Component-Interaction Automata as a Verification-Oriented Component-Based System Specification

Luboš Brim[*] , Ivana Černá[†] , Pavlína Vařeková[‡] , Barbora Zimmerova[‡]
Faculty of Informatics
Masaryk University, Brno
602 00 Brno, Czech Republic
{brim,cerna,xvareko1,zimmerova}@fi.muni.cz

ABSTRACT

In the paper, we present a new approach to component interaction specification and verification process which combines the advantages of both architecture description languages (ADLs) at the beginning of the process, and a general formal verification-oriented model connected to verification tools at the end. After examining current general formal models with respect to their suitability for description of component-based systems, we propose a new verification-oriented model, *Component-Interaction automata*, and discuss its features. The model is designed to preserve all the interaction properties to provide a rich base for further verification, and allows the system behaviour to be configurable according to the architecture description (bindings among components) and other specifics (type of communication used in the synchronization of components).

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.11 [**Software Engineering**]: Software Architecture

General Terms

Component-based specification languages

Keywords

ADLs, I/O automata, Interface automata, Team automata, Component-Interaction automata, component interaction, verification.

[*]The author has been partially supported by the grant GACR 201/03/0509

[†]The author has been partially supported by the grant No. 1ET408050503

[‡]The authors have been supported by the grant No. 1ET400300504

1. INTRODUCTION

Verification of interaction properties in component-based software systems is highly dependent on the chosen specification language. There are many possibilities how to specify component behaviour and interaction in component-based software systems. The specification languages typically fall into two classes with diverse pros and cons.

The first set of specification languages is called Architecture Description Languages (ADLs). Architecture description languages, like Wright [3], Darwin/Tracta [15, 16], Rapide [11], and SOFA [17, 2], are very suitable for specification of hierarchical component architecture with defined interconnection among components and behaviour constraints put on component communication and interaction. Moreover they are very comprehensible for software engineers and often provide a tool support. The essential drawback of the ADLs is that their specification power is limited by the underlying model which is often not general enough to preserve all the interaction properties which might arise through the component composition. Additionally, the verification within an ADL framework usually supports a verification of only a small fixed set of properties often unique for the language.

The second set consists of general formal models usually based on the automata theory (I/O automata [14, 12], Interface automata [8], Team automata [5]). These automata-based models (as opposite to ADLs) are highly formal and general, and usually supported by automated verification tools (model-checkers in particular). However, these models are designed for modelling of component interaction only and therefore are unable to describe the interconnection structure of hierarchical component architecture which also influences the behaviour. That is one of the reasons why these models are often considered to be unusable in software engineering practice.

The point we want to address in our research is to combine these two approaches to gain the benefits of both of them. In particular, we would like to develop a general automata-based formalism which allows for the specification of component interactions according to the interconnection structure described in particular ADL. The transition set of the model should be therefore configurable according to the architecture description (bindings among components) and other specifics (type of communication used in the synchronization of components). In addition, the formalism should allow for an easy application of available automated verifica-

tion techniques. The idea is to support the specification and verification process automatically or semi-automatically so that it is accessible also for users with no special theoretical knowledge of the underlying model. The specification and verification process will constitute of the following phases.

1. The user selects an appropriate ADL and specifies the system architecture and component behaviour using an ADL tool.
2. Component behaviour description is transformed into the general formal model automatically using the model framework.
3. The hierarchical component composition is build within the framework with respect to the architecture description and synchronization type.
4. The result is verified directly within the model framework or transformed to a format accepted by verification tools.

In this paper we want to address the first step towards this goal. We propose an appropriate general verification-oriented specification formalism which covers all important features of component interaction in component-based systems, including hierarchical interconnection interaction, and enables its adjustment to the ADL specification. At the same time, the model is defined in such a way that a direct application of model checking techniques is possible.

The paper is organised as follows. After discussing related work in Section 2, Section 3 focuses on the automata-based models appropriate for component interaction specification, and gives the reasons for introduction of a new model in Section 4. Section 5 concludes by discussion the most important features of the proposed model, and presents the plans for future work.

2. RELATED WORK

Our approach to support the specification and verification process by combining ADL specification and a general formal model verification has not been considered yet. The reason is that the architecture description languages usually support some kind of verification of the interaction properties and therefore there is no visible need for use of a new general verification-oriented model.

