Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 263 (2010) 179–195
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Action Prefixes: Reified Synchronization Paths in Minimal Component Interaction Automata

Markus Lumpe[1]

Faculty of Information & Communication Technologies, Swinburne University of Technology
P.O. Box 218, Hawthorn, VIC 3122, AUSTRALIA

Abstract

Component Interaction Automata provide a fitting model to capture and analyze the temporal facets of hierarchical-structured component-oriented software systems. However, the rules governing composition, as is typical for all automata-based approaches, suffer from *combinatorial state explosion*, an effect that can have significant ramifications on the successful application of the *Component Interaction Automata* formalism to real-world scenarios. We must, therefore, find some appropriate means to counteract state explosion – one of which is *partition refinement* through weak bisimulation. But, while this technique can yield the desired state space reduction, it does not consider *synchronization cliques*, *i.e.*, groups of states that are interconnected solely by internal synchronization transitions. Synchronization cliques give rise to *action prefixes* that capture pre-conditions for a component's ability to interact with the environment. Current practice does not pay attention to these cliques, but ignoring them can result in a loss of valuable information. For this reason we show, in this paper, how synchronization cliques emerge and how we can capture their behavior in order to make state space reduction aware of the presence of synchronization cliques.

Keywords: Component Interaction Automata, Partition Refinement, Emerging Properties

1 Introduction

Component-based software engineering has become the prevalent trend in present-day software and system engineering [6]. In this approach the focus is on *well-defined interfaces* [3,7,11,26] that provide appropriate means for decomposing an engineered system into logical and interacting entities, the *components*, and constructing their respective aggregations, the *composites*, to yield the desired system functionality at matching levels of abstraction and granularity. Moreover, according to this technique, new components are created by combining pre-existing ones with new software, the *glue* [24], using only the information published in the interface specifications of the components being composed.

[1] Email: mlumpe@swin.edu.au

1571-0661 © 2010 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2010.05.011

Component interfaces can convey a variety of information [1] that collectively form a *contractual specification*. Ideally, all assumptions about a component's environment should be stated explicitly and formally as part of the interface specification [25]. However, even if the interfaces have been organized in such a way that their embodied contractual specifications guarantee safe deployment in new contexts, the information pertaining to the interfaces must not provide any instruments to circumvent component encapsulation. On the other hand, the purpose of contractual specifications is to prevent errors, at both design time and run-time. Therefore, component contracts should impose a well-balanced set of constraints to enforce contractual obligations, but must be defined in a manner so that the reasons why a particular contract verification has failed are self-evident [4].

In this paper, we are concerned with the specification of *behavioral* and *synchronization* contracts [1] between interacting components. In particular, we study the effectiveness of *Component Interaction Automata* [2,5], an automata-based modeling language for the specification of hierarchical-structured component-based systems. Components synchronize through answering mutual service requests. However, some service requests should only occur in certain situations [23] depending on the component's readiness to satisfy a given request (*pre-condition*) and its cumulative interaction profile (*post-condition*). The description of these temporal aspects corresponds best to *finite automata* in which *acceptable service requests* are modeled as state transitions between activating sets (*i.e.*, states of the modeling automaton) [23].

Unfortunately, automata-based formalisms suffer from *combinatorial state explosion*, a major obstacle to the successful application of these approaches for the specification of the interactive behavior in component-based systems. More precisely, to capture the complete behavior of an automata-based system, we need to construct the *product automaton* of the system's individual components [10]. This operation exhibits exponential space and time complexity and the resource consumption quickly reaches a level at which an effective specification of a composite system is not feasible anymore [13]. We need, therefore, workable abstraction methods that allow for a reduction of the composite state space at acceptable costs.

For this reason, we have developed a bisimulation-based partition refinement algorithm for *Component Interaction Automata* [13]. Partition refinement [9,20] is a state space reduction technique that, driven by a corresponding equivalence relation, merges equivalent states into one unifying representative. On termination, partition refinement yields a new automaton that reproduces the behavior of the original one up to the defined equivalence, but is *minimal* (*i.e.*, a fixed-point) with respect to the number of required states.

Partition refinement can effectively reduce the size and the complexity of composite component interaction automata [13]. There are, however, instances in which partition refinement produces unexpected outcomes. In particular, we notice a frequent appearance of newly-observable *non-deterministic* transitions in minimal composite component interaction automata, even when there were none before. These non-deterministic transitions can cause harm since their elimination, in order to im-

plement the automaton, may require exponentially more states [10], which is clearly not a desirable scenario.

Upon closer inspection we find that these non-deterministic transitions are directly linked to states that are involved in internal component synchronizations and that become unified as result of partition refinement. Following network theory [17], these states form *synchronization cliques*, groups of states, which embed in their structure *regular sublanguages* over an alphabet of internal component synchronizations. The sentences of these regular sublanguages serve as *prefixes* (or pre-conditions) in the interface of a given composite component interaction automaton. Before refinement, these prefixes are woven into the fabric of the composite automaton. Partition refinement, however, is blind for this additional information, as, independent of its presence, observable equivalence is always preserved between the original and the reduced automaton.

Synchronization cliques are *intrinsic* to automata-based approaches that enumerate internal synchronization actions [2,5,6,14,27] rather than modeling them by τ – a *perfect action* [16]. As a consequence, an external observer can monitor not only the occurrences of internal synchronizations (through the passing of time), but also the order in which actions actually trigger the internal synchronizations. We can capture the alternating sequences of states and internal synchronization actions in synchronization paths [5,13]. However, weak bisimulation is an equivalence relation that abstracts from internal actions, resulting in a loss of information, including the ability to monitor the sequence of internal synchronizations. We show, in this paper, that we can recover this information by representing the existing synchronization paths in a system as *action prefixes* in the corresponding reduced component interaction automaton, if needed.

The rest of this paper is organized as follows: in Section 2, we review the *Component Interaction Automata* formalism and demonstrate its expressive power on a tailored version of a simple e-commerce application. We proceed by developing the core ingredients of observable equivalence and partition refinement for component interaction automata in Section 3 and construct, in Section 4, the machinery to distill action prefixes from synchronization cliques. We discuss possible implications of the existence of synchronization cliques in Section 5 and conclude with a brief summary of our main observations and an outlook to future work in Section 6.

2 The Component Interaction Automata Modeling Language

I/O Automata [14], *Interface Automata* [6], and *Team Automata* [27] have all emerged as light-weight contenders for capturing concisely the *temporal* aspects of component-based software systems. These formalisms use an *automata-based language* to represent both the assumptions about a system's environment and the order in which interactions with the environment can occur. However, none of these models cater directly for multiple instantiations of the same component within a single system or allow for a more fine-grained specification of hierarchical relationships

between organizational entities in a system.

