FI MU

Faculty of Informatics
Masaryk University Brno

Component-Interaction Automata Modelling Language

by

Ivana Černá
Pavlína Vařeková
Barbora Zimmerova

FI MU Report Series FIMU-RS-2006-08

October 2006

Component-Interaction Automata Modelling Language

Ivana ČERNÁ, Pavlína VAŘEKOVÁ, Barbora ZIMMEROVA*

Faculty of Informatics, Masaryk University

Brno, Czech Republic

`{cerna,xvareko1,zimmerova}@fi.muni.cz`

October 15, 2006

Abstract

The paper introduces a *Component-interaction automata* language, which was designed for modelling of component interactions in hierarchical component-based software systems. The language supports modelling of important interaction attributes of such systems, and hence provides a rich base for further application of formal methods. Component-interaction automata provide a flexible form of component composition which can respect architectural assembly of the system, communication mechanisms, and other specifics of component-based systems. This allows the language to capture interactions in many different kinds of component-based systems (built on different component models for instance). This paper provides a detailed study of the language including discussion of its practical application and comparison with related work. We intend to use this language as a framework for verifying coordination errors, checking of reconfiguration correctness, and formal analysis of component interactions in component-based systems, which is our ongoing and future work.

*The work has been supported by the grant No. 1ET400300504.

Contents

1 Introduction

Component-based and service-oriented development [15], developing software systems through hierarchical composition of autonomous components providing services to others, is becoming a standard practice in development of large-scale software systems. Although this technique offers great benefits of reusing prefabricated components, increasing flexibility of software products and reducing development costs, it also poses serious questions on faultlessness and correctness of such systems. The main aspect that makes the correctness task difficult is that the components are usually developed by third parties with no knowledge of the environment they will become a part of. Hence even if we assume that each individual component in the system is correct, there is no guarantee that the components will cooperate correctly to deliver expected results. We refer to this as interaction correctness.

For further investigation of interaction correctness, proper specification of component interactions in assembled system is indispensable. Choice of an appropriate specification language should be done very carefully with respect to the following aspects. Firstly, the language should be flexible enough to be usable for component-based systems built on various component models (using different communication mechanisms). That then enables introduction of a set of formal methods that are applicable to systems built on different platforms. Secondly, the language should model the reality as closely as possible capturing important interaction attributes. Lastly, it should be of manageable complexity to allow us to build and analyse models of systems that are of significant size.

This paper presents such a language, named *Component-interaction automata*, which was designed with the primary purpose oriented to component-based systems and their specifics. The language is a simple combination of inspiring features of several existing languages with some additional constructs, which allow the language to fit the needs described above and still remain simple and general with many interesting applications. The language concentrates on capturing of important interaction aspects of systems (in what order components intend to communicate their actions, what is their architecture like) and update such information during component assembly as well (which components synchronized on particular actions, how the architectural structure changed). One of the essential differences to related languages (discussed in Section 4) is that Component-interaction automata language model component assembly using a flexible form of composition, which can be parametrized by a set of interactions that are feasible/infeasible in the system. This allows the language to model several variations of component-based systems (based on different component models for instance).

The paper is organized as follows. Firstly, in Section 2, the Component-interaction automata language is defined and basic examples of its use are provided. Practical application of the language is studied in Section 3 where we present several modelling issues that appear in component-based systems. Section 4 provides a discussion of related work and in Section 5 we conclude with stating our future work.

2 Component-interaction automata language

Component-interaction automata (first presented in [8]) are a specification language for modelling of component interactions in hierarchical component-based software systems, which are usually connected to a particular component model. For this reason, Component-interaction automata are very general and support modelling of component interactions in component-based systems build on various component models. The generality of the language follows from two things. First, the Component-interaction automata language does not explicitly associate action names with interfaces/services/provisions/requirements/requests/responses/events, which allows the designers to make the association themselves. Second, the language provides a flexible form of composition that can be fixed according to a specific component model (type of communication, synchronization, blocking/non-blocking strategy, etc.). In this manner, Component-interaction automata can be instantiated to a particular component model by fixing the composition operator and semantics of actions. The main reason for this generality is the following. We use the language as a formal basis for analysis and verification of component interactions in component-based systems. Hence the more systems can be modelled with it, the more systems can be formally analysed using the (uniform) set of verification and analytical techniques we propose.

Let us now summarize what intuition the language has about the component-based systems. The language regards component-based systems as systems that are assembled from autonomous (COTS) components, possibly in a hierarchical manner. Each component-interaction automaton represents a component in the system that can be either primitive or composite. A component is regarded as an encapsulated unit which interacts with the environment solely through its interfaces (possibly by provided/required services, messages, events, etc.). A component is designed as an autonomous unit. Hence it knows nothing about the environment it will become a part of. It therefore does not specify the components which should serve its requests. This is determined by other means – by bindings among components for instance. Automata (representing components) interact with their environment through actions. The designer should decide whether an action represents a service, part of a service or something else. If the same action name is used in two components, in one as an input and in the other as an output, it marks a point on which the components *may* communicate (a provided and required service of the same type for instance). Only two components may synchronize in this way (with respect to a client–supplier principle). However, more sophisticated synchronization strategies can be realized by synchronizing components or connectors (modelled as component-interaction automata also). A primitive component cannot use the same action name for two distinct services as we would not be able to distinguish between them in the automaton.

2.1 Definition of a component-interaction automaton

We focus on modelling of hierarchical component-based systems. Hence a component-interaction automaton of a system captures information about the set of primitive components (which make up the system) and the way in which they are assembled to form the system. As it is important to know what the primitive components of the system are, each primitive component in the automaton is associated with a unique name. The component names are also used to remember which components communicated on particular actions. In the case of Component-interaction automata, the names are natural numbers[1].

In Component-interaction automata, the information about the composition order that was used to compose a particular high-level (composite) component is preserved within a *hierarchy of component names*. Every hierarchy of component names is an n-tuple (where $n \in \mathbb{N}$) whose items correspond to natural numbers or lower level hierarchies of component names that can be either primitive or compound. A primitive hierarchy of component names for a component with a numerical name 1 can take form of (1) or $(((1)))$ for instance, and a compound hierarchy of component names for a composition of components 1 and 2 can take form of $(((1)),(2))$ or $(2,1)$ for instance.

