the internal τ-transition $\langle p, q\rangle \xrightarrow{\varphi_p \wedge \varphi_q, \tau, \psi_q} \langle p', q'\rangle$; and
- control states with communication errors, and control states that can autonomously reach the error states, are pruned from the product automaton.

Example 5. Figure 5(a) is the product IAM of *Customer (C)* and *Store (S)* from Figure 4. The transitions between the initial state $\langle c_0, s_0\rangle$ and successor states $\langle c_1, s_1\rangle$, $\langle c_1, s_2\rangle$ and $\langle c_1, s_6\rangle$ are constructed similarly to an IA product because their pre- and postconditions are trivially true (tt). For state $\langle c_1, s_2\rangle$, state c_1 of C and state s_2 of S both synchronise on action *select* with equivalent pre- and postconditions. For state $\langle c_2, s_3\rangle$, the *cpu!* transitions of C with preconditions $m = amd$ and $m = intel$ each synchronises with the *cpu?* transition of S with precondition $m = amd \vee m = intel$; the input's postcondition implies each output's postcondition. For state $\langle c_3, s_4\rangle$, the *ram!* transition of C with precondition $50 < c \wedge c < 100$ synchronises with the *ram?* transition of S. Based on the precondition's data state, the input's postcondition implies the output's postcondition, i.e., $\{c \in [51, 99]\} \vDash (c' = c + 55 \wedge n' = 16) \implies (c' \leq 200 \wedge n' \geq 16)$ with the evaluation $n' = 16$ and $c' \in [106, 154]$. However, for the *ram!* transition with precondition $c \leq 50$, a synchronisation with *ram?* is not possible because the input's postcondition does not imply the output's postcondition: $\{c \in [-\infty, 50]\} \nvDash (c' = c + 55 \wedge n' = 16) \implies (c' \leq c + 50 \wedge n' > 0)$ since $c' = c + 55$ can never imply $c' \leq c + 50$ for all values of c. Hence, a communication error occurs at state $\langle c_3, s_4\rangle$, which is pruned away together with the automonomous states $\langle c_2, s_3\rangle$ and $\langle c_1, s_2\rangle$, resulting in the parallel composition of Figure 5(b).



(a) Parallel product $C \otimes S$, with $C \otimes S = \{\langle c_0, s_0 \rangle, \langle c_1, s_1 \rangle, \langle c_1, s_2 \rangle, \langle c_2, s_3 \rangle, \langle c_3, s_4 \rangle, \langle c_4, s_5 \rangle, \langle c_1, s_6 \rangle\}$, $I_{C \otimes S} = \{\text{setup}\}$, and $O_{C \otimes S} = \emptyset$. State $\langle c_3, s_4 \rangle$ is crossed out because it has a communication error.



(b) Parallel composition $C \parallel S$.

Figure 5: Parallel product and composition of the IAM components *Customer (C)* and *Store (S)* from Figure 4.

As we saw from Example 5, preconditions are only evaluated on their satisfiability and not on the actual data states of their source control state. In other words, the data state at each control state is inconsequential to the behaviour of a composition. Data is only exchanged beween two IAM components when they synchronise to execute an atomic operation; the environment is free to change the value of any variable at each state. The inability to persist data makes IAM unsuitable for modelling stateful concurrent systems.

IAM supports refinement by extending IA's alternating simulation to take pre- and postconditions into account. Recall that pre- and postconditions can specify an abstract set of data states, which can be decomposed into smaller, but more concrete, data states. For example, the condition $x \neq 0$ can be decomposed into $x < 0$ and $x > 0$. Thus, the behaviour of an IAM transition could be matched by multiple transitions, i.e., a *family* of transitions. The refinement of specification Q by implementation P is defined as follows:

- whenever specification Q can take an input transition on $a?$, implementation P can as well via a family of transitions that collectively has a weaker precondition and stronger postcondition;
- whenever implementation P can take an output transition on $a!$, specification Q has a family of transitions that can take zero or more internal τ-transitions followed by an output transition on $a!$ that collectively has a weaker precondition and stronger postcondition; and
- whenever implementation P can take an internal τ-transition, specification Q can take zero or more as well.

We say that P IAM-refines Q, written $P \sqsubseteq_{IAM} Q$, if such a relation exists between P and Q.

Example 6. Figure 6 refines Figure 4(a), i.e., $C' \sqsubseteq_{IAM} C$ because output transitions have been removed from states c_2 and c_3 while maintaining the existing input transitions. Moreover, the *ram!* transition at state c_3 has a stronger precondition and a weaker postcondition; the customer now determines their expected RAM size and cost on a narrower range, while their expectation on RAM size and additional cost has widened. The components C' (Fig-

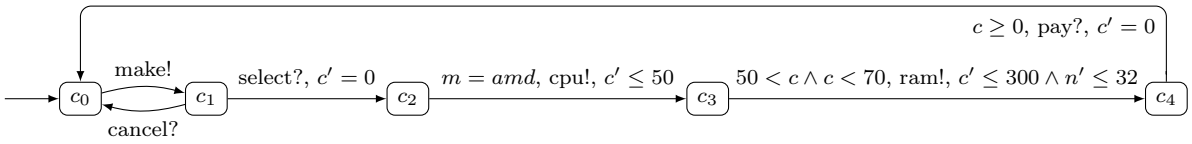


Figure 6: Refinement *C'* of *Customer C* from Figure 4(a).

Table 1: Possible data states for an execution of *Customer C'* from Figure 6

Control state	c_0	c_1	c_2	c_3
Data state	$m \in \{amd, intel, arm\}$ $n \in \mathbb{Z}$ $c \in \mathbb{Z}$		$m \in \{amd, intel, arm\}$ $n \in \mathbb{Z}$ $c = 0$	$m = amd$ $n \in \mathbb{Z}$ $c = 50$

ure 6) and S (Figure 4(b)) would work together without any communication errors.

3 Towards Stateful Interfaces

In this section, we develop a more practical variant of IAM, one that can model stateful software systems. More precisely, we want the values of shared variables to persist in each state, and the notion that components can own a set of shared variables and decide when other components are allowed to modify to them.

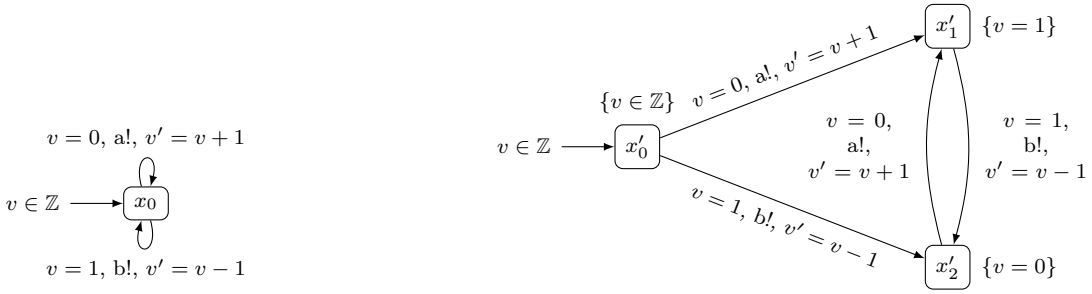
3.1 Stateful Variables

In practical software systems, variables are only modified during each computation step (i.e., the transitions), with the expectation that their values are preserved between each step (i.e., the states).