These restrictions have been relaxed in *Component Interaction Automata* [2,5]. In this approach we find two new concepts: a *hierarchy of component names* and *structured labels*. The former provides us with a means to record the architecture of a composite system, whereas the latter paves the way to specify the *action*, the *originating component*, and the *target component* in the transitions of component interaction automata as one, a feature that allows us to disambiguate multiple occurrences of the same component (or action) within a single system. Specifically, the *Component Interaction Automata* formalism supports three forms of structured labels: $(-, a, n)$, receive a from the environment as *input* at component n, $(n, a, -)$, send a from component n as *output* to the environment, and (n_1, a, n_2), components n_1 and n_2 *synchronize* internally through action a.

The *Component Interaction Automata* formalism uses *component identifiers* to uniquely identify specific component instances in a given system. However, a given component identifier can occur at most once in a composite component interaction automaton. This requirement addresses a frequent difficulty in the specification of component-based systems – the difference between components and component instances [12]. The *I/O Automata* and *Interface Automata* formalisms, for example, do not distinguish between components and their instances. Every specification involves only instances. It is for this reason that all actions of composed components have to be pairwise disjoined [6,14] (*i.e.*, a single component instance can occur at most once in a composite system). In contrast, the *Component Interaction Automata* formalism distinguishes between components and their instances. Each component is instantiated with a unique identifier that we use also to disambiguate the corresponding component transitions. Consider, for example, a component $\mathcal{C}$ that defines an input via action a and two instances of $\mathcal{C}$, named A and B. Then the structured labels for the input transitions of A and B are $(-, a, A)$ and $(-, a, B)$, respectively. The unique component identifiers A and B are what allows for the safe coexistence of multiple instances of the same component (or action) in a given system.

We presuppose a countably infinite set $\mathcal{A}$ of *component identifiers*. A hierarchy of component names is defined as follows [5]:

Definition 2.1 A hierarchy of component names H is defined recursively by

- $H = (a_1, ..., a_n)$, where $a_1, ..., a_n \in \mathcal{A}$ are pairwise disjoint component identifiers and $S(H) = \cup_{i=1}^{n} \{a_i\}$ denoting the set of component identifiers of H;
- $H = (H_1, ..., H_m)$, where $H_1, ..., H_m$ are hierarchies of component identifiers satisfying $\forall\ 1 \leq i, j \leq m, i \neq j :\ S(H_i) \cap S(H_j) = \emptyset$ and $S(H) = \cup_{i=1}^{m} S(H_i)$ denoting the set of component identifiers of H.

Definition 2.2 A component interaction automaton $\mathcal{C}$ is a quintuple (Q, Act, δ, I, H) where:

- Q is a finite set of states,
- Act is a finite set of actions,

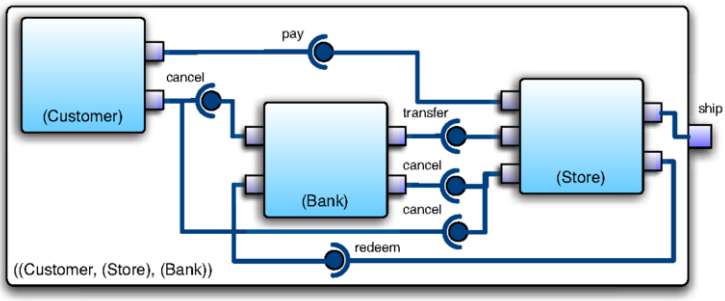


Fig. 1. A simple e-commerce system.

- $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of labeled transitions, where $\Sigma \subseteq \{(S(H) \cup \{-\} \times Act \times S(H) \cup \{-\})\} \setminus \{(\{-\} \times Act \times \{-\})\}$ is the set of structured labels induced by $\mathcal{C}$,
- $I \subseteq Q$ is a non empty set of initial states, and
- H is a tuple denoting $\mathcal{C}$'s hierarchical composition structure.

Each component interaction automaton is further characterized by two sets of $P \subseteq Act$, the provided actions, and $R \subseteq Act$, the required actions, which capture the automaton's enabled interface with an environment. We write $\mathcal{C}_R^P$ to denote an automaton $\mathcal{C}$ that is input-enabled in P and output-enabled in R.

In the original definition [2,5], the set of *provided actions* P and the set of *required actions* R originate from a secondary specification outside the *Component Interaction Automata* formalism. Incorporating these *architectural constraints* into the specification of component interaction automata does not affect the underlying composition rules, but it rather makes the relationship with the associated automata more explicit and eases the computation of composition [13]. We abbreviate, however, the annotation in a natural way if an automaton is enabled in all actions and omit the corresponding specification.

As motivating example, consider a simple electronic commerce system with three participants [10]: a *Customer*, a *Store*, and a *Bank*. The behavior of the composite system is as follows. The customer may initiate a transaction by passing a voucher to the store. The store will then redeem this voucher with the bank (*i.e.*, the bank will eventually deposit money into the store's account) and, through a third party, ship the ordered goods. In addition, the customer may cancel the order before the store had a chance to redeem the voucher, in which case the voucher will be returned to the customer immediately. We allow the customer to cancel an order with either the store or the bank. The high-level interaction protocol for this system is shown in Figure 1.

We can model *Customer*, *Store*, and *Bank* as component interaction automata as follows. We write $(Customer)$, $(Store)$, and $(Bank)$ to denote the architecture of the component interaction automata *Customer*, *Store*, and *Bank*. More precisely, the three automata are *primitive* (or plain) components with an opaque structure (*i.e.*, no explicit hierarchical relationships):

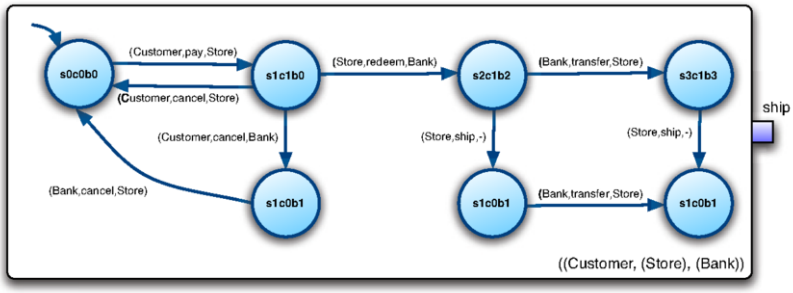


Fig. 2. The composite e-commerce component interaction automaton.

