View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science 159 (2006) 79–97

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Interface Automata with Complex Actions

Shahram Esmaeilsabzali [1] Farhad Mavaddat [2]
Nancy A. Day [3]

School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Abstract

Many formalisms use interleaving to model concurrency. To describe some system behaviours appropriately, we need to limit interleaving. For example, in component-based systems, we wish to limit interleaving to force the inputs to a method to arrive together in order. We introduce *interface automata with complex actions* (IACA), which add complex actions to de Alfaro and Henzinger's interface automata (IA). A complex action is a sequence of actions that may not be interleaved with actions from other components. The composition and refinement operations are more involved in IACA compared to IA, and we must sacrifice associativity of composition. However, we argue that the advantages of having complex actions make it a useful formalism.

Keywords: Component-based design, Service-oriented design, interleaving, complex actions.

1 Introduction

Interleaving is a common choice to model the concurrent behaviour between components of a system. Interleaving means that at each point in time only one component takes a step. The result is all possible interleavings of the actions of the components. Many formalisms, both algebraic and non-algebraic, have adopted interleaving semantics, *e.g.*, [13,12,11,10]. However, interleaving is not always appropriate to characterize system behaviour accurately because

[1] Email: sesmaeil@cs.uwaterloo.ca
[2] Email: fmavaddat@cs.uwaterloo.ca
[3] Email: nday@cs.uwaterloo.ca

1571-0661 © 2006 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2005.12.063

it is based on the assumption that the concurrent behaviours of a system consist of all possible orderings of actions in a system. Some software artifacts have multiple constituent elements but represent a single software artifact, thus, we may wish to group multiple actions such that their behaviour cannot be interleaved with the behaviour of another component.

In this work, we introduce *interface automata with complex actions* (IACA), designed to model component-based/service-oriented systems. IACA uses interleaving with complex actions as its semantics for concurrency. A *complex action* consists of multiple simple actions that cannot be interleaved with the behaviour of another component. In component-based systems, at its signature level a method of a component can be characterized by the method's name and a set of parameters. Some formalisms choose to model methods by abstracting away their details using, for example, only its name, *e.g.,* [4]. To model the details of the parameter communication, the arrival of the inputs should not be interleaved with the behaviour of another component. Thus, we require complex actions to model the semantics of the concurrent behaviour of component-based systems at this level of detail. In Web services, communication with service requesters and other Web services occurs through complex XML messages, which are streams of data delimited appropriately into multiple simple messages. We would like to model composite XML messages of Web services as non-interruptible software artifacts.

Various approaches have been proposed for grouping multiple actions together ("atomic actions")[4] [3,9] or refining a single action into multiple actions ("action refinement") [1,14]. Most of these approaches are proposed in a process algebraic context. We are interested in defining an automata-based model with complex actions, which complies as closely as possible with the class of *interface models*, introduced by de Alfaro and Henzinger [4]. These models assume a helpful environment, which supplies needed inputs and receives all outputs. They also have well-formedness criteria that support *top-down design*, which means that a refinement of a component can be substituted for the original in the context of its composition with other components. Composition must be commutative and associative. *Interface automata (IA)* are an interface model used to model the behaviour of component-based systems [4]. Composition for IA uses interleaving semantics.

Our model, *interface automata with complex actions (IACA)*, extends IA with the complex actions needed to model methods of component-based systems and messages of Web services. The main challenge for IACA is how to define a composition operator and a refinement relation that do not al-

[4] To avoid ambiguity, for actions that consist of multiple simple actions, we choose the name "complex actions" instead of "atomic actions".

low interleaving of complex actions, but support top-down design as much as possible. Compared to other formalisms with complex actions, in IACA we must respect the helpful environment, which means that in composition a state should not be reached where one component would wait for communication from the other. Additionally, IACA composition requires that two IACAs with compatible complex actions synchronize with each other. Our definition of composition leads to a natural definition for IACA refinement that is comparable with programming language concepts such as subclasses, and optional parameters. IACA satisfies all of the well-formedness criteria of interface models except the associativity of the composition operator. Other approaches that have used types of interleaving with complex actions have also suffered from the loss of associativity, *e.g.,* A^2CCS [9]. Despite the non-associativity of composition, IACA is useful for modelling software artifacts with complex actions.

We begin by providing background on interface automata. Next, we describe IACA and its composition operator and refinement relation. In Section 4, we compare IACA with similar models that support complex actions, and summarize and discuss future work in Section 5.

2 Background: Interface Automata

An interface automaton (IA) [4,6], introduced by de Alfaro and Henzinger, is an automata-based model suitable for specifying component-based systems. IA is part of a class of models called *interface models* [5], which are intended to specify concisely *how* systems can be used and to adhere to certain well-formedness criteria that make them appropriate for modelling component-based systems. The two main characteristics of interface models are that they assume a *helpful environment* and support *top-down design.* A *helpful environment* for an interface provides the inputs it needs and always accepts all its outputs. Therefore, interfaces are optimistic, and do not usually specify all possible behaviours of the systems. For example, they often do not include fault scenarios. *Top-down design* is based on a notion of refinement, which relates two instances of a model. A refinement of a model can be substituted for the original. In a well-formed interface model, the binary operator *composition* and a *refinement* relation are defined. Composition is both commutative and associative. Top-down design means that for three interface models P, P', Q, and the composition of P and Q, $P \parallel Q$, if P' refines P, *i.e.,* $P' \preceq P$, then: $(P' \parallel Q) \preceq (P \parallel Q)$.