[1]The set of possible component names is selected as $\mathbb{N}$ for the sake of simplicity. Any other set of names (not containing special symbols like parentheses and commas) could play the same role.

Definition 2.1. *A* hierarchy of component names *is an n-tuple* $H = (H_1, \dots, H_n)$, $n \in \mathbb{N}$, *of one of the following forms; S_H denotes the set of component names corresponding to* H.

- *The first case is that $H_1, \dots, H_n$ are pairwise different natural numbers; then $S_H = \bigcup_{i=1}^{n} \{H_i\}$.*
- *The second case is that $H_1, \dots, H_n$ are hierarchies of component names where $S_{H_1}, \dots, S_{H_n}$ are pairwise disjoint; then $S_H = \bigcup_{i=1}^{n} S_{H_i}$.*

A set of hierarchies of component names is denoted $\mathcal{H}$.
A hierarchy of component names $H \in \mathcal{H}$ is primitive *iff* $|S_H| = 1$.

Now we can proceed to the definition of a component-interaction automaton which is a labelled transition system with structured labels and a hierarchy of names of the component whose composition the automaton represents.

Definition 2.2. *A* component-interaction automaton *(or* CI automaton *for short) is a 5-tuple* $\mathcal{C} = (Q, \mathsf{Act}, \delta, I, H)$ *where*

- Q *is a finite set of states,*
- Act *is a finite set of actions,*
  $\Sigma = ((S_H \cup \{-\}) \times \mathsf{Act} \times (S_H \cup \{-\})) \setminus (\{-\} \times \mathsf{Act} \times \{-\})$ *is a set of* labels,
- $\delta \subseteq Q \times \Sigma \times Q$ *is a finite set of labelled transitions,*
- $I \subseteq Q$ *is a nonempty set of initial states and*
- $H \in \mathcal{H}$ *is a hierarchy of component names.*

The labels have semantics of input, output, or internal, based on their structure. In the triple, the middle item represents an action name, the first item represents a name of the component that outputs the action, and the third item represents a name of the component that inputs the action. Examples of three CI automata are in Figure 2, and are discussed in Example 2.5. Before we proceed to the example, we complete the set of definitions with the definition of a path in a CI automaton, and a notation for sets of labels in a CI automaton.

Definition 2.3. *A* path *of a CI automaton $\mathcal{C} = (Q, \mathsf{Act}, \delta, I, H)$ is an alternating sequence of states and labels given by δ that is either infinite, or is finite in case that it ends with a state from which there is no transition in δ. The set of all paths of a CI automaton $\mathcal{C}$ is denoted* $\mathsf{Path}(\mathcal{C})$. *The set of all finite prefixes of paths from $\mathsf{Path}(\mathcal{C})$ that end with a state is denoted* $\mathsf{FinPath}(\mathcal{C})$.

Notation 2.4. For a given CI automaton $\mathcal{C} = (Q, \mathsf{Act}, \delta, I, H)$ we denote

- $\mathcal{L}_{\mathcal{C}} = \{l \mid \exists\ q_0, l_0, \dots, q_{k-1}, l_{k-1}, q_k \in \mathsf{FinPath}(\mathcal{C}) : q_0 \in I\ \wedge\ l_{k-1} = l\}$
  the set of all labels reachable in $\mathcal{C}$,

- $\mathcal{L}_{inp,\mathcal{C}} = \mathcal{L}_{\mathcal{C}} \cap \{(-, a, n_2) \mid a \in \mathit{Act},\ n_2 \in \mathbb{N}\}$
  the set of all input labels reachable in $\mathcal{C}$ (a component n_2 inputs an action a),
- $\mathcal{L}_{out,\mathcal{C}} = \mathcal{L}_{\mathcal{C}} \cap \{(n_1, a, -) \mid a \in \mathit{Act},\ n_1 \in \mathbb{N}\}$
  the set of all output labels reachable in $\mathcal{C}$ (a component n_1 outputs an action a),
- $\mathcal{L}_{int,\mathcal{C}} = \mathcal{L}_{\mathcal{C}} \cap \{(n_1, a, n_2) \mid a \in \mathit{Act},\ n_1, n_2 \in \mathbb{N}\}$
  the set of all internal labels reachable in $\mathcal{C}$ (n_1 and n_2 synchronize on a),
- $\mathcal{L}_{ext,\mathcal{C}} = \mathcal{L}_{inp,\mathcal{C}} \cup \mathcal{L}_{out,\mathcal{C}} = \mathcal{L}_{\mathcal{C}} \setminus \mathcal{L}_{int,\mathcal{C}}$
  the set of all external (input and output) labels reachable in $\mathcal{C}$.

The structure of symbols shows that at most two components participate in any transition. It is a natural form of component communication according to a client–supplier principle. However, if it becomes necessary to address a multi-way synchronization, the model could be naturally extended to *Multi Component-interaction automata* with labels $(\mathsf{A}, a, \mathsf{B})$ where A is a set of components sending an action a and B a set of components receiving a.

Example 2.5. *Let us consider a simple example shown in Figure 1 and Figure 2 capturing a* WordProcessor*-like application consisting of three sub-components* Document1*,* Document2 *and* SpellChecker*. Figure 1 shows only the component architecture of the application in UML 2.0*[2]*, whereas the description in Figure 2 explains the behaviour of the components as CI automata.*

According to the description, both Document1 *(automaton $\mathcal{C}_1$) and* Document2 *(automaton $\mathcal{C}_2$) represent primitive components with numerical names* 1 *and* 2*, respectively. Each of them first requests spell checking (sends an action* $\mathtt{check.req}$*) and waits for a response with a result afterwards (receives an action* $\mathtt{check.resp}$*). On the other hand,* SpellChecker *(automaton $\mathcal{C}_3$) is a composite component that has two primitive sub-components with numerical names* 3 *and* 4*. In such case, all external actions of* SpellChecker *are in fact delegated to the primitive sub-components that perform all the functionality. In particular, the spell checking is realized by a component* 3*, which after receiving a request for checking a part of a text (action* $\mathtt{check.req}$*), may ask a component* 4 *(in several loops) to find some words (actions* $\mathtt{find.req}$*,* $\mathtt{find.resp}$*), and then returns the result (action* $\mathtt{check.resp}$*).*

[2] Any other architecture description language could be used. In this example, we have chosen UML because it is widely known and accepted.