$$
\begin{aligned}
Customer = (&\{c_0, c_1\}, \{pay, cancel\},\\
&\{(c_0, (Customer, pay, -), c_1), (c_1, (Customer, cancel, -), c_0)\},\\
&\{c_0\}, (Customer))
\end{aligned}
$$

$$
\begin{aligned}
Store = (&\{s_0, s_1, s_2, s_3, s_4, s_5\}, \{pay, redeem, transfer, ship\},\\
&\{(s_0, (-, pay, Store), s_1), (s_1, (Store, redeem, -), s_2), (s_1, (-, cancel, Store), s_0),\\
&(s_2, (-, transfer, Store), s_3), (s_2, (Store, ship, -), s_4), (s_3, (Store, ship, -), s_5),\\
&(s_4, (-, transfer, Store), s_5)\}, \{s_0\}, (Store))
\end{aligned}
$$

$$
\begin{aligned}
Bank = (&\{b_0, b_1, b_2, b_3\}, \{cancel, redeem, transfer\},\\
&\{(b_0, (-, cancel, Bank), b_1), (b_0, (-, redeem, Bank), b_2),\\
&(b_1, (Bank, cancel, -), b_0), (b_2, (Bank, transfer, -), b_3)\},\\
&\{b_0\}, (Bank))
\end{aligned}
$$

The *Customer* automaton has two states and two output transitions (*i.e.*, customer requests). The *Store* automaton, on the other hand, defines six states and seven transitions and guarantees that orders will only be shipped, if the payment voucher has been redeemed successfully. The *Store* receives a voucher (*i.e.*, action *pay*), money (*i.e.*, action *transfer*), or a cancelation as input and issues as output the shipment of goods and the request to redeem the voucher. Finally, the *Bank* automaton, defining four states and four transitions, coordinates *Customer* and *Store*. If the *Store* has not yet cashed the voucher, then the *Customer* can still cancel the order and receive a refund. The *Bank* will forward a cancelation notice to the *Store*. If the *Store* has already submitted the voucher, then the *Bank* will eventually transfer funds to the *Store*. At this point, the *Customer* cannot cancel the order anymore.

The composition of component interaction automata is defined as the *cross-product* over their state spaces. Furthermore, the sets P and R determine, which input and output transitions occur in the composite system (*i.e.*, interface with the environment). By convention, if any state is rendered inaccessible in the composite automaton, then we remove it immediately from the state space in order to obtain the most concise result. The behavior of the composite automaton is *completely* captured by its reachable states.

Definition 2.3 Let $\mathcal{S}_R^P = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$ be a system of pairwise disjoint component interaction automata, where $\mathcal{I}$ is a finite indexing set and P, R are the provided and required actions. Then $\mathcal{C}_R^P = (\prod_{i\in\mathcal{I}} Q_i, \cup_{i\in\mathcal{I}} Act_i, \delta_{\mathcal{S}_R^P}, \prod_{i\in\mathcal{I}} I_i, (H_i)_{i\in\mathcal{I}})$ is a composite component interaction automaton where q_j denotes a function $\prod_{i\in\mathcal{I}} Q_i \to Q_j$, the projection from product state q to jth component state q and

$$\delta_{\mathcal{S}_R^P} = \delta_{OldSync} \cup \delta_{NewSync} \cup \delta_{Input} \cup \delta_{Output}$$

with

$$\begin{aligned}
\delta_{OldSync} &= \{(q,(n_1,a,n_2),q') \mid \exists i \in \mathcal{I} : (q_i,(n_1,a,n_2),q_i') \in \delta_i \ \wedge \\
&\qquad \forall j \in \mathcal{I}, j \neq i : \ q_j = q_j'\}, \\
\delta_{NewSync} &= \{(q,(n_1,a,n_2),q') \mid \exists i_1,i_2 \in \mathcal{I}, i_1 \neq i_2 : (q_{i_1},(n_1,a,-),q_{i_1}') \in \delta_{i_1} \wedge \\
&\qquad (q_{i_2},(-,a,n_2),q_{i_2}') \in \delta_{i_2} \wedge \ \forall j \in \mathcal{I}, i_1 \neq j \neq i_2 : q_j = q_j'\}, \\
\delta_{Input} &= \{(q,(-,a,n),q') \mid a \in R \ \wedge \ \exists i \in \mathcal{I} : (q_i,(-,a,n),q_i') \in \delta_i \ \wedge \\
&\qquad \forall j \in \mathcal{I}, j \neq i : q_j = q_j'\}, \\
\delta_{Output} &= \{(q,(n,a,-),q') \mid a \in P \ \wedge \ \exists i \in \mathcal{I} : (q_i,(n,a,-),q_i') \in \delta_i \ \wedge \\
&\qquad \forall j \in \mathcal{I}, j \neq i : q_j = q_j'\}.
\end{aligned}$$

The composition rule builds the product automaton for a given system $\mathcal{S}_R^P$. It does so by simultaneously recombining the behavior of all individual component interaction automata in $\{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$. The transitions of the composite automaton result from four sets: the transposed preexisting internal synchronizations $\delta_{OldSync}$ of the individual component interaction automata, the newly formed internal synchronizations $\delta_{NewSync}$ due to interactions between the individual component interaction automata, and the sets δ_{Input} and δ_{Output}, the transposed remaining interactions of the product automaton with the environment.

Applied to our e-commerce system, we can denote the composition of the three components *Customer*, *Store*, and *Bank* using the following expression:

$$\mathcal{S}_\emptyset^{\{ship\}} = \{Customer, Store, Bank\},$$

which yields a composite automaton with 7 reachable states (out of 48 product states). Moreover, due to the architectural constraints $P = \{ship\}$ and $R = \emptyset$ the composite system can only interact with its environment by emitting a *ship* action. A graphical representation of the composite system is shown in Figure 2.

3 Observable Equivalence and Partition Refinement

The problem of *combinatorial state explosion* does not only occur when constructing new composite components or systems, but also when we wish to study their inherent properties [13]. A measure to alleviate state explosion is *partition refinement* [8,9,13,18,20], which allows, by means of some equivalence relation, for the identification of states that exhibit the same interactive behavior with respect to an external observer. Partition refinement merges equivalent states into one and removes the remaining superfluous states and their transitions from the system. We use *bisimulation* [19], in particular a notion of *weak bisimulation* [13,16], as the desired observable equivalence relation for the reduction of component interaction

automata. From an external observer's point of view, weak bisimulation yields a *co-inductive* testing strategy in which two component interface automata cannot be distinguished, if they only differ in their internal synchronizations.