Interface automata are interface models. They are syntactically similar to *Input/Output Automata* [11], but have different semantics. Figure 1 shows

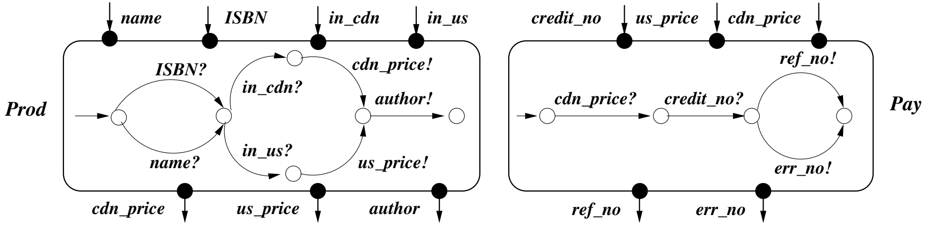

Fig. 1. Two IAs: *Prod* and *Pay*.

two IAs. The arrows on top represent the inputs of the system and arrows at the bottom represent the outputs of the system. The initial state of the IA is designated by an arrow with no source. IA *Prod* is a component (service) that receives either an *ISBN* or a *name* of a book, and based on the request provides the price of the book in Canadian or US dollars. The author of book is also provided as an output of the system. Input actions are followed by "?" and outputs by "!". IA *Pay* carries out a credit card payment by receiving an amount in Canadian dollars and a credit card number, and produces either a reference number for a successful transaction or an error number. IAs assume helpful environments; *Pay*, for example, assumes that the environment first provides input value *cdn_price* and then *credit_no*.

Definition 2.1 An interface automaton (IA) $P = \langle V_P, i_P, \mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H, \tau_P \rangle$ consists of V_P a finite set of states, $i_P \in V_P$ the initial state, $\mathcal{A}_P^I$, $\mathcal{A}_P^O$ and $\mathcal{A}_P^H$, which are disjoint sets of input, output, and hidden actions, respectively, and τ_P the set of transitions between states such that $\tau_P \subseteq V_P \times \mathcal{A}_P \times V_P$, where $\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O \cup \mathcal{A}_P^H$. □

Well-formed IAs are required to be deterministic on inputs [6]. In other words, for any two input transitions (u, a, v) and (u, a, l), $v = l$.

The composition of two IAs consists of all possible interleaved transitions of the two IAs, except for those actions that are shared. Two IAs are composable if they do not take any of the same inputs, do not produce any of the same outputs and the hidden actions of the two components do not overlap. A hidden action is created through the composition of IA when an output action of one component is internally consumed by an input action of another component. This synchronization reduces the two actions to a hidden action on a single transition.

Because of the assumption of a helpful environment, neither component should have to wait to synchronize, *i.e.,* if one component is ready to send an action, the other should be ready to receive the action immediately. A state of the product where one component would have to wait is considered an *illegal state* and is eliminated (with transitions leading to it) from the composed IA.

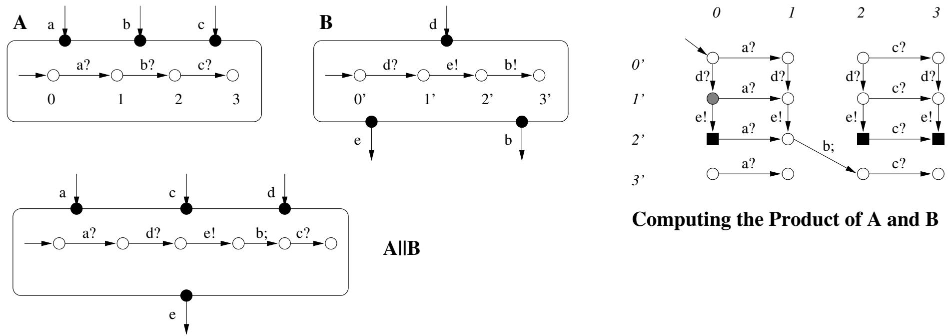

Fig. 2. Two composable IAs A and B and their composition $A \parallel B$.

In other words, the resulting composition will consist of only those states that can satisfy environmental assumptions of both IAs.

As a simple example consider the IAs in Figure 2. IAs A and B are composable and b is a shared action of the two IAs. The composition of A and B is computed by considering the product of their states shown at the right of the figure. Transitions on non-shared events are interleaved, and a transition is created on a hidden event to represent the synchronization on b (from state (1,2') to state (2,3')). Hidden events have ";" following their names. In state (0,2'), IA B is immediately ready to send b but IA A is not yet ready to receive it. State (0,2') is an illegal state, as are (2,2') and (3,2') shown in black boxes. These states are not included in the composition. States and transitions on paths that lead to these illegal states where the path consists *entirely* of output and hidden actions are also not included. State (0,1') (shown with filled in circle) is enabled with "*e*!" and as such can lead to an illegal state. Thus we consider that state itself an illegal state. In the presence of a helpful environment, execution could lead to an illegal state from that state since the environment does not have any control over output and hidden transitions. Non-reachable states are also eliminated. The IA resulting from the composition of A and B is labelled $A \parallel B$ in Figure 2.

The composition of the IAs *Prod* and *Pay* in Figure 1 is shown in Figure 3. All states where *Prod* generates the output "*us_price*" and *Pay* is not ready to receive it are considered illegal states and are not included in the composition. In this example, by removing such illegal states, transitions on "*in_us*?" are removed; however, "*in_us*" still appears as an input of the composition.

IA Q refines IA P if Q provides the services of P; it can have more inputs but no more output actions. As such, a refinement of an IA does not constrain the environment more than the original IA does. As an example, *GenPay* in Figure 4 refines *Pay* in Figure 1; *GenPay* provides more services than *Pay* since it can carry out payments in both Canadian and US dollars. Top-down

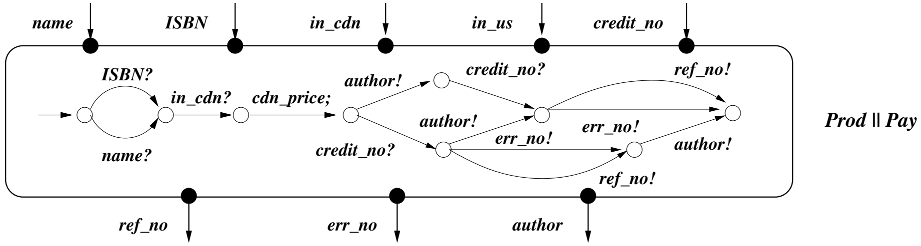

Fig. 3. *Prod* ‖ *Pay* is the composition of two composable IAs in Figure 1.