Example 7. Returning to IAM C' of Figure 6, we illustrate a stateful execution that produces the data states detailed in Table 1. Suppose that the initial control state c_0 has an initial data state of undefined values, i.e., each variable could be any value in its domain (see column c_0 in Table 1). C' begins by taking the *make!* transition to c_1 with no changes to the data state. Suppose the *select?* transition to state c_2 is taken, which updates variable c to 0. Next, to take the *cpu!* transition, variable m has to be *amd* to satisfy the precondition $m = amd$. This is possible because $m \in \{amd, intel, arm\}$ in state c_2. After taking this transition to state c_3, $m = amd$ and nondeterministically choose $c = 50$. To take the *ram!* transition, c has to satisfy the precondition $50 < c \wedge c < 70$. However, this is not possible because $c = 50$ in the data state of c_3. In fact, no possible execution of C' can produce a data state at c_3 that satisfies the precondition, so the transition can never be taken. In contrast, under the original semantics of IAM (Section 2.2), the transition can be taken because only the satisfiability of the precondition is considered; the data states are inconsequential.

A stateful IAM would need to consider the data states that can be reached at each control state to properly evaluate a transition's pre- and postconditions. This implies the need to track the evolution of a component's initial data state during execution. For a static analysis, this means a fixed point computation of data states over the reachable control states.

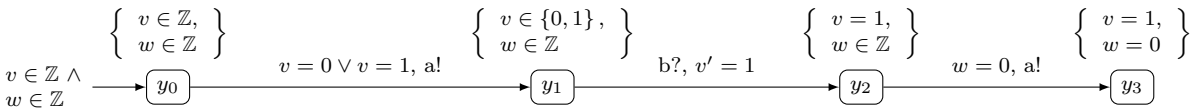
Example 8. We illustrate the tracking of data states with IAM component X in Figure 7(a) that has two self-loops and the integer global variable v. Before executing X, suppose that variable v has an undefined integer value, indicated by $v \in \mathbb{Z}$ on the initial arrow of



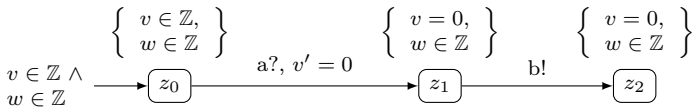
(a) IAM X, with global variables $V = \{v\}$.

(b) IAM X', with global variables $V = \{v\}$. Control states are labelled by their concrete data states.

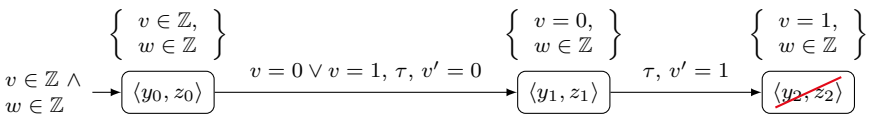
Figure 7: Example IAMs to demonstrate the tracking of data states.



(a) IAM Y, with $I_Y = \{b\}$, $O_Y = \{a\}$, and global variables $V = \{v, w\}$.



(b) IAM Z, with $I_Z = \{a\}$, $O_Z = \{b\}$, and global variables $V = \{v, w\}$.



(c) Parallel product $Y \otimes Z$, with $I_{Y \otimes Z} = \emptyset$, $O_{Y \otimes Z} = \emptyset$, and global variables $V = \{v, w\}$. Note that state $\langle y_2, z_2 \rangle$ may not actually experience a communication error on $a!$.

Figure 8: Example IAMs to demonstrate stateful parallel composition. Control states are labelled by their concrete data states.

Figure 7(a). For the $a!$ loop to be taken, variable v has to be 0 for the precondition and then updated to $0 + 1 = 1$ in the postcondition. Thus, the data state of x_0 afterwards would be $\{v = 1\}$. If instead the $b!$ loop was taken, variable v has to be 1, and the data state of x_0 afterwards would be $\{v = 0\}$. Concretely, component X alternates between the loops because the resulting data state of one loop can only satisfy the precondition of the other. Consequently, the data state of x_0 alternates between $\{v = 1\}$ and $\{v = 0\}$, rather than being initially $\{v \in \mathbb{Z}\}$ and then $\{v \in \{0, 1\}\}$ as implied by the depiction of x_0 as a single control state in Figure 7(a). Component X is more accurately modelled by X' in Figure 7(b) by enumerating x_0 by its concrete data states.

Example 8 reveals a major problem: for stateful IAMs with cyclic behaviour and global variables of infinite domain, their enumeration may be unbounded. However, even if we assume that it is possible to accurately enumerate the control states of an IAM, computing the parallel composition of such IAMs would require the tracking of data states to correctly evaluate the preconditions and to enumerate the composite control states. Such a tight coupling between data and control states does not bode well for the refinement of stateful IAMs.

Example 9. Figures 8(a) and 8(b) are two stateful IAMs components, Y and Z, that synchronise on actions a and b and operate on global variable v together; only component Y operates on w. Both v and w are integers and are initially undefined. To compute the parallel composition of Y and Z, we first compute their parallel product, shown in Figure 8(c). The initial state is $\langle y_0, z_0 \rangle$ with data state $\{v \in \mathbb{Z}, w \in \mathbb{Z}\}$. Components Y and Z synchronise on action a and arrive at $\langle y_1, z_1 \rangle$ with data state $\{v = 0, w \in \mathbb{Z}\}$. This data state is narrower than what state y_1 in component Y had expected. Next, they synchronise on action b and arrive at $\langle y_2, z_2 \rangle$ with data state $\{v = 1, w \in \mathbb{Z}\}$. Note how v has a completely different value to what state z_2 in component Z expected. This is because transition $z_1 \xrightarrow{b!}_Z z_2$ wrongly assumed that v would not be modified by another component. Finally, at state $\langle y_2, z_2 \rangle$, component Y can take its $a!$ transition, but component Z is not ready for the output, so a communication error occurs. If another component was to exist to make the modification $w \neq 0$, component Y would not be able to take its $a!$ transition and state $\langle y_2, z_2 \rangle$ would not experience a communication error. Thus, the actual occurrence of a communication error cannot be known until the system is closed off from further composition.

Thus far, we have seen that a stateful extension of IAM suffers from the following problems: control states need to be enumerated to disambiguate the reachable data states, and the correct evaluation of communication errors during parallel composition depends on to-be-composed components. The latter is the result of strong data dependencies between components and such dependencies have to be decoupled for compositionality to be achievable. Moreover, the behaviour of race conditions has to be addressed, which our examples have been carefully designed to avoid. Section 3.2 explores the possibility of abstracting data states judiciously to avoid the unbounded enumeration of control states, and Section 3.3 discusses the modelling of variable ownership to control concurrent modifications to shared variables.

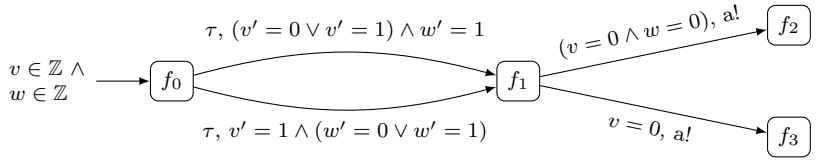
3.2 Abstract Statefulness

Example 8 highlighted that the data state of an IAM control state can evolve over time and that the control state can be enumerated to disambiguate between the possible date states. Since this can cause an unbounded enumeration of control states, we now explore whether enumeration can be avoided by abstracting the data states without sacrificing too much precision.