However, the *Component Interaction Automata* formalism requires an additional criterion to be met: two component interaction automata A and B are considered equivalent, if and only if they are bisimular and adhere to the same underlying composition structure [13]. In other words, any technique to reduce the complexity of a given component interaction automaton has also to retain its underlying hierarchical composition structure. This means, two states q, p with transitions $(q, (-, a, A), r)$ and $(p, (-, a, B), r)$ must not be equated, as the target components in the transition labels differ.

An important element in the definition of an observable equivalence relation over component interaction automata is the notion of *synchronization path*.

Definition 3.1 If $(n_1, a_1, n_1') \cdots (n_k, a_k, n_k') \in \Sigma$ are internal synchronizations of a component interaction automaton $\mathcal{C}$, then we write $q \stackrel{*}{\Longrightarrow} p$ to denote the reflexive transitive closure of

$$q \xrightarrow{(n_1,a_1,n_1')} r_1 \xrightarrow{(n_2,a_2,n_2')} \cdots \xrightarrow{(n_{k-1},a_{k-1},n_{k-1}')} r_{k-1} \xrightarrow{(n_k,a_k,n_k')} p,$$

called synchronization path between q and p.

Synchronization paths give rise to *weak transitions*.

Definition 3.2 If $l \in \Sigma$ is a structured label, then $q \stackrel{l}{\Longrightarrow} p$ is a weak transition from q to p over label l, if there exists r, r' such that

$$q \stackrel{*}{\Longrightarrow} r \stackrel{l}{\longrightarrow} r' \stackrel{*}{\Longrightarrow} p.$$

Using the concept of weak transitions, we can define now a *weak bisimulation* over component interaction automata.

Definition 3.3 Given $A = (Q_A, Act_A, \delta_A, I_A, H)$ and $B = (Q_B, Act_B, \delta_B, H)$, two component interaction automata with an identical composition hierarchy H, a binary relation $\mathcal{R} \subseteq Q \times Q$ with $Q = Q_A \cup Q_B$ is a weak bisimulation, if it is symmetric and $(q, p) \in \mathcal{R}$ implies, for all $l \in \Sigma$, $\Sigma = \Sigma_A \cup \Sigma_B$ being the set of structured labels induced by A and B,

- whenever $q \stackrel{l}{\longrightarrow} q'$, then $\exists p'$ such that $p \stackrel{l}{\Longrightarrow} p'$ and $(q', p') \in \mathcal{R}$.

Two component interaction automata A and B are weakly bisimilar, written $A \approx B$, if they are related by some weak bisimulation.

Applying the preceding definition, we can find a new automaton, $\mathcal{R}_{\emptyset}^{\{ship\}}$, capable of reproducing the interactive behavior of our e-commerce systems up to weak bisimulation. $\mathcal{R}_{\emptyset}^{\{ship\}}$ (cf. Figure 3) satisfies two requirements: (i) it interacts with the environment through the structured label $(Store, ship, -)$, and (ii) it adheres to the hierarchical composition structure $((Customer), (Store), (Bank))$.

To show that $\mathcal{R}_{\emptyset}^{\{ship\}}$ and $\mathcal{S}_{\emptyset}^{\{ship\}} = \{Customer, Store, Bank\}$ are indeed ob-

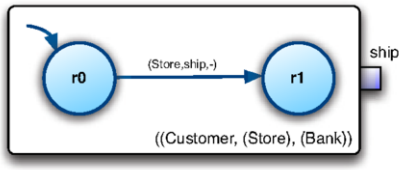


Fig. 3. The weakly-bisimular e-commerce component interaction automaton $\mathcal{R}_{\emptyset}^{\{ship\}}$.

servably equivalent with respect to an external observer, we have to find a weak bisimulation $\mathcal{R}$ such that $\mathcal{R}_{\emptyset}^{\{ship\}} \approx \mathcal{S}_{\emptyset}^{\{ship\}}$. Such a relation exists and is defined as $\mathcal{R} = r \cup r^{-1}$ with

$$r = \{(s0c0b0, r0), (s1c1b0, r0), (s1c0b1, r0), (s2c1b2, r0), (s3c1b3, r0), \\ (s4c1b2, r1), (s5c1b3, r1)\}.$$

There are only two states in $\mathcal{S}_{\emptyset}^{\{ship\}}$, $s2c1b2$ and $s3c1b3$, that require the automaton $\mathcal{R}_{\emptyset}^{\{ship\}}$ to move. Consider, for example, state $s2c1b2$ of $\mathcal{S}_{\emptyset}^{\{ship\}}$. Since $(s2c1b2, r0) \in \mathcal{R}$ and $\mathcal{S}_{\emptyset}^{\{ship\}}$ can perform $(s2c1b2, (Store, ship, -), s4c1b2)$, we select as a matching move the transition $(r0, (Store, ship, -), r1)$ of $\mathcal{R}_{\emptyset}^{\{ship\}}$ that yields the pair $(s4c1b2, r1) \in \mathcal{R}$, as required. For all states in $\mathcal{S}_{\emptyset}^{\{ship\}}$ other than $s2c1b2$ and $s3c1b3$, $\mathcal{R}_{\emptyset}^{\{ship\}}$ pauses, since all internal synchronization have been factored out in $\mathcal{R}_{\emptyset}^{\{ship\}}$.

The global tactic for the computation of bisimularity is *partition refinement*, which factorizes a given state space into equivalence classes [8,9,18,20]. The result of partition refinement is a surjective function that maps the elements of the original state space to its corresponding representatives of the computed equivalence classes. Partition refinement always yields a minimal automaton.

In the heart of partition refinement is a *splitter function* that determines the granularity of the computed equivalence classes. A splitter for component interaction automata is a boolean predicate $\gamma : Q \times \Sigma \times \mathcal{S} \mapsto \{\texttt{true}, \texttt{false}\}$, where $\mathcal{S} \subseteq 2^Q$ is a set of candidate equivalence classes for $\mathcal{C} = (Q, Act, \delta, I, H)$, the component interaction automaton in question.

Definition 3.4 Let $q \in Q$ be a state, $P \in \mathcal{S}$ be candidate equivalence class, and $l \in \Sigma$ be a structured label for a component interaction automaton $\mathcal{C} = (Q, Act, \delta, I, H)$. Then

$$\gamma(q, l, P) := \begin{cases} \texttt{true} & \textit{if there is } p \in P \textit{ such that } q \stackrel{l}{\Longrightarrow} p, \\ \texttt{false} & \textit{otherwise} \end{cases}$$

We obtain with this definition a means of expressing the computation of a weakly-bisimular component interaction automaton as the possibility of a set of its states, P, to evolve into another set of states, P', with the same observable behavior, where P' is the equivalence class of P.