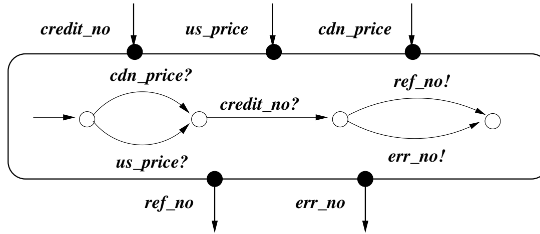

Fig. 4. *GenPay* refines *Pay* in Figure 1.

design then guarantees that $(Prod \parallel GenPay)$ refines $(Prod \parallel Pay)$.

Refinement of IA is defined using a refinement relation between the states of two IAs. If IA Q refines IA P, stated as $Q \preceq P$, then an *alternating simulation relation* [2] exists between the states of Q and P. For $q \in V_Q$ and $p \in V_P$, $q \preceq p$ if q has more than or the same input actions as p, and less than or the same output actions as p. Also, for any state q' reachable from q, immediately or through hidden actions, there is a corresponding state p' similarly reachable from p such that $q' \preceq p'$. All states reachable from a state *only* through hidden actions are considered the same state for the purposes of refinement. The initial state of Q must refine the initial state of P.

3 Interface Automata with Complex Actions

Interface automata with complex actions extend interface automata with the ability to declare a sequence of transitions to be a *complex action*, which cannot be interleaved with transitions from another component in composition. Complex actions in IACA are meant to model software artifacts, such as methods or complex messages, which can have multiple constituent elements but should not be interleaved with other actions in composition. As an example, Figure 5 shows an IACA, *CompPay*, with a complex action *pay_in_cdn*, represented by the dashed transition. This automaton is similar to IA *Pay* of Figure 1 except that the actions *cdn_price?* and *credit_no?* cannot be interleaved with actions from another component in composition. In fact,

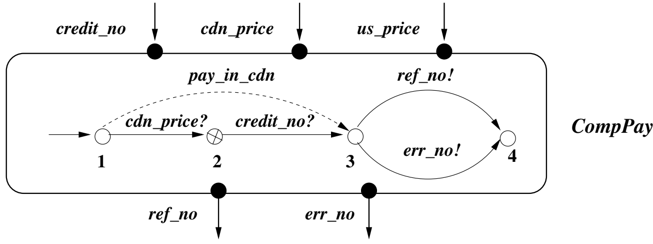

Fig. 5. IACA *CompPay* represents similar functionality as IA *Pay* in Figure 1.

pay_in_cdn models the environment assumption that the values for *cdn_price?* and *credit_no?* arrive sequentially.

Complex actions represent either input or output behaviours, and thus should consist entirely of either input or output actions, possibly along with some hidden actions. Complex actions can only be a linear sequence of transitions. The states within a complex action are called *internal states* and are represented by circles with an "x" in them. In Figure 5, states 1, 3, and 4 are normal states of *CompPay* and state 2 is the only internal state.

First, we introduce IACA formally and then define its composition operator and refinement relation. In this paper, because of space limitations, we omit some of the formalism and proofs of our claims; the reader can refer to [8] for the complete details.

Definition 3.1 An interface automaton with complex actions (IACA) $P = \langle V_P^N, V_P^{int}, i_P, \mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H, \mathcal{A}_P^C, \tau_P, \phi_P \rangle$ has the following elements:

- V_P^N is the set of normal states.
- V_P^{Int} is the set of internal states. $V_P^N \cap V_P^{Int} = \emptyset$. The internal states are the ones inside a complex action. We denote $V_P = V_P^N \cup V_P^{Int}$ as the set of all states.
- i_P is the initial state. $i_P \in V_P^N$.
- $\mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H$ are disjoint sets of input, output and hidden actions. These are normal, non-complex actions. We denote $\mathcal{A}_P^N = \mathcal{A}_P^I \cup \mathcal{A}_P^O \cup \mathcal{A}_P^H$.
- $\mathcal{A}_P^C$ is the set of complex actions where $\mathcal{A}_P^C \cap \mathcal{A}_P^N = \emptyset$. We let $\mathcal{A}_P = \mathcal{A}_P^C \cup \mathcal{A}_P^N$.
- $\tau_P \subseteq V_P \times \mathcal{A}_P^N \times V_P$ is the set of normal (non-complex) transitions. We require that each $v \in V_P^{Int}$ is the source of *exactly* one transition and the destination of *exactly* one transition in τ_P. Furthermore, IACA is input deterministic, *i.e.*,
  $\forall (u, a, v) \in \tau_P, (u, a, v') \in \tau_P \cdot a \in \mathcal{A}_P^I \Rightarrow (v = v')$
- $\phi_P \subseteq V_P^N \times \mathcal{A}_P^C \times V_P^N$ is the set of complex transitions. Every $(u, c, v) \in \phi_P$ is associated with a sequence of non-complex transitions in τ_p called a *complex*

fragment. A complex fragment is defined as an alternating sequence of states and normal actions:

$frag(u, c, v) = \langle u, a_0, s_0, a_1, s_1, \ldots, s_{n-1}, a_n, v\rangle$ where

· $\forall i \cdot s_i \in V_P^{Int}$ (all s_i's are internal states), and

· $(\forall i \cdot a_i \in \mathcal{A}_P^I \cup \mathcal{A}_P^H) \vee (\forall i \cdot a_i \in \mathcal{A}_P^O \cup \mathcal{A}_P^H)$ (all actions either belong to union of input and hidden actions, or belong to union of output and hidden actions), and

· $((u, a_0, s_0) \in \tau_P) \wedge ((s_{n-1}, a_n, v) \in \tau_P) \wedge \forall i(0 < i < n) \cdot (s_{i-1}, a_i, s_i) \in \tau_P$ (every step is a non-complex transition).