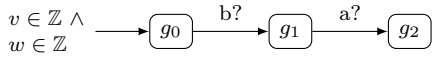
When a control state has multiple incoming transitions, we have the problem of reconciling (or summarising) the incoming data states into a single (abstract) data state. How should the data states be abstracted? We could take either the *union* or *intersection* of the incoming data states.

Example 10. Returning to component X in Figure 7(a), the possible data states of control state x_0 are $\{v \in \mathbb{Z}\}$ from initialisation, and $\{v = 0\}$ and $\{v = 1\}$ from the two self-loops. If we take the union, we have the abstract data state $\{v \in \mathbb{Z}\}$, but this implies that each loop can be taken consecutively, which is not actually possible. Moreover, the value of variable v would again be forgotten by the abstraction. If we instead take the intersection, we have an invalid abstraction because $\mathbb{Z} \cap \{0\} \cap \{1\} = \emptyset$, and no transitions can be taken.

From this example, we see that taking the union or intersection of data states results in an unsatisfactory abstraction; it is either an over-approximation or under-approximation, respectively, that leads to an optimistic or pessimistic interpretation of component behaviour.



(a) IAM F, with $I_F = \emptyset$, $O_F = \{a\}$, and global variables $V = \{v, w\}$.



(b) IAM G, with $I_G = \{a, b\}$, $O_G = \emptyset$, and global variables $V = \{v, w\}$.

Figure 9: Example IAMs to demonstrate the abstraction of data states.

Table 2: Concrete and abstract data states of control state f_1 from Figure 9(a)

Concrete	**Over-approximation**	**Under-approximation**
$\{v \in \{0,1\}, w = 1\}$ or $\{v = 1, w \in \{0,1\}\}$	$\{v \in \{0,1\}, w \in \{0,1\}\}$	$\{v = 1, w = 1\}$

In addition, an imprecise abstraction of data states can mask the existence of communication errors during parallel composition.

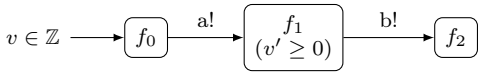
Example 11. We wish to compose components F and G from Figures 9(a) and 9(b) together, which only synchronise on action a. The internal τ-transitions from f_0 to f_1 in F modify the global variables v and w. Component G waits for input b followed by a without modifying any global variables. Suppose component F takes the top internal τ-transition from f_0 to f_1, resulting in the concrete data state $\{v \in \{0,1\}, w = 1\}$. Based on this data state, only the bottom *a!* transition to state f_3 is possible. This will cause a communication error if component G is still in state g_0. If component F had taken the bottom internal τ-transition to f_1, then the resulting concrete data state $\{v = 1, w \in \{0,1\}\}$ would not satisfy the precondition of either *a!* transitions and component F would remain in f_1.

Consider now that we *over-approximate* the concrete data states of f_1 by taking their union, i.e., $\{v \in \{0,1\}, w \in \{0,1\}\}$. This abstract data state would (incorrectly) satisfy the precondition of the *a!* transition from f_1 to f_2 and taking it would cause a communication error; this is a *false positive* error caused by an imprecise abstraction. Conversely, if we *under-approximate* the concrete data states of f_1 by taking their intersection, i.e., $\{v = 1, w = 1\}$, it would (incorrectly) not satisfy the precondition of either *a!* transitions; this is a *false negative* that hides a concrete error that would not pruned from the parallel composition, leading to serious correctness issues. The concrete and abstract data states are summarised in Table 2.

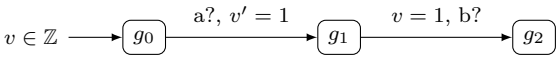
Communication errors could indeed be detected at run-time, but this would defeat the goal of independent and incremental development; the benefits of compositionality. We do not see an elegant solution that avoids the enumeration of control states.

3.3 Variable Ownership

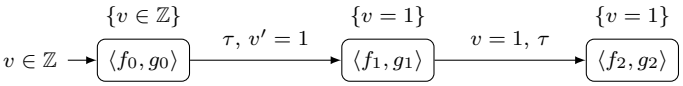
Race conditions occur when a pair of IAM components synchronise together and both modify the same shared variable in their postconditions. Some well-known programming languages incorporate the idea of variable ownership as a means to improve memory management,



(a) IAM F, with $I_F = \emptyset$, $O_F = \{a, b\}$, global variables $V = \{v\}$, and owns variable v.



(b) IAM G, with $I_G = \{a, b\}$, $O_G = \emptyset$, global variables $V = \{v\}$, and does not own any variables.



(c) Parallel composition $F \parallel G$, with global variables $V = \{v\}$, and owns variable v. Control states are labelled by their concrete data states.

Figure 10: Example IAMs to demonstrate variable ownership.

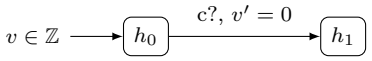
e.g., Rust [19] with ownership and borrowing semantics, C++ [20] with smart pointers and move semantics, Eiffel [9] with objects marked as *separate*, and SPARK [21] with a memory ownership policy. Perhaps we could decouple the data dependencies between concurrent components by defining an interface for shared variables.

The idea would be for a component to define a set of shared variables that it owns exclusively. The component can freely modify its own shared variables and, while it is in a state and not executing, it can allow other components to modify its shared variables. At a control state where external modification is allowed, it is labelled with an *Allows constraint* that specifies the values the shared variables can be modified to. During parallel composition, transitions that modify another component's shared variable can only be taken if they also satisfy that component's Allows constraint. To model mutual exclusion (e.g., lock acquire and release), an Allows constraint is "consumed" when an external modification is successful.

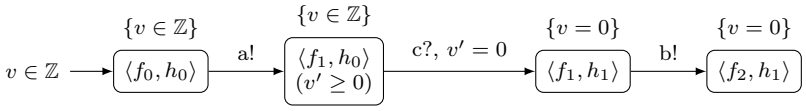
Example 12. Figure 10(a) depicts component F that owns global variable v. State f_1 has an Allows constraint $(v' \geq 0)$ that allows other components to modify v to a positive value. At this state, component F is not allowed to modify v, so all transitions to f_1 cannot contain v' in their postcondition. Let us compose F with component G depicted in Figure 10(b). Both components can synchronise on action a because the postcondition of G's transition satisfies the Allows constraint of state f_1 in F. The Allows constraint is consumed and the resulting data state is $\{v = 1\}$. Next, the components synchronise on action b. The parallel composition is depicted in Figure 10(c).

Unfortunately, such a shared variable interface still does not help to decouple data dependencies between components. Depending on the order in which components are composed together, different compositions may be achieved. In fact, an error-free synchronisation for a given composition may turn into a communication error when components are composed in a different order.

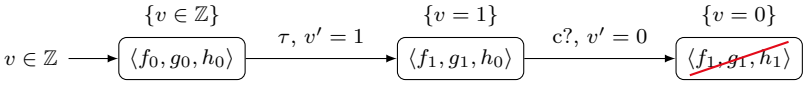
Example 13. When component F from Figure 10(a) is composed with component H depicted in Figure 11(a), we get $F \parallel H$ shown in Figure 11(b). When $F \parallel H$ is composed with component G from Figure 10(b), G experiences a a communication error at state g_1 because it is no longer able to handle the output action b. The parallel product demonstrating this communication error is depicted in Figure 11(c). We see that the data state at $\langle f_1, g_1, h_1 \rangle$ is $\{v = 0\}$ rather than $\{v = 1\}$ at state $\langle f_1, g_1 \rangle$ of $F \parallel G$ shown previously in Figure 10(c).