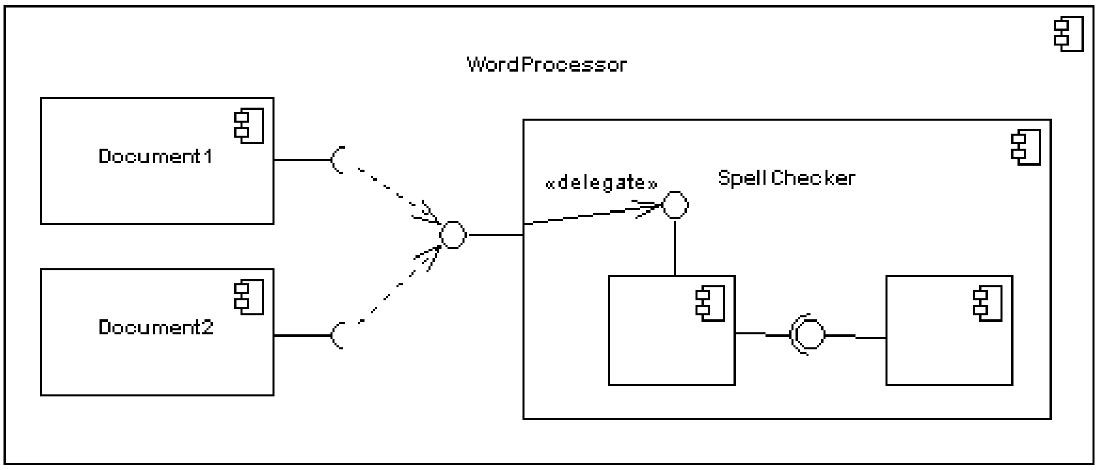


Figure 1: Component model of a simple application

$$\mathcal{C}_1 : \quad \begin{tikzcd} {} \arrow[r] & 0 \arrow[r, shift left, "{(1\,,\mathrm{check.req}\,,-)}"] & 1 \arrow[l, shift left, "{(-\,,\mathrm{check.resp}\,,1)}"] \end{tikzcd}$$

Hierarchy of component names: (1)

$$\mathcal{C}_2 : \quad \begin{tikzcd} {} \arrow[r] & 0 \arrow[r, shift left, "{(2\,,\mathrm{check.req}\,,-)}"] & 1 \arrow[l, shift left, "{(-\,,\mathrm{check.resp}\,,2)}"] \end{tikzcd}$$

Hierarchy of component names: (2)

$$\mathcal{C}_3 : \quad \begin{tikzcd} {} \arrow[r] & 0 \arrow[r, shift left, "{(-\,,\mathrm{check.req}\,,3)}"] & 1 \arrow[l, shift left, "{(3\,,\mathrm{check.resp}\,,-)}"] \arrow[r, shift left, "{(3\,,\mathrm{find.req}\,,4)}"] & 2 \arrow[l, shift left, "{(4\,,\mathrm{find.resp}\,,3)}"] \end{tikzcd}$$

Hierarchy of component names: $(3,4)$

Figure 2: CI automata $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$

The simplest form of a CI automaton according to the hierarchy of component names is an automaton representing one individual component only (with a primitive hierarchy of component names).

Definition 2.6. *A CI automaton* $\mathcal{C} = (Q, Act, \delta, I, H)$ *is* primitive *iff* H *is primitive.*

Example 2.7. *The automata* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *in Figure 2 are primitive as their hierarchies consist of one component only, the automaton* $\mathcal{C}_3$ *is not.*

In some cases it is useful to abstract from the inner hierarchy of a CI automaton and consider it as a primitive one to make the system less complex for further verification. For such a view on a CI automaton it suffice to replace all component names with a one unique name and change the hierarchy of component names.

Definition 2.8. *Let* $\mathcal{C} = (Q, Act, \delta, I, H)$ *be a CI automaton. Then a CI automaton* $\mathcal{C}' = (Q, Act, \delta', I, (n))$ *is* primitive to $\mathcal{C}$ *iff*

- $n \in \mathbb{N}$,
- $(q, (n, a, n), q') \in \delta'$ *iff* $\exists n_1, n_2 \in \mathbb{N} : (q, (n_1, a, n_2), q') \in \delta$,
- $(q, (-, a, n), q') \in \delta'$ *iff* $\exists n_2 \in \mathbb{N} : (q, (-, a, n_2), q') \in \delta$,
- $(q, (n, a, -), q') \in \delta'$ *iff* $\exists n_1 \in \mathbb{N} : (q, (n_1, a, -), q') \in \delta$.

Example 2.9. *Considering the automata depicted in Figure 2, both the automaton* $\mathcal{C}_1$ *is primitive to* $\mathcal{C}_2$*, and the automaton* $\mathcal{C}_2$ *is primitive to* $\mathcal{C}_1$*. An example of the CI automaton which is primitive to the automaton* $\mathcal{C}_3$ *is depicted in Figure 3 as an automaton* $\mathcal{C}_4$*.*

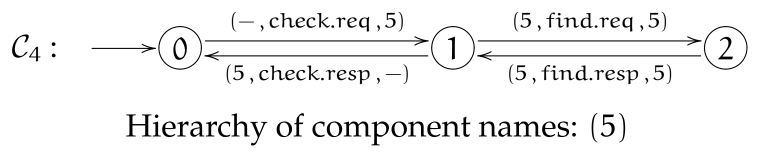


Figure 3: CI automaton $\mathcal{C}_4$

2.2 Composition of component-interaction automata

CI automata (a set of them) can be composed to form a higher level CI automaton. The language of Component-interaction automata allows us to compose any set of CI automata that have disjoint sets of component names. This assures that each primitive component in the assembled system has a unique name.

Notation 2.10. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a nonempty set of integers with $i_1 < \cdots < i_n$. Then for a set $\{H_i\}_{i\in\mathcal{I}}$ the symbol $(H_i)_{i\in\mathcal{I}}$ denotes the n-tuple $(H_{i_1}, H_{i_2}, \ldots, H_{i_n})$.