Some of the architecture description languages addressing the issue of formal verification of behaviour properties of the system composed from components are *Wright* [3], *Darwin/Tracta* [15, 16], *Rapide* [11] and *SOFA* [17, 2]. *Wright* uses consistency and completeness checks defined in terms of its underlying model in CSP. Verification of component behaviour in *Darwin* is supported by the *Tracta* approach which defines component interactions using labelled transition systems (LTS) and employs model checking [10] to verify some of its properties (reachability analysis, safety and liveness properties). *Rapide* generates a system execution in a form of partially ordered set of events and allows its check against properties. *SOFA* uses *behavior protocols* to specify behaviour of a component frame (black-box specification view) and architecture (grey-box implementation view) and employs a compliance checking to verify their conformance.

As we emphasised in the introduction, these languages typically support verification of a limited set of interaction properties. Even if some ADLs attempt to support an exhaustive verification [10], they are limited by the underlying behaviour model which is usually designed for one particular type of communication and notion of erroneous behaviour and thus does not cover some important interaction properties. For example the parallel composition operator $\|$ used in Tracta does not assure, that we will be later able to detect all states where one of the (sub)components is ready to synchronize on a shared action but the others are not because in Tracta's notion of communication it is not an interesting behaviour to capture.

Another approach is a support of the specification and verification process using the formal general language only (I/O automata [14], Interface automata [8], Team automata [5]). The essential drawback of this approach is that the models specify just the interaction behaviour without describing the underlying architectural framework. That is restrictive especially when the interconnection structure of a system differs from the complete interconnection space determined by the actions shared among components.

3. AUTOMATA-BASED LANGUAGES

As already mentioned in the introduction, automata-based models are typically supported by automated verification tools. In this section we primarily concentrate on their applicability for capturing of behaviours of component-based systems, especially the interaction among components of the system. The best known models used in this context are I/O automata, Interface automata, and Team automata. For each of the models we give a brief definition and review its main features interesting in a light of modelling component-based systems.

Notation: Let $\mathcal{I} \subseteq \mathbb{N}$ be a finite set with cardinality k, and let for each $i \in \mathcal{I}$, S_i be a set. Then $\Pi_{i\in\mathcal{I}} S_i$ denotes the set $\{(x_{i_1}, x_{i_2}, \ldots, x_{i_k}) \mid (\forall j \in \{1, \ldots, k\} : x_{i_j} \in S_{i_j}) \wedge \{i_1, i_2, \ldots, i_k\} = \mathcal{I} \wedge (\forall j_1, j_2 \in \{1, \ldots, k\} : j_1 < j_2 \Rightarrow i_{j_1} < i_{j_2})\}$. If $\mathcal{I} = \emptyset$ then $\Pi_{i\in\mathcal{I}} S_i = \emptyset$. For $j \in \mathcal{I}$, $proj_j$ denotes the function $proj_j : \Pi_{i\in\mathcal{I}} S_i \to S_j$ for which $proj_j((q_i)_{i\in\mathcal{I}}) = q_j$.

3.1 I/O automata

The *Input/Output automata* model (*I/O automata* for short) was defined by Nancy A. Lynch and Mark R. Tuttle in [18, 13] as a labelled transition system model based on nondeterministic automata. The I/O automata model is suitable for modelling distributed and concurrent systems with different input, output and internal actions. I/O automata can be composed to form a higher-level I/O automaton and thus form a hierarchy of components of the system.

Definition: A *(safe) I/O automaton* is a tuple $\mathcal{A} = (Q, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, I)$, where

- Q is a set of states.
- $\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}$ are pairwise disjoint sets of *input, output* and *internal* actions, respectively. Let $\Sigma = \Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}$ be called a *set of actions.*
- $\delta \subseteq Q \times \Sigma \times Q$ is a set of *labelled transitions* such that for each $a \in \Sigma_{inp}$ and $q \in Q$ there is a transition $(q, a, q') \in \delta$ (*input enableness*).
- $I \subseteq Q$ is a nonempty set of *initial states.*

Important feature to mention is that I/O automata are input enabled in all states, they can never block the input. It

means that in I/O automata we are unable to directly reason about properties capturing that a component A is ready to send output action a to a component B which is not ready to receive it (e.g. needs to finish some computation first). Other feature to notice is that the sets of input, output and internal actions of I/O automaton have to be pairwise disjoint. It can limit us in modelling some practical systems. For example when we want to model a system in Figure 1, consisting of n component instances of the same type that enable event delegation (the component C_i can delegate the method call which received from the component C_{i-1} to the component C_{i+1}), we cannot do it directly. We have to use appropriate relabelling.