(a) IAM H, with $I_H = \emptyset$, $O_H = \emptyset$, global variables $V = \{v\}$, and does not own any variables.

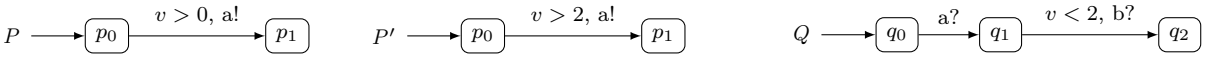


(b) Parallel composition $F \parallel H$, with global variables $V = \{v\}$, and owns variable v. Control states are labelled by their concrete data states.



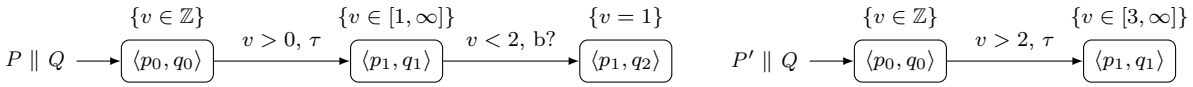
(c) Parallel product $(F \parallel H) \otimes G$, with global variables $V = \{v\}$, and owns variable v. Control states are labelled by their concrete data states.

Figure 11: Example IAMs to demonstrate variable ownership problems.



(a) IAMs P and P', where $P' \sqsubseteq_{IAM} P$.

(b) IAM Q.



(c) Parallel compositions $P \parallel Q$ and $P' \parallel Q$, where $P' \parallel Q \not\sqsubseteq_{IAM} P \parallel Q$. Control states are labelled by their concrete data states.

Figure 12: Example IAMs to demonstrate problems with refining output transitions. All IAMs access the global variable $v \in \mathbb{Z}$, which is not owned by any IAMs.

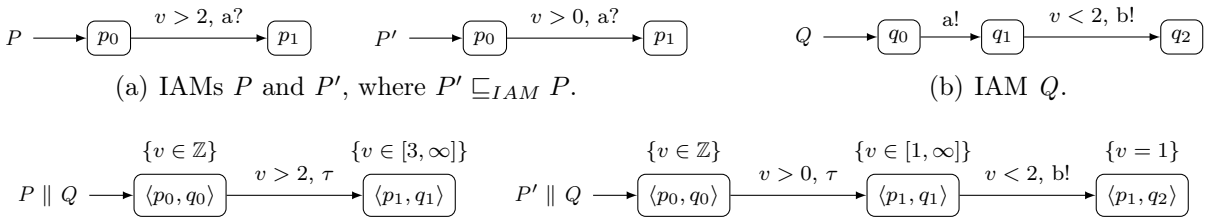
Hence, the occurrence of communication errors is sensitive to the order in which components F, G, and H are composed, and this sensitivity severely hinders compositionality.

While variable ownership and Allows constraints help to define an interface for modifying shared variables, they do not weaken the tight coupling of components caused by data dependencies.

3.4 Stateful Refinement

Let us put aside all the issues identified so far in Section 3 and focus on defining a stateful refinement of IAM. In classical IA [13], the alternating simulation refinement of input and output transitions is defined in a contra-variant manner: $P \sqsubseteq_{IA} Q$ if implementation P supports more inputs, but less outputs, of its specification Q. In classical IAM [15], refinement follows the same philosophy: for an implementation P to support more inputs, but less outputs, than its specification Q, the input and output transitions of P must have, respectively, weaker and stronger preconditions than those of Q. The postconditions of implementation P's input and output transitions are, respectively, stronger and weaker than that of specification Q.

Example 14. Figure 12(a) depicts two IAM components P and P', where $P' \sqsubseteq_{IAM} P$ according to IAM refinement. When they are each composed with component Q in Fig-



(a) IAMs P and P', where $P' \sqsubseteq_{IAM} P$.

(b) IAM Q.

(c) Parallel compositions $P \parallel Q$ and $P' \parallel Q$, where $P' \parallel Q \not\sqsubseteq_{IAM} P \parallel Q$. Control states are labelled by their concrete data states.

Figure 13: Example IAMs to demonstrate problems with refining input transitions. All IAMs access the global variable $v \in \mathbb{Z}$, which is not owned by any IAMs.

ure 12(b), we obtain their respective parallel compositions, $P \parallel Q$ and $P' \parallel Q$, depicted in Figure 12(c). We see that $P' \parallel Q$ supports *less inputs* than $P \parallel Q$ at state $\langle p_1, q_1 \rangle$. This is problematic because, although $P' \sqsubseteq_{IAM} P$, compositionality is not preserved. Figure 13 demonstrates the compositionality problem analogously for output transitions, where $P' \sqsubseteq_{IAM} P$ but $P' \parallel Q \not\sqsubseteq_{IAM} P \parallel Q$ because $P' \parallel Q$ supports *more outputs* at state $\langle p_1, q_1 \rangle$.

Observe that all the IAMs in Example 14 have transitions that do not have any postconditions, i.e., none of the transitions explicitly modify any shared variables. Instead, when a transition is taken, the source state's data state is restricted by the transition's precondition and this has the side-effect of implicitly modifying (restricting) the target state's data state. We have tried different combinations of weakening and strengthening pre- and postconditions with no success.

4 Retrospective

In pursuit of a stateful variant of IAM, we encountered the following significant challenges to compositionality that we illustrated in Section 3. (1) To support stateful shared variables, it is necessary to track the evolution of data states as a IAM executes so that pre- and postconditions can be evaluated properly. We attempted to encode the data states via an enumeration of control states, but at the risk of state space explosions, and also tried to find a suitable data state abstraction, but at the cost of under- or over-approximating the execution behaviour incorrectly and masking actual communication errors. (2) The incorporation of a shared variable interface and ownership model does not help to decouple data dependencies between components. We saw from Example 13 that a component's data states in isolation do not always correlate with those of a parallel composition it is part of. (3) Parallel composition is not an associative or commutative operation. We can only determine the states with communication errors after all components have been composed together and the system has been closed off from further changes. (4) Any strengthening (or weakening) of a transition's precondition can implicitly strengthen (or weaken) the data states of subsequent transitions. Although components can be refined in isolation, refinement might not hold on their parallel composition with other components.

We conjecture that shared variables create a tight coupling between the control-flows of components. Even with an interface for modifying shared variables (using variable ownership and Allows constraints), the uncertainty that shared variables introduce to data states can only be resolved after the system has been closed off from further modifications and compositions. In summary, compositionality in the sense of IA and IAM is unlikely, and the compositional development of concurrent software systems likely remains an ambition.

4.1 A General Shared Variable Problem?

In our attempts to resolve the problems of stateful IAM, we chose not to employ restrictions that would make the modelling style of IAM awkward and only suitable to very specific systems. We believe we have taken a general approach to the modelling of stateful concurrent software systems: our work is based on IAM, which is an automata-based formalism that shares many similarities to State Machines (SMs) often used in practice [1, 2, 3, 4, 5] to model the operational behaviour concurrent systems. Just as SM [1] supports the use of variables in so-called transition *guards* and *effects*, IAM does as well via pre- and postconditions, which act as contracts [6, 7]. In practical systems, mutual exclusion is used to protect shared variable accesses, and some also incorporate variable ownership [9, 19, 20, 21]. The variable ownership model and Allows constraints that we introduced to IAM aimed to serve a similar purpose but in a more abstract manner.