Definition 2.11. *Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$ be a set of CI automata. We say that the set $\mathcal{S}$ is* composable *iff $\mathcal{I} \subseteq \mathbb{N}$ is finite and $(H_i)_{i\in\mathcal{I}} \in \mathcal{H}$.*

We can now proceed to the automaton which results as a composition of the set of CI automata. The transition set of the composite automaton is defined over a complete transition space which represents all potentially feasible transitions of the system. The complete transition space for a set of CI automata consists of all transitions capturing that (1) one of the automata follows its original transition or (2) two automata synchronize on complementary transitions.

Notation 2.12. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a nonempty set of integers with $i_1 < \cdots < i_n$, and let Q_i be a set for each $i \in \mathcal{I}$. Then $\Pi_{i\in\mathcal{I}}Q_i$ denotes the set $\{(q_{i_1}, q_{i_2}, \ldots, q_{i_n}) \mid \forall j \in \{1, \ldots, n\} : q_{i_j} \in Q_{i_j}\}$. For any $j \in \mathcal{I}$, $proj_j$ denotes a function $proj_j : \Pi_{i\in\mathcal{I}}Q_i \to Q_j$ such that $proj_j((q_i)_{i\in\mathcal{I}}) = q_j$.

Definition 2.13. *Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$ be a composable set of CI automata. Then the* complete transition space *for $\mathcal{S}$ is $\Delta_{\mathcal{S}} = \Delta_{\mathcal{S},old} \cup \Delta_{\mathcal{S},new}$ where*

- $\Delta_{\mathcal{S},old} = \{(q, x, q') \mid q, q' \in \Pi_{i\in\mathcal{I}}Q_i,\ \exists j \in \mathcal{I} : [(proj_j(q), x, proj_j(q')) \in \delta_j\ \wedge\ \forall i \in (\mathcal{I} \setminus \{j\}) :\ proj_i(q) = proj_i(q')]\}$
- $\Delta_{\mathcal{S},new} = \{(q, (n_1, a, n_2), q') \mid q, q' \in \Pi_{i\in\mathcal{I}}Q_i\ \wedge\ \exists j_1, j_2 \in \mathcal{I},\ j_1 \neq j_2 : [(proj_{j_1}(q), (n_1, a, -), proj_{j_1}(q')) \in \delta_{j_1}\ \wedge (proj_{j_2}(q), (-, a, n_2), proj_{j_2}(q')) \in \delta_{j_2}\ \wedge\ \forall\, i \in (\mathcal{I} \setminus \{j_1, j_2\}) :\ proj_i(q) = proj_i(q')]\}$

Example 2.14. *The complete transition space for a set of CI automata $\{\mathcal{C}_i\}_{i\in\{1,2,3\}}$ in Figure 2 is given in Figure 4. For lucidity, an action* check.req *is shortened to* c*, action* check.resp *to* c'*, action* find.req *to* f*, and* find.resp *to* f'*, and every state $(q_1, q_2, q_3) \in Q_1 \times Q_2 \times Q_3$ is represented as a sequence $q_1q_2q_3$.*

The composition of a set of CI automata is defined in a flexible manner as a CI automaton over the set, and can take several forms for the same set of CI automata. In particular, the CI automaton over the set is defined as a product automaton whose transition set is a subset of the complete transition space. Thanks to this fact the resulting automaton may consist of only those transition that are really feasible in the system, according to the architectural assembly or type of synchronization for instance.

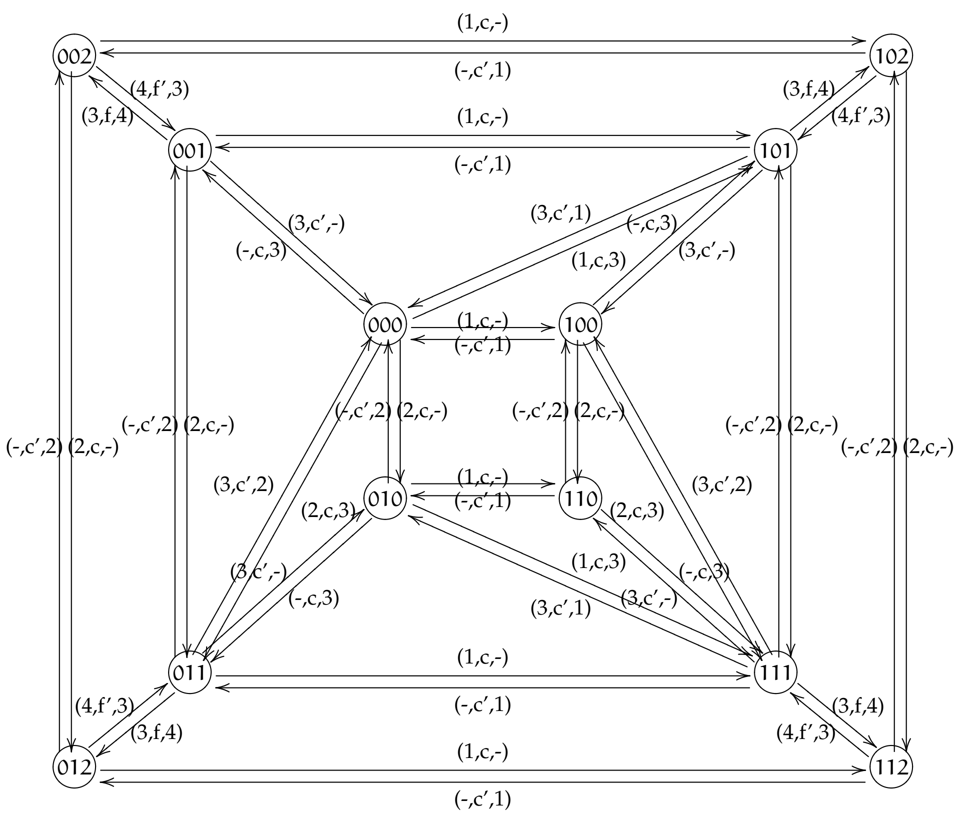


Figure 4: Complete transition space for $\{\mathcal{C}_i\}_{i\in\{1,2,3\}}$

Definition 2.15. *Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i \in \mathcal{I}}$ be a composable set of CI automata. Then $\mathcal{C} = (\, \Pi_{i\in\mathcal{I}} Q_i \,,\, \cup_{i\in\mathcal{I}} Act_i \,,\, \delta \,,\, \Pi_{i\in\mathcal{I}} I_i \,,\, (H_i)_{i\in\mathcal{I}} \,)$ is a* component-interaction automaton over *$\mathcal{S}$ iff $\delta \subseteq \Delta_{\mathcal{S}}$.*

The preceding definition provides the notion of what the composition of a set of CI automata into another CI automaton looks like. However, it does not determine the resulting automaton uniquely. The following definition introduces an operator (a set of operators in fact) that gives precision to the composition of a set of CI automata.