Definition 3.5 Let γ be a splitter function generating weakly-bisimular equiva-

lence classes for a component interaction automaton $\mathcal{C} = (Q, Act, \delta, I, H)$. Then

$$refine(X, l, P) := \cup_{P' \in X}(\cup_{v \in \{\texttt{true},\texttt{false}\}}\{q \mid \forall q \in P'.\ \gamma(q, l, P) = v\}) - \{\emptyset\}$$

The actual refinement process is defined by a procedure, $refine : \mathcal{X} \times \Sigma \times \mathcal{S} \times \mathcal{X}$, that takes a set of partitions $X \in \mathcal{X}$, a structured label $l \in \Sigma$, and a candidate equivalence class $P \in \mathcal{S}$ to yield, possibly new, candidate equivalence classes. Partition refinement, starting with $X = \{Q\}$ as initial partition set, repeatedly applies *refine* to X and its derivatives for all $l \in \Sigma$ until a fixed-point is reached [9].

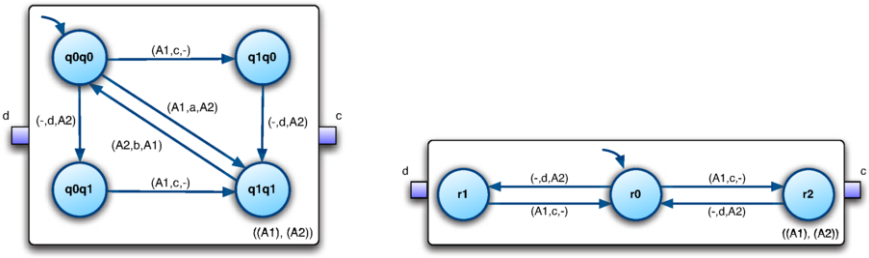
When applied to our composite e-commerce system $\mathcal{S}_{\emptyset}^{\{ship\}}$, partition refinement computes the following equivalence classes:

$$\{r0 = \{s0c0b0, s1c1b0, s1c0b1, s2c1b2, s3c1b3\}, r1 = \{s4c1b2, s5c1b3\}\},$$

which correspond exactly to the weak bisimulation $\mathcal{R}$, shown earlier. More precisely, we can use these equivalence classes to construct the automaton $\mathcal{R}_{\emptyset}^{\{ship\}}$.

4 Action Prefixes

Partition refinement, up to weak bisimulation, can eliminate most if not all, as in case of $\mathcal{R}_{\emptyset}^{\{ship\}}$, internal synchronizations from a given component interaction automaton. It provides, therefore, a suitable abstraction method that lets system designers focus on the essence of the behavioral protocol defined by a given component interaction automaton. As shown in Section 3, when using the perspective of an external observer, only the output $(Store, ship, -)$ remains in the interface of $\mathcal{R}_{\emptyset}^{\{ship\}}$, a significant improvement with respect to the original complexity of $\mathcal{S}_{\emptyset}^{\{ship\}}$.



(b) $A\text{-}R_{\{d\}}^{\{c\}}$

Fig. 4. The weakly-bisimular component interaction automata $A_{\{d\}}^{\{c\}}$ and $A\text{-}R_{\{d\}}^{\{c\}}$.

Unfortunately, there are also situations in which partition refinement can eliminate information, which, in itself, can be viewed vital for the understanding of the interactive behavior of a component interaction automaton. Consider, for example, the two automata $A_{\{d\}}^{\{c\}}$ and $A\text{-}R_{\{d\}}^{\{c\}}$, as shown in Figure 4. Both are weakly-bisimular, $A_{\{d\}}^{\{c\}}$, however, contains a subgraph that produces a condition similar to the *small-world effect* [17]. In particular, the states $q0q0$ and $q1q1$ in automaton $A_{\{d\}}^{\{c\}}$ form a *synchronization clique* generating a distinct regular sublanguage,

$L_{q0q0} = \{(ab)^n | n \geq 0\} \cup \{b(ab)^m | m \geq 0\}$, of synchronization paths, which can originate from any clique state and terminate in the designated state $q0q0$. The prefix strings emerging from this sublanguage define a pre-condition that determines, when the transitions $(r0, (-, d, A2), r1)$ and $(r0, (A1, c, -), r2)$ can actually occur in the reduced automaton $A\text{-}R^{\{c\}}_{\{d\}}$.

Definition 4.1 Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be a component interaction automaton and $X \in \mathcal{X}$ be a set of equivalence classes up to weak bisimulation for $\mathcal{C}$. Then a synchronization clique is a non-empty directed graph (V, E), where $V \subseteq Q$ is a set of clique states and $E \subseteq \delta$ is a set of internal synchronizations $(q, (n, a, n'), p)$ with $q, p \in V$ and $q \neq p$, if there exists $P \in X$ such that $q, p \in P$.

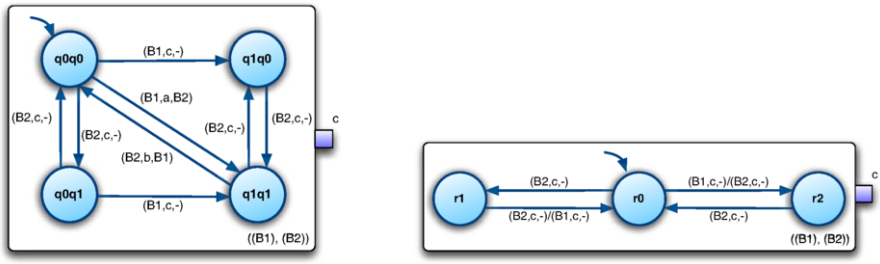
A synchronization clique appears, when partition refinement creates new *reflexive* internal synchronizations due to mapping the endpoints of these transitions onto the same equivalence class. By default, we can ignore preexisting reflexive internal synchronizations, as they can occur, interleaving, in any order. However, the newly formed reflexive internal synchronizations are of a different kind, as their non-reflexive originals encode a specific partial order over internal synchronizations. This property is lost in the refinement process. We can, however, recover this information through the notion of action prefixes. For example, in $A\text{-}R^{\{c\}}_{\{d\}}$ the transition $(r0, (-, d, A2), r1)$ can occur only after a, possibly empty, sequence of internal synchronizations over actions drawn from the alphabet $\{a, b\}$, captured by the prefix $[b](ab)^*$ that reifies the required synchronization paths to arrive in state $q0q0$ from synchronization clique $\{q0q0, q1q1\}$. In other words, the internal synchronizations have disappeared in automaton $A\text{-}R^{\{c\}}_{\{d\}}$, but we can use the prefix $[b](ab)^*$ to reenforce the existing pre-condition for the occurrence of transition $(r0, (-, d, A2), r1)$ in automaton $A\text{-}R^{\{c\}}_{\{d\}}$. That is, $(r0, (-, d, A2), r1)$ can occur immediate, after a single b, or after a sequence of paired a's and b's possibly preceded by a leading b.