While the formal defintion of parallel composition for SM and IAM are similar in principle, IAM is more stringent in that it has a notion of communication error, which SM does not. In practical systems, it is typically up to the implementor to decide how run-time errors should be handled, e.g., by verifying inputs before use, or by recovering using *try-catch* statements. In essence, such run-time errors can be prevented by disallowing specific sequences of method calls or service requests at design time; IAM takes this approach by pruning error states from parallel compositions.

In practice, "refinement" is often loosely defined and could mean many things: the implementation of a design specification, the fixing of bugs, or the addition of functionality. It can be argued that IAM's notion of refinement, while mathematically sound, may not be well-suited to practical concurrent software systems. For example, the demand for an implementation to support more inputs, but less outputs, than its specification could be relaxed. However, even if we drop refinement from IAM, the problems of stateful parallel composition would still prevent the incremental development of software.

4.2 Sharing by Copying?

Message passing is a common alternative to shared memory communication, where data values are sent between concurrent software components. Because the values in a message cannot be modified after they have been sent, data dependencies between components are likely to only occur during synchronisation. The authors of [22] extend IA with message passing and show that their formalisation is compositional and, thus, able to support the independent and incremental development of concurrent software systems. Each automaton has a set of *private variables* that no other automaton can observe, and each transition has an action with *data parameters*, a *guard*, and *assignments* to private variables. An automaton has an initial data state that evolves as transitions are taken. A transition can only be taken if its guard evaluates to true under the current data state and data parameters. If the transition is taken, the current data state and data parameters are used in the updating of private variables.

The refinement relation is, however, defined on traces through an implementation and its specification: each trace represents an execution for a specific evolution of data states. To cope with an infinite state space caused by variables with unbounded domains, the data states are computed symbolically using a fixed point algorithm. It is unclear if the algorithm always terminates in the presence of cycles and monotonically increasing data assignments. A similar fixed point computation is also required to prune communication errors from a parallel composition. The advantage of IA's alternating simulation is that it is computed

in a context-free manner on pairs of transitions without needing to consider the execution history of their source states.

5 Conclusions

In this article, we have examined the possibility of developing concurrent software systems in a stateful and compositional manner, in the sense of IA [13] and IAM [15]. While IA-based formalisms for shared memory systems have been proposed, they often have severe restrictions, e.g., variable values do not persist beyond each transition, or variables can only be assigned absolute values, or refinement is not defined. We highlighted four impediments we encountered when trying to extend IAM with stateful variables: (1) the need to track data states, (2) the strong coupling of components caused by shared variables, (3) the inability to detect communication errors in an associative and commutative manner during parallel composition, and (4) a consistent refinement relation could not be found. Our belief is that there is a fundamental problem with shared memory that prevents the compositional development of concurrent software systems.

References

[1] Object Management Group, "Unified Modeling Language, v2.5.1," Dec. 2017. Available at `https://www.omg.org/spec/UML`. Last accessed May 2024.

[2] MathWorks, "Products and Services - MATLAB & Simulink." Available at `https://www.mathworks.com/products`. Last accessed May 2024.

[3] F. X. Dormoy, "SCADE 6 A Model Based Solution For Safety Critical Software Development," in *Embedded Real Time Software and Systems (ERTS)*, Jan. 2008.

[4] H. Elmqvist, F. Gaucher, S. E. Matsson, and F. Dupont, "State Machines in Modelica," in *International MODELICA Conference*, pp. 37–46, Modelica Association and Linköping University Electronic Press, Sept. 2012.

[5] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications: HW/SW-Synthesis for a Conservative Extension of Synchronous Statecharts," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 372–383, ACM, June 2014.

[6] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, pp. 40–51, Oct. 1992.

[7] P. Nienaltowski, B. Meyer, and J. S. Ostroff, "Contracts for Concurrency," *Formal Aspects of Computing*, vol. 21, no. 4, pp. 305–318, 2009.

[8] J.-M. Jézéquel and B. Meyer, "Design by Contract: The Lessons of Ariane," *Computer*, vol. 30, no. 1, pp. 129–130, 1997.

[9] Eiffel, "Separate Calls," 2023. Available at `https://www.eiffel.org/doc/solutions/Separate_Calls`. Last accessed May 2024.

[10] John Barnes Informatics, "Rationale for Ada 2012," 2013. Available at `http://www.ada-auth.org/standards/12rat/html/Rat12-1-3-1.html#I1022`. Last accessed May 2024.

[11] The dafny-lang community, "Dafny Reference Manual, Chapter 7: Specifications," 2024. Available at `https://dafny.org/latest/DafnyRef/DafnyRef#sec-specifications`. Last accessed May 2024.

[12] Kotlin Foundation, "What's new in Kotlin 1.3: Contracts," 2018. Available at `https://kotlinlang.org/docs/whatsnew13.html#contracts`. Last accessed May 2024.

[13] L. de Alfaro and T. A. Henzinger, "Interface-Based Design," in *Engineering Theories of Software Intensive Systems*, vol. 195 of *NATO Science Series*, pp. 83–104, Springer, 2005.

[14] N. A. Lynch and M. R. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," in *6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 137–151, ACM, 1987.

[15] A. Schinko, W. Vogler, J. Gareis, N. T. Nguyen, and G. Lüttgen, "Interface Automata for Shared Memory," *Acta Informatica*, vol. 59, no. 5, pp. 521–556, 2022.

[16] S. S. Bauer, R. Hennicker, and M. Wirsing, "Interface Theories for Concurrency and Data," *Theoretical Computer Science*, vol. 412, no. 28, pp. 3101–3121, 2011.

[17] S. Mouelhi, S. Chouali, and H. Mountassir, "Refinement of Interface Automata Strengthened by Action Semantics," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 1, pp. 111–126, 2009. 6th International Workshop on Formal Engineering approches to Software Components and Architectures (FESCA).

[18] K. J. Goldman and N. A. Lynch, "Modelling Shared State in a Shared Action Model," in *5th Annual IEEE Symposium on Logic in Computer Science*, pp. 450–463, IEEE, 1990.

[19] S. Klabnik and C. Nichols, *The Rust Programming Language, Chapter 4: Understanding Ownership*. No Starch Press, Inc., 2nd ed., 2022.

[20] Standard C++ Foundation, "Standard C++," 2024. Available at `https://isocpp.org`. Last accessed May 2024.

[21] AdaCore and Capgemini Engineering, "SPARK User's Guide, Chapter 5.1.5: Memory Ownership Policy," 2024. Available at `https://docs.adacore.com/spark2014-docs/html/ug/en/source/language_restrictions.html#memory-ownership-policy`. Last accessed May 2024.

[22] L. Holík, M. Isberner, and B. Jonsson, *Mediator Synthesis in a Component Algebra with Data*, pp. 238–259. Springer, 2015.