· Considering the sequence of actions in a complex fragment,$act(u, c, v)$, where $act(u, c, v) = \langle a_0, a_1, \ldots, a_n\rangle$, the following condition must hold: $(\forall(u, c, v) \in \phi_P \wedge \forall(u', c, v') \in \phi_P \cdot act(u, c, v) = act(u', c, v')) \wedge$ $(\forall(u, c, v) \in \phi_P \wedge \forall(u', d, v') \in \phi_P \cdot act(u, c, v) = act(u', d, v') \Rightarrow d = c)$ (Complex transitions with same complex actions have same sequence of actions in their fragments, and complex actions with similar sequence of actions have the same complex action names.) □

The constraints on definition of ϕ_P guarantee complex transitions are associated with unique sequence of actions.

Every IA is an IACA with empty sets of complex transitions and complex actions. We call the IA that consists of all parts of an IACA except the complex transitions and the complex actions, the *equivalent IA* to an IACA.

3.1 Composition

IACA composition is a binary function mapping two composable IACAs into a new IACA. The main difference between IACA and IA composition is that transitions within a complex action are not interleaved in IACA composition. This behaviour is necessary to ensure all parameters of a method call or a message arrive together in the exact order required. Synchronization between actions of the two components may occur within a complex fragment, but each complex fragment in the two IACAs being composed maintains its sequence of actions in the composition (possibly with some actions having become hidden actions). Furthermore, either the whole complex fragment is present in the result or the complex fragment should not appear in the result at all. The IACA composition of two composable IACAs is a subset of the IA composition of the equivalent IAs for those two IACAs.

Figure 6 shows the composition of *CompPay*, in Figure 5, and component *Prod* (now viewed as an IACA), in Figure 1. The transitions within the complex transition *pay_in_cdn* in *CompPay* are not interleaved with other actions and *pay_in_cdn* remains a complex action in $Prod \parallel CompPay$. The composition did involve a synchronization between the input *cdn_price* in *CompPay*

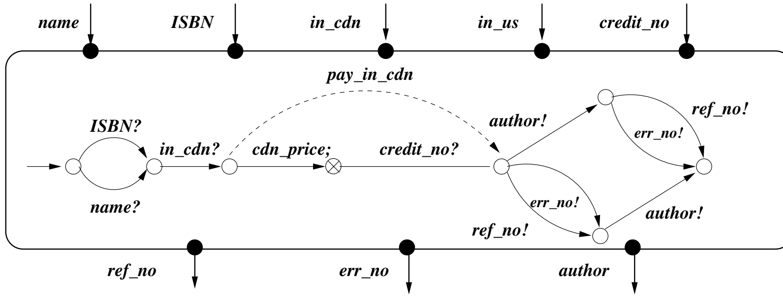

Fig. 6. $Prod \parallel CompPay$

and the output *cdn_price* in *Prod*, which results in a hidden action within the complex action *pay_in_cdn*. Similar to IA composition in Figure 3, states where shared output action *us_price* is an output but there is no input ready from the other IACA are illegal states and are not included in the product.

We define the composition of two IACAs using the following sequence of constructive steps:

(1) Compute the interleaved product of two IACAs, $P * Q$, leaving out interleaving of transitions within complex actions.

(2) Remove illegal normal states from the product.

(3) Remove illegal internal states from the interleaved result of (2). We call the result the legal interleaved product $P \circledast Q$.

(4) Compute the complex transitions to result in $P \parallel Q$.

We use the simple IACAs of Figure 7 to illustrate these steps.

Two IACAs are composable if the equivalent IAs are composable, and the normal actions in a complex transition either do not overlap with the actions in the other component's complex transitions, or for any that overlap, one is a prefix of the actions of the other. For example, in Figure 7 the complex action M of IACA B is a prefix of the complex action L of IACA A, thus they are composable.

In the first step, we compute the *interleaved product*, $P * Q$, illustrated by part (a) in Figure 7.

Definition 3.2 For two composable IACAs, P and Q, their sets of *interleaved states*, V_{P*Q}, and *interleaved transitions*, τ_{P*Q}, are:

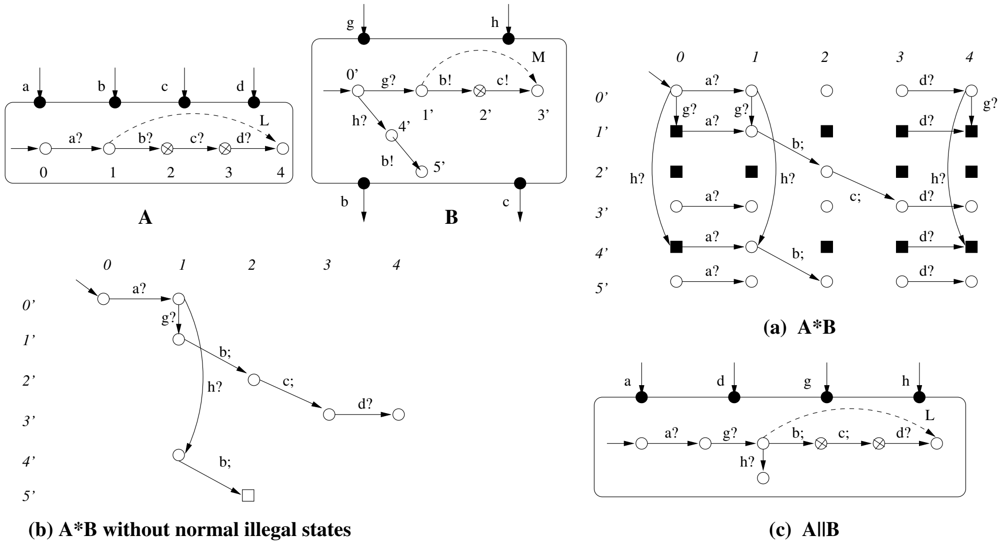

Fig. 7. Computing IACA composition for two composable IACAs.