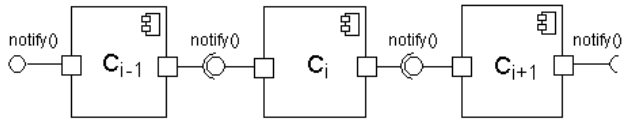

Figure 1: Delegation of a method call, UML 2.0

A set of I/O automata is *strongly compatible* if the sets of output actions of component automata are pairwise disjoint and the set of internal actions of every component automaton is disjoint with the action sets of all other component automata. Therefore a set of automata where two or more automata have the same output action is not strongly compatible and cannot be composed according to the next definition. At the same time this property is quite often in practical component-based systems, for example when two components are using the same service of another component. This problem could be solved by relabelling of the transitions. Note that simple including the identity of the component in the action name would not suffice because I/O automata are able to synchronize only on actions with the same name.

Definition: Let $\mathcal{S} = \{(Q_i, \Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}, \delta_i, I_i)\}_{i\in\mathcal{I}}$, where $\mathcal{I} \subseteq \mathbb{N}$ is finite, be a strongly compatible set of I/O automata. An I/O automaton $(\Pi_{i\in\mathcal{I}}Q_i, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, \Pi_{i\in\mathcal{I}}I_i)$ is a *composition* of $\mathcal{S}$ iff

- $\Sigma_{inp} = \left(\bigcup_{i\in\mathcal{I}} \Sigma_{i,inp}\right) \setminus \left(\bigcup_{i\in\mathcal{I}} \Sigma_{i,out}\right)$,
- $\Sigma_{out} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,out}$,
- $\Sigma_{int} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,int}$ and
- for each $q, q' \in \Pi_{i\in\mathcal{I}}Q_i$ and $a \in \Sigma$, $(q, a, q') \in \delta$ iff for all $i \in \mathcal{I}$ if $a \in \Sigma_i$ then $(proj_i(q), a, proj_i(q')) \in \delta_i$ and if $a \notin \Sigma_i$ then $proj_i(q) = proj_i(q')$.

In the composition of strongly compatible I/O automata each input action a, for which an appropriate output action a exists, is removed to preserve the condition of disjoint input and output action sets. The input actions then cannot be delegated out of the composed component to be linked in a higher level of composition. Another property of I/O automata to mention is that they do not allow to specify which outputs and inputs should be bound and which should stay unbound (according to the architecture description or type of synchronization).

3.2 Interface automata

The *Interface automata* model [8] was introduced in [7] by Luca de Alfaro and Thomas A. Henzinger. The model is designed for documentation and validation of systems made of components communicating throw their interfaces. Interface automata, as distinct from I/O automata, are not input enabled in all states and allow composition of two automata only. Moreover, composition is based on synchronization of one output and one input action (with the same name) which becomes hidden after the composition. That is natural for practical component-based systems.

An *interface automaton* is defined in the same way as I/O automaton with the only difference that interface automaton need not to be input enabled. The sets of input, output, and internal (called hidden in this case) actions again have to be pairwise disjoint.

Definition: Let $\mathcal{A}_i = (Q_i,\ \Sigma_{i,inp},\ \Sigma_{i,out},\ \Sigma_{i,int},\ \delta_i,\ I_i)$, $i = 1, 2$, be interface automata. Then the set $\Sigma_1 \cap \Sigma_2$ is called $shared(\mathcal{A}_1, \mathcal{A}_2)$. Automata $\mathcal{A}_1, \mathcal{A}_2$ are *composable* iff

$$shared(\mathcal{A}_1, \mathcal{A}_2) = (\Sigma_{1,inp} \cap \Sigma_{2,out}) \cup (\Sigma_{2,inp} \cap \Sigma_{1,out}).$$

It means that, except of actions which are input of the first and output of the second automaton or vice versa, the sets of actions of two composable automata have to be disjoint.

Definition: Let $\mathcal{A}_i = (Q_i,\ \Sigma_{i,inp},\ \Sigma_{i,out},\ \Sigma_{i,int},\ \delta_i,\ I_i)$, $i = 1, 2$, be composable interface automata. Then $(Q_1 \times Q_2, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, I_1 \times I_2)$ is a *product of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ iff

- $\Sigma_{inp} = (\Sigma_{1,inp} \cup \Sigma_{2,inp}) \setminus shared(\mathcal{A}_1, \mathcal{A}_2)$,
- $\Sigma_{out} = (\Sigma_{1,out} \cup \Sigma_{2,out}) \setminus shared(\mathcal{A}_1, \mathcal{A}_2)$,
- $\Sigma_{int} = (\Sigma_{1,int} \cup \Sigma_{2,int}) \cup shared(\mathcal{A}_1, \mathcal{A}_2)$ and
- $((q_1, q_2), a, (q_1', q_2')) \in \delta$ iff
  $a \notin shared(\mathcal{A}_1, \mathcal{A}_2)\ \wedge\ (q_1, a, q_1') \in \delta_1\ \wedge\ q_2 = q_2'$
  $a \notin shared(\mathcal{A}_1, \mathcal{A}_2)\ \wedge\ q_1 = q_1'\ \wedge\ (q_2, a, q_2') \in \delta_2$
  $a \in shared(\mathcal{A}_1, \mathcal{A}_2) \wedge (q_1, a, q_1') \in \delta_1 \wedge (q_2, a, q_2') \in \delta_2$.