Bamberger Beiträge zur Wirtschaftsinformatik

Nr. 1 (1989) Augsburger W., Bartmann D., Sinz E.J.: Das Bamberger Modell: Der Diplom-Studiengang Wirtschaftsinformatik an der Universität Bamberg (Nachdruck Dez. 1990)

Nr. 2 (1990) Esswein W.: Definition, Implementierung und Einsatz einer kompatiblen Datenbankschnittstelle für PROLOG

Nr. 3 (1990) Augsburger W., Rieder H., Schwab J.: Endbenutzerorientierte Informationsgewinnung aus numerischen Daten am Beispiel von Unternehmenskennzahlen

Nr. 4 (1990) Ferstl O.K., Sinz E.J.: Objektmodellierung betrieblicher Informationsmodelle im Semantischen Objektmodell (SOM) (Nachdruck Nov. 1990)

Nr. 5 (1990) Ferstl O.K., Sinz E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM)

Nr. 6 (1991) Augsburger W., Rieder H., Schwab J.: Systemtheoretische Repräsentation von Strukturen und Bewertungsfunktionen über zeitabhängigen betrieblichen numerischen Daten

Nr. 7 (1991) Augsburger W., Rieder H., Schwab J.: Wissensbasiertes, inhaltsorientiertes Retrieval statistischer Daten mit EISREVU / Ein Verarbeitungsmodell für eine modulare Bewertung von Kennzahlenwerten für den Endanwender

Nr. 8 (1991) Schwab J.: Ein computergestütztes Modellierungssystem zur Kennzahlenbewertung

Nr. 9 (1992) Gross H.-P.: Eine semantiktreue Transformation vom Entity-Relationship-Modell in das Strukturierte Entity-Relationship-Modell

Nr. 10 (1992) Sinz E.J.: Datenmodellierung im Strukturierten Entity-Relationship-Modell (SERM)

Nr. 11 (1992) Ferstl O.K., Sinz E. J.: Glossar zum Begriffsystem des Semantischen Objektmodells

Nr. 12 (1992) Sinz E. J., Popp K.M.: Zur Ableitung der Grobstruktur des konzeptuellen Schemas aus dem Modell der betrieblichen Diskurswelt

Nr. 13 (1992) Esswein W., Locarek H.: Objektorientierte Programmierung mit dem Objekt-Rollenmodell

Nr. 14 (1992) Esswein W.: Das Rollenmodell der Organsiation: Die Berücksichtigung aufbauorganisatorische Regelungen in Unternehmensmodellen

Nr. 15 (1992) Schwab H. J.: EISREVU-Modellierungssystem. Benutzerhandbuch

Nr. 16 (1992) Schwab K.: Die Implementierung eines relationalen DBMS nach dem Client/Server-Prinzip

Nr. 17 (1993) Schwab K.: Konzeption, Entwicklung und Implementierung eines computergestützten Bürovorgangssystems zur Modellierung von Vorgangsklassen und Abwicklung und Überwachung von Vorgängen. Dissertation

Nr. 18 (1993) Ferstl O.K., Sinz E.J.: Der Modellierungsansatz des Semantischen Objektmodells

Nr. 19 (1994) Ferstl O.K., Sinz E.J., Amberg M., Hagemann U., Malischewski C.: Tool-Based Business Process Modeling Using the SOM Approach

Nr. 20 (1994) Ferstl O.K., Sinz E.J.: From Business Process Modeling to the Specification of Distributed Business Application Systems - An Object-Oriented Approach -. 1st edition, June 1994

Ferstl O.K., Sinz E.J. : Multi-Layered Development of Business Process Models and Distributed Business Application Systems - An Object-Oriented Approach - . 2nd edition, November 1994

Nr. 21 (1994) Ferstl O.K., Sinz E.J.: Der Ansatz des Semantischen Objektmodells zur Modellierung von Geschäftsprozessen

Nr. 22 (1994) Augsburger W., Schwab K.: Using Formalism and Semi-Formal Constructs for Modeling Information Systems

Nr. 23 (1994) Ferstl O.K., Hagemann U.: Simulation hierarischer objekt- und transaktionsorientierter Modelle

Nr. 24 (1994) Sinz E.J.: Das Informationssystem der Universität als Instrument zur zielgerichteten Lenkung von Universitätsprozessen

Nr. 25 (1994) Wittke M., Mekinic, G.: Kooperierende Informationsräume. Ein Ansatz für verteilte Führungsinformationssysteme

Nr. 26 (1995) Ferstl O.K., Sinz E.J.: Re-Engineering von Geschäftsprozessen auf der Grundlage des SOM-Ansatzes

Nr. 27 (1995) Ferstl, O.K., Mannmeusel, Th.: Dezentrale Produktionslenkung. Erscheint in CIM-Management 3/1995

Nr. 28 (1995) Ludwig, H., Schwab, K.: Integrating cooperation systems: an event-based approach

Nr. 30 (1995) Augsburger W., Ludwig H., Schwab K.: Koordinationsmethoden und -werkzeuge bei der computergestützten kooperativen Arbeit

Nr. 31 (1995) Ferstl O.K., Mannmeusel T.: Gestaltung industrieller Geschäftsprozesse

Nr. 32 (1995) Gunzenhäuser R., Duske A., Ferstl O.K., Ludwig H., Mekinic G., Rieder H., Schwab H.-J., Schwab K., Sinz E.J., Wittke M: Festschrift zum 60. Geburtstag von Walter Augsburger

Nr. 33 (1995) Sinz, E.J.: Kann das Geschäftsprozeßmodell der Unternehmung das unternehmensweite Datenschema ablösen?

Nr. 34 (1995) Sinz E.J.: Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme - Entwicklung, aktueller Stand und Trends -

Nr. 35 (1995) Sinz E.J.: Serviceorientierung der Hochschulverwaltung und ihre Unterstützung durch workflow-orientierte Anwendungssysteme

Nr. 36 (1996) Ferstl O.K., Sinz, E.J., Amberg M.: Stichwörter zum Fachgebiet Wirtschaftsinformatik. Erscheint in: Broy M., Spaniol O. (Hrsg.): Lexikon Informatik und Kommunikationstechnik, 2. Auflage, VDI-Verlag, Düsseldorf 1996

Nr. 37 (1996) Ferstl O.K., Sinz E.J.: Flexible Organizations Through Object-oriented and Transaction-oriented Information Systems, July 1996

Nr. 38 (1996) Ferstl O.K., Schäfer R.: Eine Lernumgebung für die betriebliche Aus- und Weiterbildung on demand, Juli 1996

Nr. 39 (1996) Hazebrouck J.-P.: Einsatzpotentiale von Fuzzy-Logic im Strategischen Management dargestellt an Fuzzy-System-Konzepten für Portfolio-Ansätze

Nr. 40 (1997) Sinz E.J.: Architektur betrieblicher Informationssysteme. In: Rechenberg P., Pomberger G. (Hrsg.): Handbuch der Informatik, Hanser-Verlag, München 1997

Nr. 41 (1997) Sinz E.J.: Analyse und Gestaltung universitärer Geschäftsprozesse und Anwendungssysteme. Angenommen für: Informatik '97. Informatik als Innovationsmotor. 27. Jahrestagung der Gesellschaft für Informatik, Aachen 24.-26.9.1997

Nr. 42 (1997) Ferstl O.K., Sinz E.J., Hammel C., Schlitt M., Wolf S.: Application Objects – fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme. Angenommen für: HMD – Theorie und Praxis der Wirtschaftsinformatik. Schwerpunkheft ComponentWare, 1997

Nr. 43 (1997): Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using the Semantic Object Model (SOM) – A Methodological Framework - . Accepted for: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1997

Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using (SOM), 2nd Edition. Appears in: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1998

Nr. 44 (1997) Ferstl O.K., Schmitz K.: Zur Nutzung von Hypertextkonzepten in Lernumgebungen. In: Conradi H., Kreutz R., Spitzer K. (Hrsg.): CBT in der Medizin – Methoden, Techniken, Anwendungen -. Proceedings zum Workshop in Aachen 6. – 7. Juni 1997. 1. Auflage Aachen: Verlag der Augustinus Buchhandlung