The presence of synchronization cliques can cause even more worries, as illustrated in Figure 5. The automaton $B^{\{c\}}_{\emptyset}$ is defined as the composition of the following two automata $B1$ and $B2$ [2]:

$$
\begin{aligned}
B1 = (&\{q_0, q_1\}, \{a, b, c\},\\
&\{(q_0, (B1, a, -), q_1), (q_0, (B1, c, -), q_1), (q_1, (-, b, B1), q_0)\},\\
&\{q_0\}, (B1))\\
B2 = (&\{q_0, q_1\}, \{a, b, c\},\\
&\{(q_0, (-, a, B2), q_1), (q_0, (B2, c, -), q_1), (q_1, (B2, c, -), q_0), (q_1, (B2, b, -), q_0)\},\\
&\{q_0\}, (B2))
\end{aligned}
$$

The composition of $B1$ and $B2$, the automaton $B^{\{c\}}_{\emptyset}$, yields also a synchronization clique generating two sublanguages $L_{q0q0} = \{(ab)^n | n \geq 0\} \cup \{b(ab)^m | m \geq 0\}$ and $L_{q1q1} = \{a(ba)^n | n \geq 0\} \cup \{(ba)^m | m \geq 0\}$. Moreover, the reduction of au-

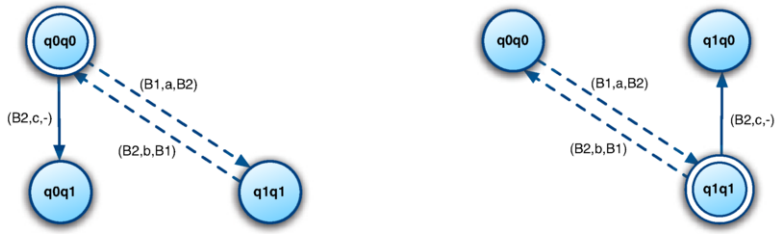
[2] These automata have been especially designed to reproduce an effect, which we have observed in many system specifications that we have analyzed over time [13].



(b) $B\text{-}R_{\emptyset}^{\{c\}}$

Fig. 5. The weakly-bisimular component interaction automata $B_{\emptyset}^{\{c\}}$ and $B\text{-}R_{\emptyset}^{\{c\}}$.

tomaton $B_{\emptyset}^{\{c\}}$ results in a non-deterministic automaton (cf. Figure 5(b)). It is transition $(q_0, (B1, c, -), q_1)$ of $B1$ that enables this phenomenon, not the flip-flop between automaton $B2$'s states over c. Fortunately, the notion of action prefixes provide us with the means to disambiguate the conflicting transitions. In particular, $(r0, (B2, c, -), r1)$ can only occur after a synchronization sequence $[b](ab)^*$, whereas $(r0, (B2, c, -), r2)$ is enabled by $[a](ba)^*$. The prefixes $[b](ab)^*$ and $[a](ba)^*$ capture the possible corresponding reified synchronization paths within automaton $B_{\emptyset}^{\{c\}}$ induced by synchronization clique $\{q0q0, q1q1\}$, as illustrated in Figure 6.



(a) Prefix $[b](ab)^*$.

(b) Prefix $[a](ba)^*$.

Fig. 6. The prefix-giving interaction sequences in $B_{\emptyset}^{\{c\}}$.

Definition 4.2 Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be a component interaction automaton, Σ be the induced alphabet of $\mathcal{C}$, and (V, E) be a synchronization clique in $\mathcal{C}$. Then a finite action prefix generator is a quadruple $\mathcal{C}_P = (V, \overline{Act}, \overline{E}, q_P)$, where

- V is the set of clique states,
- $\overline{Act} = \{a | (q, (n, a, n'), p) \in E\}$ is the prefix alphabet,
- $\overline{E} = \{(q, a, p) | (q, (n, a, n')p) \in E\}$ is the prefix transition function,
- $q_P \in V$ is a prefix state, if there is $l \in \Sigma$ such that $q_P \xrightarrow{l} p' \in \delta$, $p' \notin V$, and

all states in V are start states. We write $A_P[q]$ to denote the action prefix generator for prefix state q. If W_P is the set of all prefix strings that $A_P[q]$ accepts in q, we say that W_P is the action prefix language of $A_P[q]$ and write $L(A_P[q]) = W_P$.

An action prefix generator simultaneously explores all possible paths in a syn-

chronization clique in order to distill the required action prefixes for a given prefix state. From a technical point of view, a prefix generator iterates over all clique states in (V, E) and constructs for each a finite state machine whose language is the union of all accepted action prefixes for a given prefix state. For example, in automaton $B_{\emptyset}^{\{c\}}$, both states in the synchronization clique $\{q0q0, q1q1\}$ are prefix states and the generated languages are $L(A_P[q0q0]) = \{b^{0:1}(ab)^n | n \geq 0\}$ and $L(A_P[q1q1]) = \{a^{0:1}(ba)^m | m \geq 0\}$, which we denote by the action prefixes $[b](ab)^*$ and $[a](ba)^*$. On the other hand, state $q1q1$ in automaton $A_{\{d\}}^{\{c\}}$ (cf. Figure 4(a)) is not a prefix state and we, therefore, obtain only a prefix for state $q0q0$ (*i.e.*, $[b](ab)^*$).