Definition 2.16. *Let T be a set of transitions, then $\otimes_T$ denotes a unary composition operator on composable sets of CI automata. If $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$ is a composable set of CI automata, then $\otimes_T \mathcal{S} = (\, \Pi_{i\in\mathcal{I}} Q_i \,,\, \cup_{i\in\mathcal{I}} Act_i \,,\, \Delta_{\mathcal{S}} \cap T \,,\, \Pi_{i\in\mathcal{I}} I_i \,,\, (H_i)_{i\in\mathcal{I}} \,)$.*

It directly follows from Definition 2.16 that the composite automaton $\otimes_T \mathcal{S}$ is again a *component-interaction automaton*, and moreover for any set of transitions T, it is a *component-interaction automaton over $\mathcal{S}$*.

In fact, the composition operator $\otimes_T$ defines the composite automaton in a way that it consists of only the transitions from the complete transition space that are part of T. Hence it is the set T that introduces the flexibility into the composition. This starts to be interesting when we realize that T can be defined using conditions that delimit the behaviour that is possible/impossible in the system. Such conditions allow us to characterize feasible transitions in the automaton and remove infeasible ones to model a real system (built on a particular component model) as closely as possible.

The choice of the set T can in fact determine a subset of operators and hence instantiate the operator to a particular component model. We illustrate one of such instantiations in the rest of this section. For a given set of (feasible) labels $\mathcal{F}$, let $T_{\mathcal{F}} = \{(q, x, q') \mid x \in \mathcal{F}\}$. This choice allows the composite automaton $\otimes_{T_{\mathcal{F}}} \mathcal{S}$ (where $\mathcal{S}$ is a set of CI automata) to provide only the transitions with labels from $\mathcal{F}$.

The set of operators $\otimes_{T_{\mathcal{F}}}$, denote it $\otimes^{\mathcal{F}}$, is weaker then $\otimes_T$ in a sense that for each $\mathcal{F}'$ we can express $\otimes^{\mathcal{F}'}$ as $\otimes_{T'}$ for some T', but not vice versa. Each operator $\otimes^{\mathcal{F}}$ chooses legal transitions only with respect to their labels. It does not regard the states. So it cannot analyse the transitions based on their position in automaton's execution. However, it is still quite strong and has several practical applications including component assembly modelling which allows synchronization of only those components that are connected by a binding in the system. It is possible thanks to the information about participants of communication that CI automata include in labels. We may choose $\mathcal{F}$ as the set of those labels that represent communication between the components that are connected by communicational binding in the system. As $\otimes^{\mathcal{F}}$ is of our high interest, we provide its precise definition here, even if it is just a special case of $\otimes_{T}$.

Definition 2.17. *Let $\mathcal{F}$ be a set of labels, then $\otimes^{\mathcal{F}}$ denotes a unary composition operator on composable sets of CI automata. If $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$ is a composable set of CI automata, then $\otimes^{\mathcal{F}}\mathcal{S} = (\Pi_{i\in\mathcal{I}}Q_i, \cup_{i\in\mathcal{I}}Act_i, \delta, \Pi_{i\in\mathcal{I}}I_i, (H_i)_{i\in\mathcal{I}})$ where $\delta = \{(q, x, q') \mid (q, x, q') \in \Delta_{\mathcal{S}} \wedge x \in \mathcal{F}\}$.*

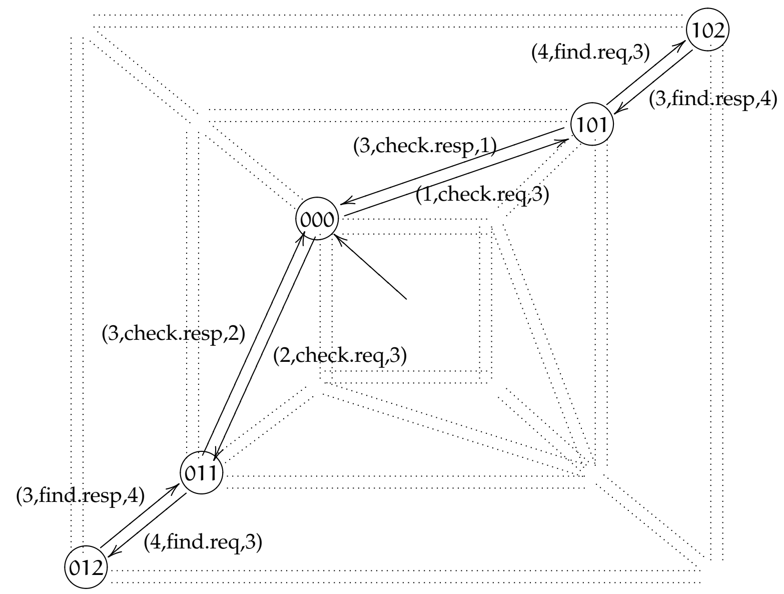
As the set $\mathcal{F}$ represents component assembly of the system, we require that it contains all internal labels of former automata (that are to be composed) since the assembly binds only external services of the components. It does not concern the former internal behaviour.

Definition 2.18. *We say that the automaton $\otimes^{\mathcal{F}}\{\mathcal{C}_i\}_{i\in\mathcal{I}}$ is* defined *iff $\{\mathcal{C}_i\}_{i\in\mathcal{I}}$ is a composable set of CI automata and $\mathcal{F} \supseteq \bigcup_{i\in\mathcal{I}} \mathcal{L}_{int,\mathcal{C}_i}$.*

When $\mathcal{F}$ is selected as a set of possible inner and inter-component communication (indicated by bindings in the system), $\otimes^{\mathcal{F}}$ models standard blocking (handshake) communication of components. If $\mathcal{F}$ includes also output behaviour of components (output labels of automata), then $\otimes^{\mathcal{F}}$ models one-to-one output non-blocking synchronization of components with respect to $\mathcal{F}$.