Nr. 45 (1998) Ferstl O.K.: Datenkommunikation. In. Schulte Ch. (Hrsg.): Lexikon der Logistik, Oldenbourg-Verlag, München 1998

Nr. 46 (1998) Sinz E.J.: Prozeßgestaltung und Prozeßunterstützung im Prüfungswesen. Erschienen in: Proceedings Workshop „Informationssysteme für das Hochschulmanagement". Aachen, September 1997

Nr. 47 (1998) Sinz, E.J.:, Wismans B.: Das „Elektronische Prüfungsamt". Erscheint in: Wirtschaftswissenschaftliches Studium WiSt, 1998

Nr. 48 (1998) Haase, O., Henrich, A.: A Hybrid Respresentation of Vague Collections for Distributed Object Management Systems. Erscheint in: IEEE Transactions on Knowledge and Data Engineering

Nr. 49 (1998) Henrich, A.: Applying Document Retrieval Techniques in Software Engineering Environments. In: Proc. International Conference on Database and Expert Systems Applications. (DEXA 98), Vienna, Austria, Aug. 98, pp. 240-249, Springer, Lecture Notes in Computer Sciences, No. 1460

Nr. 50 (1999) Henrich, A., Jamin, S.: On the Optimization of Queries containing Regular Path Expressions. Erscheint in: Proceedings of the Fourth Workshop on Next Generation Information Technologies and Systems (NGITS'99), Zikhron-Yaakov, Israel, July, 1999 (Springer, Lecture Notes)

Nr. 51 (1999) Haase O., Henrich, A.: A Closed Approach to Vague Collections in Partly Inaccessible Distributed Databases. Erscheint in: Proceedings of the Third East-European Conference on Advances in Databases and Information Systems – ADBIS'99, Maribor, Slovenia, September 1999 (Springer, Lecture Notes in Computer Science)

Nr. 52 (1999) Sinz E.J., Böhnlein M., Ulbrich-vom Ende A.: Konzeption eines Data Warehouse-Systems für Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule" im Rahmen der 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 6. Oktober 1999

Nr. 53 (1999) Sinz E.J.: Konstruktion von Informationssystemen. Der Beitrag wurde in geringfügig modifizierter Fassung angenommen für: Rechenberg P., Pomberger G. (Hrsg.): Informatik-Handbuch. 2., aktualisierte und erweiterte Auflage, Hanser, München 1999

Nr. 54 (1999) Herda N., Janson A., Reif M., Schindler T., Augsburger W.: Entwicklung des Intranets SPICE: Erfahrungsbericht einer Praxiskooperation.

Nr. 55 (2000) Böhnlein M., Ulbrich-vom Ende A.: Grundlagen des Data Warehousing. Modellierung und Architektur

Nr. 56 (2000) Freitag B, Sinz E.J., Wismans B.: Die informationstechnische Infrastruktur der Virtuellen Hochschule Bayern (vhb). Angenommen für Workshop "Unternehmen Hochschule 2000" im Rahmen der Jahrestagung der Gesellschaft f. Informatik, Berlin 19. - 22. September 2000

Nr. 57 (2000) Böhnlein M., Ulbrich-vom Ende A.: Developing Data Warehouse Structures from Business Process Models.

Nr. 58 (2000) Knobloch B.: Der Data-Mining-Ansatz zur Analyse betriebswirtschaftlicher Daten.

Nr. 59 (2001) Sinz E.J., Böhnlein M., Plaha M., Ulbrich-vom Ende A.: Architekturkonzept eines verteilten Data-Warehouse-Systems für das Hochschulwesen. Angenommen für: WI-IF 2001, Augsburg, 19.-21. September 2001

Nr. 60 (2001) Sinz E.J., Wismans B.: Anforderungen an die IV-Infrastruktur von Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule 2001" im Rahmen der Jahrestagung der Gesellschaft für Informatik, Wien 25. – 28. September 2001

Änderung des Titels der Schriftenreihe *Bamberger Beiträge zur Wirtschaftsinformatik* in *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* ab Nr. 61

Note: The title of our technical report series has been changed from *Bamberger Beiträge zur Wirtschaftsinformatik* to *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* starting with TR No. 61

Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik

Nr. 61 (2002) Goré R., Mendler M., de Paiva V. (Hrsg.): Proceedings of the International Workshop on Intuitionistic Modal Logic and Applications (IMLA 2002), Copenhagen, July 2002.

Nr. 62 (2002) Sinz E.J., Plaha M., Ulbrich-vom Ende A.: Datenschutz und Datensicherheit in einem landesweiten Data-Warehouse-System für das Hochschulwesen. Erscheint in: Beiträge zur Hochschulforschung, Heft 4-2002, Bayerisches Staatsinstitut für Hochschulforschung und Hochschulplanung, München 2002

Nr. 63 (2005) Aguado, J., Mendler, M.: Constructive Semantics for Instantaneous Reactions

Nr. 64 (2005) Ferstl, O.K.: Lebenslanges Lernen und virtuelle Lehre: globale und lokale Verbesserungspotenziale. Erschienen in: Kerres, Michael; Keil-Slawik, Reinhard (Hrsg.); Hochschulen im digitalen Zeitalter: Innovationspotenziale und Strukturwandel, S. 247 – 263; Reihe education quality forum, herausgegeben durch das Centrum für eCompetence in Hochschulen NRW, Band 2, Münster/New York/München/Berlin: Waxmann 2005

Nr. 65 (2006) Schönberger, Andreas: Modelling and Validating Business Collaborations: A Case Study on RosettaNet

Nr. 66 (2006) Markus Dorsch, Martin Grote, Knut Hildebrandt, Maximilian Röglinger, Matthias Sehr, Christian Wilms, Karsten Loesing, and Guido Wirtz: Concealing Presence Information in Instant Messaging Systems, April 2006

Nr. 67 (2006) Marco Fischer, Andreas Grünert, Sebastian Hudert, Stefan König, Kira Lenskaya, Gregor Scheithauer, Sven Kaffille, and Guido Wirtz: Decentralized Reputation Management for Cooperating Software Agents in Open Multi-Agent Systems, April 2006

Nr. 68 (2006) Michael Mendler, Thomas R. Shiple, Gérard Berry: Constructive Circuits and the Exactness of Ternary Simulation

Nr. 69 (2007) Sebastian Hudert: A Proposal for a Web Services Agreement Negotiation Protocol Framework . February 2007

Nr. 70 (2007) Thomas Meins: Integration eines allgemeinen Service-Centers für PC-und Medientechnik an der Universität Bamberg – Analyse und Realisierungs-Szenarien. February 2007 (out of print)

Nr. 71 (2007) Andreas Grünert: Life-cycle assistance capabilities of cooperating Software Agents for Virtual Enterprises. März 2007

Nr. 72 (2007) Michael Mendler, Gerald Lüttgen: Is Observational Congruence on μ-Expressions Axiomatisable in Equational Horn Logic?

Nr. 73 (2007) Martin Schissler: out of print

Nr. 74 (2007) Sven Kaffille, Karsten Loesing: Open chord version 1.0.4 User's Manual. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 74, Bamberg University, October 2007. ISSN 0937-3349.

Nr. 75 (2008) Karsten Loesing (Hrsg.): Extended Abstracts of the Second *Privacy Enhancing Technologies Convention* (PET-CON 2008.1). Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 75, Bamberg University, April 2008. ISSN 0937-3349.