$$
\begin{aligned}
V_{P*Q} \quad &= \quad V_P \times V_Q \\
\tau_{P*Q} \quad &= \\
&\{((p,q),a,(p',q')) \mid (p,a,p') \in \tau_P \wedge (q,a,q') \in \tau_Q \ \wedge a \in Shared(P,Q)\} \\
\cup\ &\{((p,q),a,(p',q)) \ \mid (p,a,p') \in \tau_P \wedge a \notin Shared(P,Q) \wedge p \in V_P^N \wedge q \in V_Q^N\} \\
\cup\ &\{((p,q),a,(p',q)) \ \mid (p,a,p') \in \tau_P \wedge a \notin Shared(P,Q) \wedge \\
&\qquad\qquad\qquad\qquad p \in V_P^{Int} \wedge q \in V_Q^N\} \\
\cup\ &\qquad /* \text{ similar to second and third sets above for } Q */
\end{aligned}
$$

where $shared(P,Q) = \mathcal{A}_P^N \cap \mathcal{A}_Q^N$. □

The first set of transitions in Definition 3.2 consists of synchronizations that can happen between two IACAs on shared actions; the composability criteria guarantee that each action a is an input action of one IACA and an output of the other IACA. The second set of transitions consists of those that do not involve shared actions and are transitions that exit normal states. The third set is for transitions originating from internal states in one component; these are not interleaved with the transitions from the other component. The definition of the third set excludes the transitions where two internal states of two IACAs issue different actions; no transitions will be initiated in those situations. However, if two such actions can synchronize, then they are included in the first set. For each state $(p,q) \in V_{P*Q}$, if either p or q is an internal state, then (p,q) is an internal state of the product. For example, in Figure 7, all

internal states associated with state 3 of A only initiate action $d?$ and do not interleave with other actions. At internal state $(3, 2')$ there is not any action initiated since these are both internal states and they cannot synchronize with their corresponding actions at those states, *i.e.,* $d?$ and $c!$. We have proven that in the interleaved product, the internal states of the product satisfy the constraints that they have at most one incoming and one outgoing transition.

In the second step, we remove the *illegal normal states* from the interleaved product. The illegal normal states of two IACAs are the same as illegal states for their equivalent IAs. Figure 7 shows the illegal normal states of the two IACAs as black filled boxes in the interleaved product. These are states where one component is ready to output an action but the other component is not ready to receive it. We remove all illegal normal states and transitions that are on paths that consist *entirely* of output and hidden actions that lead to illegal normal states. We also remove all non-reachable transitions. The result is shown in part (b) of Figure 7.

In the third step, we consider the *illegal internal states*, which are those internal states of the product that have no outgoing transitions. These cannot be part of a complex action because all complex fragments must terminate in a normal state. The empty box in part (b) of Figure 7 is an illegal internal state and thus should be removed. Removing illegal internal states may create some additional illegal internal states that should be removed until there are no more illegal internal states. We call the result at this point the *legal interleaved product* $(P \circledast Q)$.

We have proven that in the legal interleaved product, every reachable internal state $(p, q) \in V_{P \circledast Q}$ is part of a unique complex fragment in $P \circledast Q$. The fourth and final step is to determine the complex action associated with each complex fragment. Given two composable IACAs, P and Q, Δ is a function that returns a complex fragment associated with an internal state of the product. Each complex fragment $s = \langle (p_0, q_0), a_0, (p_1, q_1), \ldots, (p_n, q_n) \rangle$ returned by function Δ can itself be projected into two alternating sequence of states and actions, one belonging to P the other to Q. We define $\pi_P(s) = \langle p_0, a_0, p_1, \ldots p_n \rangle$ and π_Q similarly. For these complex fragments, we have proven the following:

Lemma 3.3 *For a reachable internal state $(p, q) \in V_{P \circledast Q}$, its complex fragment $s = \Delta(p, q)$, and the projections of s, $\pi_P(s)$ and $\pi_Q(s)$, one of the following is true:*

- $\exists!(p, d, p') \in \phi_P \cdot \mathit{frag}(p, d, p') = \pi_P(s)$
- $\exists!(q, e, q') \in \phi_Q \cdot \mathit{frag}(q, e, q') = \pi_Q(s)$

where $\exists!$ means "there exists a unique".

Using Lemma 3.3, we can define the function $complex_{P \circledast Q}$ that maps an internal state into exactly one complex transition with action c. The complex action c is uniquely identified by Lemma 3.3 except in the case where both conditions are true; in this situation, we pick the complex action that has some input constituent elements. Part (c) of Figure 7 shows the composition of A and B.

Definition 3.4 The composition of two composable IACAs P and Q, $P \parallel Q$, is an IACA defined as follows:

$$
\begin{aligned}
V^N_{P\|Q} &= V^N_P \times V^N_Q \\
V^{Int}_{P\|Q} &= \{(p,q) \in (V_P \times V_Q) |\ (p \in V^{Int}_P) \vee (q \in V^{Int}_Q)\} \\
i_{P\|Q} &= (i_P, i_Q) \\
\mathcal{A}^I_{P\|Q} &= (\mathcal{A}^I_P \cup \mathcal{A}^I_Q) \setminus Shared(P,Q) \\
\mathcal{A}^O_{P\|Q} &= (\mathcal{A}^O_P \cup \mathcal{A}^O_Q) \setminus Shared(P,Q) \\
\mathcal{A}^H_{P\|Q} &= \mathcal{A}^H_P \cup \mathcal{A}^H_P \cup Shared(P,Q) \\
\mathcal{A}^C_{P\|Q} &= \mathcal{A}^C_P \cup \mathcal{A}^C_Q \\
\tau_{P\|Q} &= \tau_{P \circledast Q} \\
\phi_{P\|Q} &= \{complex_{P \circledast Q}(p,q) |\ (p,q) \in V^{Int}_{P\|Q}\}
\end{aligned}
$$

□

In practice, we can use only one of the internal states (the first one) of a complex fragment to compute the complex transitions rather than all of them.