Example 2.19. *Let us again consider the system with architectural description in Figure 1 and automata specification in Figure 2. We will construct the composite automaton $\otimes^{\mathcal{F}}\mathcal{S}$ where $\mathcal{S} = \{\mathcal{C}_i\}_{i\in\{1,2,3\}}$. We want to capture the situation that only the components that are connected by a binding may communicate (by handshake synchronization) and the application has no free inputs or outputs (because no services of components can be delegated outside the system according to the architecture). For this purpose, we will use the operator $\otimes^{\mathcal{F}}$ and we construct the set of feasible labels (representing feasible bindings among components and outside of the application) as $\mathcal{F} = \{(1, \mathsf{check.req}, 3), (3, \mathsf{check.resp}, 1), (2, \mathsf{check.req}, 3), (3, \mathsf{check.resp}, 2), (3, \mathsf{find.req}, 4), (4, \mathsf{find.resp}, 3)\}$. The composite automaton should then consist only of the transitions from the complete transition space that have labels from the set $\mathcal{F}$.*

The selected labels indicate that both Document1 *and* Document2 *components may participate only in internal communication with the* SpellChecker *component. Their external actions modelled by input and output labels are not allowed in the composition as the architecture does* *not enable their delegation out of the application. The same applies to* SpellChecker*. Now we can model the composite automaton as* $\otimes^{\mathcal{F}}\mathcal{S}$*. The diagram in Figure 5 depicts the automaton considering reachable transitions (solid lines) in comparison to the complete transition space (dotted lines). Every state* $(q_1, q_2, q_3) \in Q_1 \times Q_2 \times Q_3$ *is again represented as a sequence* $q_1q_2q_3$ *for short.*



Hierarchy of component names: $((1),(2),(3,4))$

Figure 5: CI automaton $\otimes^{\mathcal{F}}\mathcal{S}$ where $\mathcal{S} = \{\mathcal{C}_i\}_{i\in\{1,2,3\}}$

It remains to be explained why we define the composition as a unary operation where the argument is a set of CI automata, when it is more usual in other languages that the composition is a binary operation. What motivated us is the fact that each composition of CI automata represents a step up in the hierarchy of components in the system. Thus we want to compose all the components representing a particular level in the hierarchy to remember the information about the hierarchical structure of the system. Moreover, the composition operator allows us to compose the set with only one automaton which represents an encapsulation of a component into another component that publishes only selected services of the former one. It is given by way of removal of the transitions that represent the services that are not delegated outside the component.

3 Application

We have already shown that the Component-interaction automata language allows us to model hierarchical component-based systems preserving information about interaction among primitive components of the system. We have introduced a flexible concept of composition which offers a choice of the transition set according to the behaviour that is really possible in the system. This helps the language to model component-based systems quite precisely even if the language of Component-interaction automata is very simple. In this section, we outline some specifics of component-based systems that can also be modelled using this language to provide a better idea of its use.

3.1 Architectural assembly

One of the basic specifics of component-based systems is the architectural assembly of components that constitute the system. In the previous section, we have introduced a composition operator that can be parametrized by the information about the assembly (bindings among components) which allows it to respect the architecture of the system. As a result, the composite automaton can consist only of those transitions that demonstrate existing bindings among components.

This is possible thanks to the information about communicating components which is associated with each use of an action on a transition (in a label in particular). This information facilitates more than just the modeling of static bindings in the system. It for instance allows us to capture that, when a component C receives an action a from a component C', it rebinds from the component C' to a component C'' and from that state C will communicate with C'' only. We can encode such information into the set T that parameterises the composition $\otimes_T$. In particular, such T would not contain those transitions that signifies communication of C and C' on all paths where they are preceded by a transition with the label (C', a, C). Note that this is possible just because we regard an action shared among components as a point of *possible* communication, not as a point of *requisite* interaction as in other languages (Darwin/Tracta [19], Interface automata [12], SOFA Behavior protocols [20], Wright [2]).

3.2 Communication between components

The component-based systems usually have an underlying framework that supplies the components with services such as communication mechanisms. Such frameworks are called component models. We now focus on the communication mechanisms that are usual in such component models. The basic one is a communication through method calls when one component (a client) sends a request for a method to another component (a supplier) that computes the result and returns the response to the former component. In Component-interaction automata, we can model this situation by the use of two distinct actions, one symbolizing the request and the other the response of the method call.

There are two distinct types of method call communication: synchronous and asynchronous. In synchronous communication, the client, after sending the request, is not allowed to perform any action before receiving the response. This information can be captured in the primitive CI automaton for the client and is propagated through the composition naturally. In the asynchronous composition the client after sending the request can continue in the computation before the response is received. This case can be also captured in the primitive CI automaton directly and is propagated through the composition.

Besides the common method call where there is exactly one request and one response for each method call, there are several other types that can occur. They usually involve one request that is followed by zero (in the case of one-way method call) or more than one (a method call with confirmations) responses. These cases can also be modelled within the primitive components using either different or the same action names for particular responses. The special case of one-way calls is message (event) passing that can be also modelled naturally using single actions.

3.3 Types of synchronization

Communication of components in component-based systems is commonly based on a synchronization of an input of one component with an output of another one. There are two types of this synchronization that are usually distinguished.

(i) Blocking – Whenever a component is ready to perform an input/output transition (which cannot be delegated outside), it has to wait (is blocked) until a counterpart is ready (or perform another transition possible in the state).

(ii) Output non-blocking – Whenever a component is ready to perform an output transition, it is free to do so. Whenever a component is ready to perform an input transition, it has to wait until a counterpart is ready.

The most natural synchronization in the sense of Component-interaction automata is the first one (i) (used also in Tracta [19] for instance). According to this, we can let the composition operator remove all external transitions that cannot be delegated outside of the component assuming that they would be blocked as no counterpart was ready. However, in component-based (and also object-oriented) software engineering the more usual notion is the second one (ii) (used by Interface automata [12] or SOFA Behavior protocols [20] for instance). It reflects that a component does not know whether a counterpart is ready and outputs an action whenever it needs to.