Nr. 76 (2008) Gregor Scheithauer, Guido Wirtz: Applying Business Process Management Systems – A Case Study. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 76, Bamberg University, May 2008. ISSN 0937-3349.

Nr. 77 (2008) Michael Mendler, Stephan Scheele: Towards Constructive Description Logics for Abstraction and Refinement. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 77, Bamberg University, September 2008. ISSN 0937-3349.

Nr. 78 (2008) Gregor Scheithauer, Matthias Winkler: A Service Description Framework for Service Ecosystems. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 78, Bamberg University, October 2008. ISSN 0937-3349.

Nr. 79 (2008) Christian Wilms: Improving the Tor Hidden Service Protocol Aiming at Better Performances. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 79, Bamberg University, November 2008. ISSN 0937-3349.

Nr. 80 (2009) Thomas Benker, Stefan Fritzemeier, Matthias Geiger, Simon Harrer, Tristan Kessner, Johannes Schwalb, Andreas Schönberger, Guido Wirtz: QoS Enabled B2B Integration. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 80, Bamberg University, May 2009. ISSN 0937-3349.

Nr. 81 (2009) Ute Schmid, Emanuel Kitzelmann, Rinus Plasmeijer (Eds.): Proceedings of the ACM SIGPLAN Workshop on *Approaches and Applications of Inductive Programming* (AAIP'09), affiliated with ICFP 2009, Edinburgh, Scotland, September 2009. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 81, Bamberg University, September 2009. ISSN 0937-3349.

Nr. 82 (2009) Ute Schmid, Marco Ragni, Markus Knauff (Eds.): Proceedings of the KI 2009 Workshop *Complex Cognition*, Paderborn, Germany, September 15, 2009. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 82, Bamberg University, October 2009. ISSN 0937-3349.

Nr. 83 (2009) Andreas Schönberger, Christian Wilms and Guido Wirtz: A Requirements Analysis of Business-to-Business Integration. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 83, Bamberg University, December 2009. ISSN 0937-3349.

Nr. 84 (2010) Werner Zirkel, Guido Wirtz: A Process for Identifying Predictive Correlation Patterns in Service Management Systems. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 84, Bamberg University, February 2010. ISSN 0937-3349.

Nr. 85 (2010) Jan Tobias Mühlberg und Gerald Lüttgen: Symbolic Object Code Analysis. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 85, Bamberg University, February 2010. ISSN 0937-3349.

Nr. 86 (2010) Werner Zirkel, Guido Wirtz: Proaktives Problem Management durch Eventkorrelation – ein *Best Practice* Ansatz. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 86, Bamberg University, August 2010. ISSN 0937-3349.

Nr. 87 (2010) Johannes Schwalb, Andreas Schönberger: Analyzing the Interoperability of WS-Security and WS-ReliableMessaging Implementations. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 87, Bamberg University, September 2010. ISSN 0937-3349.

Nr. 88 (2011) Jörg Lenhard: A Pattern-based Analysis of WS-BPEL and Windows Workflow. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 88, Bamberg University, March 2011. ISSN 0937-3349.

Nr. 89 (2011) Andreas Henrich, Christoph Schlieder, Ute Schmid [eds.]: Visibility in Information Spaces and in Geographic Environments – Post-Proceedings of the KI'11 Workshop. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 89, Bamberg University, December 2011. ISSN 0937-3349.

Nr. 90 (2012) Simon Harrer, Jörg Lenhard: Betsy - A BPEL Engine Test System. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 90, Bamberg University, July 2012. ISSN 0937-3349.

Nr. 91 (2013) Michael Mendler, Stephan Scheele: On the Computational Interpretation of CKn for Contextual Information Processing - Ancillary Material. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 91, Bamberg University, May 2013. ISSN 0937-3349.

Nr. 92 (2013) Matthias Geiger: BPMN 2.0 Process Model Serialization Constraints. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 92, Bamberg University, May 2013. ISSN 0937-3349.

Nr. 93 (2014) Cedric Röck, Simon Harrer: Literature Survey of Performance Benchmarking Approaches of BPEL Engines. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 93, Bamberg University, May 2014. ISSN 0937-3349.

Nr. 94 (2014) Joaquin Aguado, Michael Mendler, Reinhard von Hanxleden, Insa Fuhrmann: Grounding Synchronous Deterministic Concurrency in Sequential Programming. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 94, Bamberg University, August 2014. ISSN 0937-3349.

Nr. 95 (2014) Michael Mendler, Bruno Bodin, Partha S Roop, Jia Jie Wang: WCRT for Synchronous Programs: Studying the Tick Alignment Problem. Bamberger

Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 95, Bamberg University, August 2014. ISSN 0937-3349.

Nr. 96 (2015) Joaquin Aguado, Michael Mendler, Reinhard von Hanxleden, Insa Fuhrmann: Denotational Fixed-Point Semantics for Constructive Scheduling of Synchronous Concurrency. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 96, Bamberg University, April 2015. ISSN 0937-3349.

Nr. 97 (2015) Thomas Benker: Konzeption einer Komponentenarchitektur für prozessorientierte OLTP- & OLAP-Anwendungssysteme. Bamberger Beiträge zur Wirtschafts-informatik und Angewandten Informatik Nr. 97, Bamberg University, Oktober 2015. ISSN 0937-3349.

Nr. 98 (2016) Sascha Fendrich, Gerald Lüttgen: A Generalised Theory of Interface Automata, Component Compatibility and Error. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 98, Bamberg University, March 2016. ISSN 0937-3349.

Nr. 99 (2014) Christian Preißinger, Simon Harrer: Static Analysis Rules of the BPEL Specification: Tagging, Formalization and Tests. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 99, Bamberg University, August 2014. ISSN 0937-3349.

Nr. 100 (2016) Cedrik Röck, Stefan Kolb: Nucleus - Unified Deployment and Management for Platform as a Service. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 100, Bamberg University, March 2016. ISSN 0937-3349.

Nr. 101 (2016) Michael Mendler, Partha S. Roop, Bruno Bodin: A Novel WCET Semantics of Synchronous Programs. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 101, Bamberg University, June 2016. ISSN 0937-3349.

Nr. 102 (2017) Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha Roop, Reinhard von Hanxleden: Clock-Synchronised Shared Objects for Deterministic Concurrency. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 102, Bamberg University, July 2017. ISSN 0937-3349.

Nr. 103 (2018) Eugene Yip, Erjola Lalo, Gerald Lüttgen, Michael Deubzer, Andreas Sailer: Optimized Buffering of Time-Triggered Automotive Software. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 103, Bamberg University, September 2018. ISSN 0937-3349.

Nr. 104 (2018) Daniel Hallmann: Die COCOMO-Modelle im Licht der agilen Softwareentwicklung. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 104, Bamberg University, October 2018. ISSN 0937-3349.

Nr. 105 (2021) Johannes Manner: SeMoDe – Simulation and Benchmarking Pipeline for Function as a Service. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 105, Bamberg University, October 2021. ISSN 0937-3349.

Nr. 106 (2024) Eugene Yip, Gerald Lüttgen: Heterogeneous Specification of Spacecraft Software. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 106, Bamberg University, August 2024. ISSN 0937-3349.

Nr. 107 (2024) Eugene Yip, Gerald Lüttgen: Concurrency, Shared Variables, Compositionality: An Unlikely Triple. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 107, Bamberg University, August 2024. ISSN 0937-3349.