We have proven:

Theorem 3.5 *Given two composable IACAs P and Q, $P \parallel Q = Q \parallel P$.*

However, IACA is not a full-fledged interface model because composition is not associative. The major consequence of lack of associativity is that we cannot reason about composition of multiple IACA in an *arbitrary* order of composition. Instead, we have to consider multiple groupings of components. We plan to investigate ways to determine groupings for composition that would yield a maximal result, *i.e.,* choosing composition parenthesizations that would increase the chance of synchronization among different IACAs. In the absence of shared actions among multiple IACAs, their composition is associative.

3.2 Refinement

A refined version of an IACA can replace it in a composition. As with IA, a refined model may have more inputs and less outputs than the model it refines. For Q to refine P, there must be an alternating simulation relation between

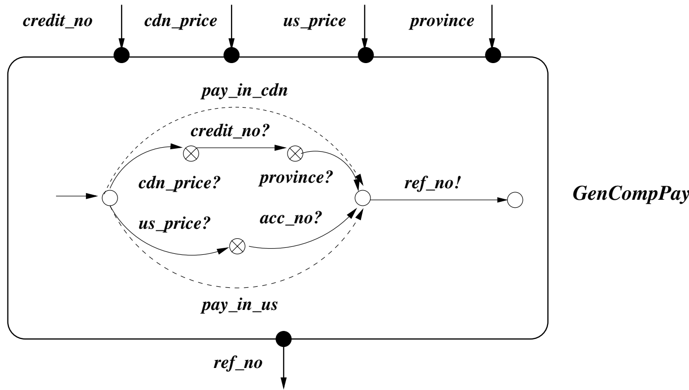

Fig. 8. *GenCompPay* is the refinement of the IACA in Figure 5.

the states of Q and P. A state q refines a state p if q has more than or the same inputs as p and less than or the same outputs as p. Additionally, for all states q' reachable from q immediately or through hidden actions, there must be a p' reachable from p such that q' refines p'. For the complex actions of IACA, a refinement may have additional input actions at the end of the complex fragment or fewer output actions from the end of the complex fragment. This restriction ensures that other IACAs that synchronize with this component in composition are still able to synchronize with the refined version.

As an example, IACA in Figure 8 is the refinement of IACA *CompPay* in Figure 5. This IACA is capable of carrying out payments in Canadian and US dollars (more inputs), however, it only provides a reference number as output and does not provide any error number (less outputs). Furthermore, the credit card payment accepts the province to determine appropriate taxation (more inputs at the end of a complex action).

Our goal in introducing IACA is to capture the idea of parameters to methods or complex messages in Web services using complex actions. IACA refinement matches the programming language concepts of optional parameters and subclasses. In programming languages, such as C/C++, conventionally, optional parameters must appear at the end of a function signature. In programming languages, such as Java and C++, a subclass of a class can have additional methods but also has the methods of its parent. Similarly, a refined version of an IACA provides all of the original IACA's complex actions and possibly more.

To define IACA refinement, we must first partition the complex actions into three sets based on whether the associated complex fragment has: (1) input and hidden actions (C_P^I), (2) output and hidden actions (C_P^O), and (3) only hidden actions (C_P^H). The definition of refinement is as follows:

Definition 3.6 IACA Q refines IACA P, $Q \preceq P$, if:

(1) $\mathcal{A}^I_P \subseteq \mathcal{A}^I_Q$ (Q has the same or more normal inputs than P)
(2) $\mathcal{A}^O_P \supseteq \mathcal{A}^O_Q$ (Q has the same or fewer normal outputs than P)
(3) $C^I_P \subseteq C^I_Q$ (Q has the same or more complex input actions than P)
(4) $C^O_P \supseteq C^O_Q$ (Q has the same or fewer complex output actions than P)
(5) $i_Q \preceq i_P$ (there is an alternating simulation relation $\preceq$ between the states of Q and P) □

Constraint (5) above propagates the alternating simulation relation to apply to all states of the two IACAs. By starting from the initial states of two IACAs, the alternating simulation relation is checked on all corresponding states. Next, we define this relation. For simplicity, we refer to the alternating simulation relation as the refinement relation on states.

First, we define the states that are reachable immediately or through hidden actions from a state:

Definition 3.7 For each normal state $p \in V^N_P$, the set $\varepsilon\text{–}closure_P(p)$ is defined as the set containing p itself and the normal states that can be reached from p through normal transitions with hidden actions. (These may include transitions of a complex action.) □

Next, we define the sets of *enabled* normal and complex actions, which are the actions that occur on transitions immediately exiting a state or reachable from a state through hidden actions. States that are reachable through hidden actions are considered the same for the purpose of refinement. We consider only the inputs that exit *all* of these states (because the environment may send an input to any of such state without knowing which of them exactly receives it and so all of them should be receptive to the input), but consider all outputs from these states (because the environment can accept any of such outputs). The functions $\mathcal{A}^I_P(p), \mathcal{A}^O_P(p), \mathcal{A}^C_P(p)$ return the input, output, and complex actions, respectively, on transitions exiting state p.