Definition 4.3 Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be component interaction automaton, $q \xrightarrow{l} q' \in \delta$, and $A_P[q] = \alpha$ be an action prefix, then $q \xrightarrow{/\alpha/l} q'$ is a α-prefixed transition, with

$$/\alpha/l = \begin{cases} (-, /\alpha/a, n) & \text{if } l = (-, a, n), \\ (n, /\alpha/a, -) & \text{if } l = (n, a, -), \text{ and} \\ (n_1, /\alpha/a, n_2) & \text{if } l = (n_1, a, n_2). \end{cases}$$

Returning to the reduced automaton $B\text{-}R_{\emptyset}^{\{c\}}$ (cf. Figure 5(b)), we can obtain $B\text{-}R'^{\{c\}}_{\emptyset}$, a new deterministic automaton, by applying the generated prefixes to the respective transitions:

$$\begin{aligned} B\text{-}R'^{\{c\}}_{\emptyset} = (&\{r_0, r_1, r2\}, \{a, b, c\}, \\ &\{(r_0, (B2, /[b](ab)^*/c, -), r_1), (r_0, (B2, /[a](ba)^*/c, -), r_2), \ (r_0, (B1, /[b](ab)^*/c, -), r_2), \\ &\ (r1, (B2, c-), r0), (r_1, (B1, c, -), r_0), (r2, (B2, c, -), r0)\}, \\ &\{r_0\}, ((B1), (B2)) \end{aligned}$$

$B\text{-}R'^{\{c\}}_{\emptyset}$ is, naturally, not weakly-bisimular to $B\text{-}R_{\emptyset}^{\{c\}}$, as prefixed transitions produce a different behavior. However, we can restore bisimularity by erasing the added prefixes. We can think of prefixes as *types* [4], or more precisely *sequence types* [23], that explicitly record interaction constraints in a reduced component interaction automaton.

The composition of action prefixes is defined in the usual way. The composition of a refined automaton containing prefixed transitions with another automaton and the successive refinement may yield new action prefixes that have to be incorporated into the final result. We use the regular *concatenation* operation to built composite action prefixes. For example, if an existing action prefix $[f](ef)^*$ needs to be prefixed by $[b](ab)^*$, then the newly composed action prefix becomes $[f](ef)^*[b](ab)^*$. That is, before an interaction prefixed with $[f](ef)^*[b](ab)^*$ can occur, the corresponding component interaction automata must have performed a, possibly empty, sequence of internal synchronizations over actions f and e, followed by a, possibly empty, sequence of internal synchronizations over actions b and a.

Finally, we need to incorporate the prefix mechanism into the composition rule (cf. Definition 2.3) for new synchronizations, $\delta_{NewSync}$. We use '?' to indicate a prefix associated with an input action and '!' to denote a prefix originating from an output action.

Definition 4.4 Let $q_1 \xrightarrow{(n_1,/\alpha_1/a,-)} q_1'$ and $q_2 \xrightarrow{(-,/\alpha_2/a,n_2)} q_2'$ be two prefixed transitions that synchronize according to Definition 2.3. Then $q \xrightarrow{(n_1,/?\alpha_1!\alpha_2/a,n_2)} q'$ is the resulting prefixed synchronization transition, where $?\alpha_1!\alpha_2$ is an atomic directional prefix.

Two component interaction automata can synchronize through matching complementary structured labels. These labels may, in turn, occur prefixed as result of a previous refinement of the underlying automata. These prefixes, however, cannot simply be concatenated as regular prefixes. Each prefix encodes either an input constraint or an output constraint, which we must retain both. On the surface, this appears cumbersome, but we facilitate the use of directional prefixes by considering them *atomic*, as if they were plain actions. We only require, as for all prefixes, that they are well-formed, that is, they are regularly composed of elements from the set, Act, of actions.

5 Discussion

We cannot underestimate the computational needs for the construction of a composite component interaction automaton. For example, even for a relatively small system consisting of components with no more than 4 states, the resulting product automaton requires easily in excess of 16,000 states with approx. 880,000 transitions and can take more than 6 hours to compute on a PC equipped with a 2.2 GHz dual-core processor and 2GB of main memory [13].

To study different means for an effective specification and construction of component interaction automata, we have developed an experimental composition framework for *Component Interaction Automata* in PLT-Scheme [21] that provides modular support for the specification, composition, and refinement of component interaction automata [13]. All analysis and transformation functions in the system are timed and can be controlled by a variety of parameters to fine-tune the induced operational semantics of an operation. The system also generates information about frequencies and distributions of states and transitions within composite automata, data that allows for an independent statistical analysis of the effects of composition and refinement.

One of the rather unexpected findings, while conducting experiments in our composition framework, is the existence of synchronization cliques in component interaction automata. To explain their presence, we have adopted some of the terminology that has been developed in network theory in order to characterize properties of complex networks [17]. Of particular interest are small-world networks that have been discovered in an astonishing number of natural phenomena, but also

in software systems. Potanin et al. [22] have studied Java programs and detected *power laws* in object graphs indicating that object-oriented systems form *scale-free networks* [17]. A consequence of the existence of power laws in object-oriented systems is that there is no *typical* size to objects [22]. We find a similar property in component interaction automata.

But there remains a curiosity as to why synchronization cliques exist. We do not find a similar phenomenon in process-based models [9,12,15,16,24]. There is, however, a difference in the way internal synchronizations are represented. Process-based formalisms use a special symbol, τ, to denote the handshake between two matching, interacting processes. The synchronization of processes takes place internally. From an external observer's point of view, we notice the occurrence of a process synchronization through a delay between adjacent interactions with the environment. Milner [16] calls τ a *perfect action*, which arises from a pair of complementary input- and output-actions. What makes τ special is the observable equivalence between a sequence $P_1 \xrightarrow{\tau} P_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_n$ of process synchronizations and a single synchronization $P_1 \xrightarrow{\tau} P_n$. A similar concept does not exist in *Component Interaction Automata* and its predecessors *I/O Automata* and *Interface Automata*. We cannot equate a sequence of internal synchronizations in a component interaction automaton with a single action. First, such an abstraction would ignore the inherent partial order defined by specific synchronization paths and second, there exists no designated action in the *Component Interaction Automata* formalism that can subsume several synchronization paths under one umbrella. Moreover, the precise sequence of internal synchronization paths conveys a valuable information. For example, the *Store* will only issue the action $(Store, ship, -)$ after a successful interaction with the *Bank* to redeem the payment voucher (cf. Figure 2). This knowledge is vital for the understanding of the behavior of the whole e-commerce system $\mathcal{S}_{\emptyset}^{\{ship\}}$.

We have chosen regular expressions like $[b](ab)^*$ rather than introducing fresh action labels to denote action prefixes in order to make pre-conditions to interactions as explicit as possible. This works well for simple prefixes. Experiments have shown, however, that action prefixes can grow in complexity rather quickly, rendering this structural technique unwieldy. We can envision a *nominal* approach to the specification of action prefixes in which we assign each action prefix a unique identifier and add a corresponding lookup table to the specification of the component interaction automata in question.