In Component-interaction automata, we can capture the synchronization type (ii) by choosing T for a set of CI automata $\mathcal{S}$ as $T \supseteq \{(q,(n_1,a,-),q') \in \Delta_{\mathcal{S}} \mid \nexists n_2, q'' : (q,(n_1,a,n_2),q'') \in \Delta_{\mathcal{S}}\}$. This allows the composite automaton to include the transitions representing output transitions in the states where they cannot be connected with appropriate input actions to form internal actions. Note that we do not have to be afraid that the non-blocked outputs (that cannot be delegated outside the system) will synchronize with another action on a higher level of composition because in architectural assembly there will be no binding that would allow it. Another important point is that there is no explicit indication that the non-blocked outputs are errors. The reason is, that the underlying component model does not need to consider the outputs as errors. In may simply ignore them. Therefore the outputs should not be explicitly marked as errors (as it is in Interface automata and SOFA Behavior protocols) but the information about them should be preserved to be detectable in the verification phase.

Some other types of synchronization can be modelled using connectors. A connector is a unit similar to a component that can simulate a specific kind of communication. For example in the case of broadcast communication, it can be modelled as a CI automaton that whenever it receives an action, it broadcasts the action to other components that are connected to it.

3.4 Other specifics

There are several other specific issues that can be considered for component-based systems. We briefly mention some of them.

The first is modeling of systems where any service of a component can be executed for at most one client at a time. Other attempts to request the service are blocked before the response of the active service is delivered. We can capture this fact in every primitive

CI automaton and guard this condition during the composition (remove all transitions that violate it).

The next specific issue is the form of bindings among components. They can bind the services (methods) or the whole interfaces (set of services/methods). It is usually given whether a service/interface can be bound to only one counterpart, or to several counterparts (where the actual counterpart is selected at run-time). All of these situations can be modelled in Component-interaction automata, as they abstract from the structure of interfaces and are interested only in the information about which components may communicate with each other on a specific action, where an action may be shared by several (not just two) components.

4 Related work

In this section, we discuss several languages that were designed for a similar purpose as Component-interaction automata. However, as the purpose is not exactly the same, we have found some difficulties in their use for specifying component interactions in a flexible way and preserving information about the interaction among components. We have divided the languages into two classes: *automata-based languages*, and languages defined within *architecture description languages*.

4.1 Automata-based languages

The languages discussed in this section have several features in common. They are automata-based and produce models in a form of (finite) transition systems labelled with three types of actions: *input*, *output* and *internal*. One of the main advantages of such languages for specification of component interactions is that automata-based models allow straightforward application of a wide range of formal methods and verification algorithms (model checking [10] in particular). We focus on *I/O automata*, *Team automata*, and *Interface automata*.

Input/Output automata The *Input/Output automata* (*I/O automata* for short) were defined by Lynch and Tuttle in [21, 17] as a labelled transition system language based on nondeterministic automata. The I/O automata language is suitable for modelling distributed and concurrent systems with differentiation of input, output and internal actions. I/O automata can be composed together to form a higher-level I/O automaton and thus form a hierarchy of components of the system. Unfortunately there are several issues that make the task of modelling component interactions using I/O automata difficult.

Firstly, the I/O automata are *input enabled* in all states which means that an I/O automaton can never block an input. Hence in I/O automata, we are unable to directly reason about properties capturing that a component C_1 is ready to send an output action a to a component C_2 which is not ready to receive it (e.g. needs to finish some computation first). Secondly, the sets of input, output and internal actions of I/O automaton have to be pairwise disjoint. This fact may be troublesome when modelling some practical systems that use *method call delegation* for instance, when a component needs to output the same action that it has received as an input. It was important for us to design Component-interaction automata in a way that the sets of input, output and internal actions do not have to be pairwise disjoint.

Regarding the composition, a set of I/O automata may be composed only if the sets of output actions of the automata are pairwise disjoint. Therefore it is not possible to compose a set of I/O automata where two or more automata have the same output action. Moreover, when composing a set of I/O automata, each input action that may synchronize in a composition is removed from the resulting automaton to preserve the condition of disjoint input and output action sets. That input actions then cannot be delegated out of the composed component to be linked in a higher level of composition.

Team automata The *Team automata* language [7] was first introduced in [13] by Ellis. This complex language is designed primarily for modelling groupware systems with communicating teams (using several types of multi-way synchronization) but can be also used for modelling component-based systems (for which one-to-one synchronization is sufficient). Team automata, unlike other automata-based languages, offer freedom of choosing the transition set of the automaton obtained when composing a set of automata, and thus are not limited to one fixed form of synchronization. This feature inspired us to define the flexible composition in Component-interaction automata.

In a team automaton, it is again required that the sets of input, output and internal actions are pairwise disjoint. It implies that the composite automaton cannot propagate some actions of the sub-automata as it would violate this condition. In particular, the composition hides every input action which is an output action of some other automaton in the composition. Therefore the input action cannot be used on a higher level of compositional hierarchy later on. Another important feature is that during the composition, valuable information about component interactions can get lost. For example information about which component automata synchronized on an action.

Interface automata The *Interface automata* model [11, 12] was introduced by de Alfaro and Henzinger. The model is designed for documentation and validation of systems made of components communicating through their interfaces. Interface automata, as distinct from I/O automata, are not input enabled in all states and may compose only two automata at a time.

A significant particularity of Interface automata is the explicit indication of erroneous behaviour (error states) in the composition of two components. An *error state* is the state where one automaton generates an output that is an illegal input for the other automaton. The composition of two interface automata is then defined in a way that the transition set of the product automaton is restricted to disable transitions to error states. It follows the *optimistic* assumption that two automata can be composed if there exists an environment that can make them work together properly (disable transitions to error states). Then the composition of the automata consists only of the transitions available in such an environment.

4.2 Languages defined within ADLs

Architecture description languages (ADLs) may specify both static (system architecture, bindings among components) and dynamical (component behaviour, interactions) aspects of hierarchical component-based systems. These languages are very close to practice and often address many practical issues which can arise in real systems. In this section we focus only on their sub-languages desired for specification of component interactions.

Tracta / Darwin *Tracta approach* [19, 14] proposed by Giannakopoulou defines a language and methods for analysing the behaviour of distributed systems using structural description of an architecture description language *Darwin* [18].

Behaviour of every component is described as a finite state *Labelled Transition System (LTS)*. Considering the composition, one of the shortcomings in the context of modelling component interactions is that Tracta supports only one type of synchronization (determined by strict broadcast semantics of the LTS parallel composition operator) without distinction of input and output actions. This is quite distant from the component-based systems that are usually based on a client–supplier communication principle.