Definition 3.8 For each normal state $p \in V^N_P$ of IACA P,

- The sets of *enabled normal input and output actions* are:

$$EnNorm^I_P(p) = \{a \mid \forall p' \in \varepsilon\text{–}closure_P(p) \cdot a \in \mathcal{A}^I_P(p')\}$$

$$EnNorm^O_P(p) = \{a \mid \exists p' \in \varepsilon\text{–}closure_P(p) \cdot a \in \mathcal{A}^O_P(p')\}$$

- The sets of *enabled complex input and complex output actions* are:

$$EnComp^I_P(p) = \{a \in C^I_P \mid \forall p' \in \varepsilon\text{–}closure_P(p) \cdot a \in \mathcal{A}^C_P(p')\}$$

$$EnComp^O_P(p) = \{a \in C^O_P \mid \exists p' \in \varepsilon\text{–}closure_P(p) \cdot a \in \mathcal{A}^C_P(p')\}$$

- $En^O_P(p) = EnNorm^O_P(p) \cup EnComp^O_P(p)$
- $En^I_P(p) = EnNorm^I_P(p) \cup EnComp^I_P(p)$ □

Having defined the sets of actions that can be expected from a state p and states reachable from p through hidden transitions, we now define the sets of states that can be reached from a state with a certain action.

Definition 3.9 For IACA P, a normal state $p \in V^N_P$, and an enabled action $a \in En^I_P(p) \cup En^O_P(p)$, the set of reachable states of p by a is:

$$Dest_P(p, a) = \{p' | \exists r \in \varepsilon\text{–}closure_P(p) \cdot (\exists (r, a, p') \in \tau_P) \vee (\exists (r, a, p') \in \phi_P)\} \square$$

Finally, we can define the refinement relation between two states. This relation intuitively says that for every state $p \in V^N_P$, there is an alternating simulation through state $q \in V^N_Q$; q is receptive to all input actions, normal or complex, to which p is receptive; q does not issue outputs that p does not.

Definition 3.10 For two IACAs, P and Q, the binary relation *alternating simulation* $\preceq \subseteq V^N_Q \times V^N_P$ between two states $q \in V^N_Q$ and $p \in V^N_P$ holds if all of the following conditions are true:

- $EnNorm^I_P(p) \subseteq EnNorm^I_Q(q)$
  (q may have the same or more normal inputs)
- $EnNorm^O_P(p) \supseteq EnNorm^O_Q(q)$
  (q may have the same or fewer normal outputs)
- $EnComp^I_P(p) \subseteq EnComp^I_Q(q)$
  (q may have the same or more complex inputs)
- $EnComp^O_P(p) \supseteq EnComp^O_Q(q)$
  (q may have the same or fewer complex outputs)
- $\forall a \in EnComp^I_P(p) \cdot \forall (m, a, n) \in \phi_P \cdot m \in \varepsilon\text{–}closure_P(p) \Rightarrow$
  $\exists (r, a, s) \in \phi_Q \cdot r \in \varepsilon\text{–}closure(q) \wedge act(m, a, n) \sqsubseteq act(r, a, s)$
  (For every reachable complex transition from p on a complex input, there is a reachable complex transition from q on the same action and the complex fragment of P must be a prefix of the complex fragment of Q.)
- $\forall a \in EnComp^O_Q(q) \cdot \forall\ (r, a, s) \in \phi_Q \cdot r \in \varepsilon\text{–}closure_P(q) \Rightarrow$
  $\exists (m, a, n) \in \phi_P \cdot m \in \varepsilon\text{–}closure(p) \wedge act(r, a, s) \sqsubseteq act(m, a, n)$
  (For every reachable complex transition from q on a complex output, there is a reachable complex transition from p on the same action and the complex fragment of Q must be a prefix of the complex fragment of P.)
- $\forall a \in En^I_P(p) \cup En^O_Q(q) \cdot \forall\ q' \in Dest_Q(q, a) \Rightarrow \exists p' \in Dest_P(p, a) \cdot q' \preceq p'$
  ($\preceq$ holds for everything reachable under inputs for p and outputs for q) □

The purpose of refinement is to support top-down design. As an example,

consider the composition of $GenCompPay$, in Figure 8, with IACA of $Prod$ in Figure 1. Since $GenCompPay \preceq CompPay$ then $(GenCompPay \parallel Prod) \preceq (CompPay \parallel Prod)$. We have proven:

Theorem 3.11 *Given three IACAs P, Q and P' such that $P' \preceq P$, and P and Q are composable and P' and Q are composable, then $(P' \parallel Q) \preceq (P \parallel Q)$, if the following conditions hold:*

- $Shared(P, Q) = Shared(P', Q)$
  (P' and P communicate with Q through the same set of shared actions)
- $\forall (p', p) \cdot (p' \preceq p) \wedge (p' \in V^N_{P'}) \wedge (p \in V^N_P) \Rightarrow$
  $((\mathcal{A}^I_{P'}(p') \backslash \mathcal{A}^I_P(p)) \notin Shared(P, Q)) \wedge ((\mathcal{A}^O_P(p) \backslash \mathcal{A}^O_{P'}(p')) \notin Shared(P, Q))$
  (States of P' that are in the simulation relation with P, do not introduce extra (nor eliminate) actions that belong to the shared actions of P and Q.)
- $\forall (p', p) \cdot (p' \preceq p) \wedge (p' \in V^N_{P'}) \wedge (p \in V^N_P) \Rightarrow$
  $(\forall (p, c, u) \in \phi_P \cdot \exists (p', c, v) \in \phi_{P'} \wedge (c \in C^O_{P'})$
  $\Rightarrow ((set(act(p, c, u))) \backslash (set(act(p', c, v))) \cap (Shared(P, Q))) = \emptyset)$
  (States of P' that are in the simulation relation with P should not introduce output complex actions, shared with P, that introduce new actions that belong to the shared actions of P and Q.)
  Operator "set" maps a sequence to a set containing all members of the sequence.
- $\forall (p', p) \cdot (p' \preceq p) \wedge (p' \in V^N_{P'}) \wedge (p \in V^N_P) \Rightarrow$
  $(\forall (p', c, v) \in \phi_{P'} \cdot \exists (p, c, u) \in \phi_P \wedge (c \in C^I_P)$
  $\Rightarrow ((set(act(p', c, v))) \backslash (set(act(p, c, u))) \cap (Shared(P, Q))) = \emptyset)$
  (States of P that are in the simulation relation with P' should not introduce input complex actions, shared with P', that introduce new actions that belong to the shared actions of P and Q.)