Finally, the outcome of partition refinement can be improved even further, if we erase the information about the underlying composition hierarchy by making the analyzed component *primitive* before refinement [13]. The composition of multiple instances of the same component can produce identical sub-structures in the resulting composite automaton. However, the unique component identifiers used to disambiguate shared actions prevent partition refinement from simplifying common sub-structures into a single, unifying one. We can overcome this difficulty by creating a fresh image of a given component interaction automaton in which all component names are the same. We will lose, though, the information, which

particular sub-component participates in an actual occurring interaction with the environment.

6 Conclusion and Future Work

In this paper we have discussed some of the effects that partition refinement can produce when we apply this state space reduction technique to *Component Interaction Automata* specifications. We use weak bisimulation as underlying equivalence relation to drive the refinement process. From an external observer's point of view, weak bisimulation yields a means to hide internal intra-component synchronizations.

While a corresponding implementation of partition refinement for *Component Interaction Automata* specifications is feasible and effective, its application has revealed a specific property of component interaction automata that mandates an additional analysis to recover pre-conditions encoded in so-called *synchronization cliques*. A synchronization clique is a subgraph of internal intra-component synchronizations that define guards for component interactions with the environment. Partition refinement removes synchronization cliques from the specification of given component interaction automaton. But, in this paper, we have presented a workable solution to restore pre-conditions in reduced automata, when necessary.

We are only beginning to understand the emerging properties of software systems in general and component-based software systems in particular. There is sufficient evidence for the existence of small-world networks in software. To further our knowledge in this area, in future work we aim at studying network effects in component interaction automata specifications. In particular, we seek to explore possibilities to (i) predict the presence of synchronization cliques, (ii) estimate the reduction ratio, and (iii) use frequency distributions to monitor evolutionary changes in component interaction automata specifications.

References

[1] Beugnard, A., J.-M. Jézéquel, N. Plouzeau and D. Watkins, *Making Components Contract Aware*, IEEE Computer **32** (1999), pp. 38–45.

[2] Brim, L., I. Černá, P. Vařeková and B. Zimmerova, *Component-Interaction Automata as a Verification-Oriented Component-Based System Specification*, SIGSOFT Software Engineering Notes **31** (2006), pp. 1–8.

[3] Broy, M., *A Core Theory of Interfaces and Architecture and Its Impact on Object Orientation*, in: R. H. Reussner, J. A. Stafford and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, LNCS 3938 (2004), pp. 26–47.

[4] Cardelli, L., *Type Systems*, in: *Handbook of Computer Science and Engineering*, CRC Press, 1997 pp. 2208–2236.

[5] Černá, I., P. Vařeková and B. Zimmerova, *Component Substitutability via Equivalencies of Component-Interaction Automata*, Electronic Notes in Theoretical Computer Science **182** (2007), pp. 39–55.

[6] de Alfaro, L. and T. A. Henzinger, *Interface Automata*, in: V. Gruhn and A. M. Tjoa, editors, *Proceedings ESEC/FSE 2001* (2001), pp. 109–120.

[7] de Alfaro, L., T. A. Henzinger and M. Stoelinga, *Timed Interfaces*, in: S.-V. A. L. and J. Sifakis, editors, *Proceedings of 2nd International Conference on Embedded Softare (EMSOFT 2002)*, LNCS 2491 (2002), pp. 108–122.

[8] Habib, M., C. Paul and L. Viennot, *Partition Refinement Techniques: An Interesting Algorithmic Tool Kit*, International Journal of Foundations of Computer Science **10** (1999), pp. 147–170.

[9] Hermanns, H., "Interactive Markov Chains: The Quest for Quantified Quality," LNCS 2428, Springer, Heidelberg, Germany, 2002.

[10] Hopcroft, J. E., R. Motwani and J. D. Ullman, "Automata Theory, Languages, and Computation," Pearson Education, 2007, 3rd edition.

[11] Lee, E. A. and Y. Xiong, *System-Level Types for Component-Based Design*, in: T. A. Henzinger and C. M. Kirsch, editors, *Proceedings of 1st International Workshop on Embedded Software (EMSOFT 2001)*, LNCS 2211 (2001), pp. 237–253.

[12] Lumpe, M., "A π-Calculus Based Approach to Software Composition," Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics (1999).

[13] Lumpe, M., L. Grunske and J.-G. Schneider, *Interface Automata*, in: M. R. V. Chaudron and C. Szyperski, editors, *CBSE 2008*, LNCS 5282 (2008), pp. 130–145.

[14] Lunch, N. A. and M. R. T. Tuttle, *Hierarchical Correctness Proofs for Distributed Algorithms*, in: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, 1987, pp. 137–151.

[15] Mateescu, R., P. Poizat and G. Salaün, *Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques*, in: *Proceedings of ICSOC 2008*, LNCS 5364 (2008), pp. 84–99.

[16] Milner, R., "Communication and Concurrency," Prentice Hall, 1989.

[17] Newman, M. E. J., *The Structure and Function of Complex Networks*, SIAM Review **45** (2003), pp. 167–256.

[18] Paige, R. and R. E. Tarjan, *Three Partition Refinement Algorithms*, SIAM Journal on Computing **16** (1987), pp. 973–989.

[19] Park, D., *Concurrency and Automata on Infinite Sequences*, in: P. Deussen, editor, *5th GI Conference on Theoretical Computer Science*, LNCS 104 (1981), pp. 167–183.

[20] Pistore, M. and D. Sangiorgi, *A Partition Refinement Algorithm for the π-Calculus*, Information and Computation **164** (2001), pp. 264–321.

[21] PLT Scheme, "v372," http://www.plt-scheme.org (2008).

[22] Potanin, A., J. Noble, M. R. Frean and R. Biddle, *Scale-Free Geometry in OO Programs*, Commun. ACM **48** (2005), pp. 99–103.

[23] Puntigam, F., *Coordination Requirements Expressed in Types for Active Objects*, in: M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, LNCS 1241 (1997), pp. 367–388.

[24] Schneider, J.-G. and O. Nierstrasz, *Components, Scripts and Glue*, in: L. Barroca, J. Hall and P. Hall, editors, *Software Architectures – Advances and Applications*, Springer, 1999 pp. 13–25.

[25] Seco, J. C. and L. Caires, *A Basic Model of Typed Components*, in: E. Bertino, editor, *Proceedings of ECOOP 2000*, LNCS 1850 (2000), pp. 108–128.

[26] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," Addison-Wesley / ACM Press, 2002, Second edition.

[27] ter Beek, M. H., C. A. Ellis, J. Kleijn and G. Rozenberg, *Synchronizations in Team Automata for Groupware Systems*, Computer Supported Cooperative Work **12** (2003), pp. 21–69.