Behavior protocols / SOFA *Behavior protocols* [20, 1] were proposed by Plasil and Visnovsky as a formalism for description of component interaction behaviour within the framework of the *SOFA Component Model* [4] which is a part of the *SOFA project (SOFtware Appliances)* that aims to create a distributed development and run time environment for component-based software systems.

A *behavior protocol* is a regular-like expression enabling behavioural description of entities like interfaces, components or component compositions. During the composition, SOFA enables one type of synchronization with explicit indication of the errors that occurred. This helps in static verification of system correctness but reduces generality of the model. It limits the applicability of behavior protocols solely to those systems that have the same notion of erroneous behaviour.

Parametrized networks of communicating automata / Fractal *Parametrized networks of communicating automata* language [5] was proposed in [6] by Barros et al. as a language for behavioural specification of components in distributed systems as parametrized transition systems. The language relies on the *Fractal component model* [9] which is a general component composition framework with a focus on dynamic reconfiguration.

This language presents a very complex approach to specification of component-based systems. Behaviour of each component is modelled as a finite state *parametrized Labelled Transition System (pLTS)* and the composition is defined using the *parametrized synchronization Network (pNet)* that is a form of a generalised parallel operator where each of its arguments is typed by a *Sort* that determines sets of its possible observable actions for synchronization. The pNet defines synchronization using a *transducer* which is a LTS with synchronization vectors in place of labels, each describing one particular synchronization of the process actions. The composition is computed as a product of the pLTSs with the transducer which controls the synchronization.

Wright An architecture description language *Wright* [3, 2] was proposed by Allen and Garland as a language for describing and analysing software architectures. Wright provides a formal basis for specifying both structure and behaviour of architectural description. Specifically, it supports the description of architectures as hierarchical graphs of components and connectors. Each component and connector is augmented with a description that helps in reasoning about behaviour of a single component and interaction among several components.

The behaviour and coordination of components is specified in Wright using the notation based on CSP [16]. The basic unit of a behaviour specification is an *event* which can be either *initiated* or *observed* (analogically to output and input actions, respectively). The composition is performed using CSP parallel operator $\|$. In the parallel composition of several processes, the processes *must* synchronize on shared actions from their alphabets and interleave independent actions (that are part of an alphabet of just a single process). The CSP interpretation removes the initiate/observe markings on events so that they will synchronize.

5 Conclusion and future work

The paper presents a specification language called Component-interaction automata, an underlying formalism for formal analysis and verification of component-based systems in view of component interactions. The language provides a transparent and understandable way of modelling component interactions thanks to the primary purpose oriented to component-based systems and their specifics. The language is inspired by some features of the languages discussed in the previous section, but differs in many others. It supports freedom of choosing the transition set which facilitates the adjustability according to an architectural description and communication mechanism (inspired by Team automata) and is based on synchronization of one input and one output action with the same name which becomes internal later on (inspired by Interface automata and SOFA Behavior protocols). The language is designed to preserve important interaction attributes to provide a rich base for further verification. As distinct from the discussed languages, it naturally preserves information about the components which participated in a synchronization and about the hierarchical structure of the system.

In future, we aim to address several issues using the language as an underlying formalism. The main area is a temporal verification of coordination errors that may consist of both standard (deadlock, computational progress) and component specific (interaction between specific components) coordination errors. The next interesting set of problems regards the reconfiguration correctness which addresses the correctness of a system after reconfiguration (modification or replacement of a component). Last, not only verification is worth studying in this context. Proper specification of component interactions may open up a gateway to component-based systems optimizations, construction strategies, or error prevention, which are the topics of real practical needs. We hope that the Component-interaction automata will contribute to answering them.

References

[1] J. Adamek. Enhancing behavior protocols. Master's thesis, Charles University, Prague, Faculty of Mathematics and Physics, Czech Republic, September 2001.

[2] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, School of Computer Science, USA, May 1997.

[3] R. J. Allen and D. Garlan. The wright architectural specification language. Technical Report CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, USA, September 1996.

[4] D. Balek, F. Plasil, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS'98)*, pages 43–51, Annapolis, Maryland, USA, May 1998. IEEE Computer Society, USA.

[5] T. Barros. *Formal Specification and Verification of Distributed Component Systems*. PhD thesis, Université de Nice – Sophia Antipolis, France, November 2005.

[6] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed Java objects. In *Proceedings of the 24th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'04)*, pages 43–60, Madrid, Spain, September 2004. LNCS Springer-Verlag.

[7] M. Beek, C. Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in Team automata for groupware systems. *Computer Supported Cooperative Work — The Journal of Collaborative Computing*, 12(1):21–69, February 2003.

[8] L. Brim, I. Černá, P. Vařeková, and B. Zimmerova. Component-Interaction automata as a verification-oriented component-based system specification. In *Proceedings of the ESEC/FSE Workshop on Specification and Verification of Component-Based

Systems (SAVCBS'05), pages 31–38, Lisbon, Portugal, September 2005. Iowa State University, USA. Published also in ACM SIGSOFT Software Engineering Notes, Volume 31, Issue 2 (March 2006).

[9] E. Bruneton, T. Coupaye, and J.-B. Stefani. *The Fractal Component Model, version 2.0-3*, February 2004.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, USA, January 2000.

[11] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 9th Annual Symposium on Foundations of Software Engineering (FSE'01)*, pages 109–120, Vienna, Austria, September 2001. ACM Press, USA.

[12] L. de Alfaro and T. A. Henzinger. Interface-based design. In *Proceedings of the 2004 Marktoberdorf Summer School*, Germany, 2005. Kluwer, The Netherlands.

[13] C. Ellis. Team automata for groupware systems. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge (GROUP'97)*, pages 415–424, Phoenix, AZ, USA, November 1997. ACM Press, USA.

[14] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, UK, January 1999.

[15] G. T. Heineman and W. T. Councill. *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, June 2001.

[16] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[17] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 137–151, Vancouver, Canada, August 1987. ACM Press, USA.

[18] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, pages 137–153, Sitges, Spain, September 1995. Springer-Verlag, UK.

[19] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA'99)*, pages 35–50, San Antonio, TX, USA, February 1999. Kluwer, The Netherlands.

[20] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, November 2002.

[21] M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, USA, April 1987.