Theorem 3.11's conditions require that P' preserves the same shared actions that P has with Q and plus requires P' to behave in accordance to P on the shared actions of P and Q. In other words, we require that P' neither increases nor decreases the shared actions that P and Q have. Additionally, we also require that at the state level, the refined state and the original state use the same set of shared actions. Comparing IACA's top-down design criteria with IA, IA is more lenient; in a similar setting IA only requires $Shared(P', Q) \subseteq Shared(P, Q)$ for a similar top-down design result as in Theorem 3.11. Our restriction arises from the fact that we are not only dealing with illegal normal states (as IAs also deal with) but also deal with illegal internal states. In other words to support top-down design, P', the refinement of P, should behave in such a way that it does not cause new illegal internal

states that the composition of P and Q does not create.

Another issue is that, in our theorem we require that P' and Q to be composable; there is no guarantee that since P and Q are composable, then P' and Q should be composable as well. This happens because of extra complex actions that P', as refinement of P, can have at its states. There is no way to guarantee that such extra complex actions observe the overlapping composability criteria of IACAs. Alternatively, we could have defined our refinement relation in such a way that it would have disallowed the extra complex actions, and hence avoid explicitly requiring P' and Q to be composable.

4 Related Work

The idea of grouping activities in a sequential, non-interruptible manner is common in many contexts. For example, in databases, the concept of a transaction is pivotal and resembles our "complex" actions. Within the context of concurrency semantics, different approaches have been proposed to augment process algebraic-like languages to support non-interruptible sequences of actions. Such approaches can be generally categorized into two groups: (1) *atomic actions* (*e.g.,* [9,3]) and (2) *action refinement* (*e.g.,* [1]). The first category includes the work most comparable to ours; Gorrieri *et al.* enhance CCS [13] to support non-interruptible actions [9]. Their proposed composition operator is non-associative and they suggest that non-associativity may be an intrinsic property of handling complex actions. Input/Output Automata [11], which inspired interface automata, also do not support associative composition with action hiding. Action refinement approaches allow *stepwise refinements* of models into their more concrete equivalents. For a recent comprehensive treatment of action refinement, in a not entirely algebraic setting, readers can refer to [14].

Promela, the language of the Spin model checker [10], implements complex actions using the keywords `atomic` and `d_step`. Promela's `atomic` sequences may block and allow interleaving if an input is not available or an output cannot be consumed. `d_step` sequences must be deterministic and do not allow interleaving; a run-time error will occur if actions grouped in a `d_step` cannot synchronize when necessary. Our complex actions are similar to `d_step`. Composition in Promela is an n-ary operator and there is no defined notion of refinement. As such, associativity in its composition is irrelevant.

While our approach has the same goals as much of the work mentioned above, we differ because we have created an automata-based interface model with complex actions that has most of the properties of interface models, which are designed to be a concise way to specify component-based systems.

5 Conclusion and Future Work

We have introduced *interface automata with complex actions* (IACA), which add complex actions to de Alfaro and Henzinger's interface automata. The transitions within a complex action are not interleaved with transitions from another component in composition. Complex actions allow us to model non-interruptible behaviour, which is needed to describe parameters of methods or Web services messages. IACA has all the properties of an interface model except for associativity of composition.

An immediate application for IACA, as described in [7], is in modelling Web services. Web services communicate with other Web services and their service requesters through input and output messages. Such messages are basically XML messages. Complex XML messages can have constituent elements that should not be interleaved with other communication, and IACA is a suitable means of modelling them.

In our future work, we plan to investigate how we can overcome the challenge of lack of associativity in IACA. Also, for Web services, we may need to combine services in response to a search query. Through heuristics, we may be able to reduce the need to search all possible associativity orderings.

References

[1] L. Aceto. *Action Refinement in Process Algebras*. Cambridge University Press, 1992.

[2] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *Proc. 9th Conferance on Concurrency Theory*, Lecture Notes in Computer Science. Springer-Verlag, September 1998.

[3] Gérard Boudol. Atomic actions (note). *Bulletin of the European Association for Theoretical Computer Science*, 38:136–144, June 1989. Technical Contributions.

[4] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *Software Engineering Notes*, pages 109–120. ACM Press, September 10–14 2001.

[5] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In *Proceedings of the First International Workshop on Embedded Software*, volume 2211, pages 148–165. Lecture Notes in Computer Science 2211, Springer-Verlag, 2001.

[6] Luca de Alfaro and Thomas A. Henzinger. Interface-Based Design. In *Proceedings of the Marktoberdorf Summer School, Kluwer*, Engineering Theories of Software Intensive Systems, 2004.

[7] Shahram Esmaeilsabzali. An Interface Approach to Discovery and Composition of Web Services. Master of Mathematics, School of Computer Science, University of Waterloo, June 2004.

[8] Shahram Esmaeilsabzali, Farhad Mavaddat, and Nancy A. Day. Interface automata with complex actions. Technical Report CS-2005-26, University of Waterloo, School of Computer Science, 2005.

[9] R. Gorrieri, S. Marchetti, and U. Montanari. A^2CCS: atomic actions for CCS. *Theoretical Computer Science*, 72(2-3):203–223, May 1990.

[10] Gerard J. Holzmann. The model checker Spin. *IEEE Trans. Soft. Eng.*, 23(5):279–295, 1997.

[11] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 519–543, 1987.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[13] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[14] Rob van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Inf.*, 37(4-5):229–327, 2000.