The definition implies that the linking of input and output action, from the set of shared actions, is compulsory as in the I/O model. Additionally, the model does not permit multiple binding on the interfaces directly without renaming (e.g. two components using the same service provided by other component). Each input (output) action after linking to an appropriate output (input) action becomes internal action and therefore is not allowable for other linking.

The transition set of the product of two interface automata contains all syntactically correct transitions. *Composition* of two interface automata is a restriction of the product automaton. The restriction is defined with the help of *error* and *compatible states*, and *compatibility* of two automata (for formal definitions see [8]). The product state (q_1, q_2) of two composable interface automata is an error state if it corresponds to a state where one of the automata is able to send an output action a and the other one is not able to receive the action a (a is a shared action). A state q is compatible if no error state is reachable from q performing only output and internal actions. Two interface automata with initial states q_1, q_2 are compatible, if they are composable and the initial state (q_1, q_2) of their product is a compatible state.

Definition: Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be compatible interface automata and $(Q, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, I)$ be their product. Interface automaton $(Q, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta', I)$ is a *composition* of $\mathcal{A}_1$ and $\mathcal{A}_2$ iff $\delta' = \delta \backslash \{(q, a, q') \mid q$ is compatible, $a \in \Sigma_{inp}$, and q' is not compatible$\}$.

The composition of interface automata is defined in two steps. In the first step the product automaton is built and the set of error and compatible states are formed. In the second step the transition function of the product automaton is restricted to disable transitions to incompatible states. It follows the *optimistic* assumption that two automata can be composed if there exists some environment that can make them work together properly. Then the composition of the automata consists of the transitions available in such environment.

The shortcoming of this approach is the explicit indication of erroneous behaviour (error states) that limits this approach to modelling solely the component-based systems with equivalent notion of what is and is not considered as an error (respecting one type of synchronization).

3.3 Team automata

The *Team automata* model [5] was first introduced in [9] by Clarence A. Ellis. This complex model is primary designed for modelling groupware systems with communicating teams but can be also used for modelling component-based systems. It is inspired by I/O automata. Team automata, as a main distinct from the previous models, allow freedom of choosing the transition set of the automaton obtained when composing a set of automata, and thus are not limited to one synchronization only.

A *team automaton* is defined in the same way as I/O automaton with the only difference that team automaton need not to be input enabled. A set of component automata is *composable* if the set of internal actions of every component automaton is disjoint with the action sets of all other component automata. The composition of team automata is defined over a *complete transition space.*

Definition: Let $\mathcal{S} = \{(Q_i, \Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}, \delta_i, I_i)\}_{i \in \mathcal{I}}$, where $\mathcal{I} \subseteq \mathbb{N}$ is finite, be a composable system of component automata and $a \in \bigcup_{i \in \mathcal{I}} \Sigma_i$. Then a *complete transition space* of a in $\mathcal{S}$ is denoted $\Delta_a(\mathcal{S})$ and defined as

$\Delta_a(\mathcal{S}) = \{(q, a, q') \mid q, q' \in \Pi_{i \in \mathcal{I}} Q_i \wedge$
$\exists j \in \mathcal{I} : (proj_j(q), a, proj_j(q')) \in \delta_j \wedge$
$\forall i \in \mathcal{I} : ((proj_i(q), a, proj_i(q')) \in \delta_i \vee proj_i(q) = proj_i(q'))\}$.

$\mathcal{T} = (\Pi_{i \in \mathcal{I}} Q_i, \Sigma_{inp}, \Sigma_{out}, \Sigma_{int}, \delta, \Pi_{i \in \mathcal{I}} I_i)$ is a *team automaton over* $\mathcal{S}$ iff

- $\Sigma_{inp} = \left(\bigcup_{i \in \mathcal{I}} \Sigma_{i,inp}\right) \setminus \left(\bigcup_{i \in \mathcal{I}} \Sigma_{i,out}\right)$,
- $\Sigma_{out} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$,
- $\Sigma_{int} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,int}$ and
- $\delta \subseteq \Pi_{i \in \mathcal{I}} Q_i \times \Sigma \times \Pi_{i \in \mathcal{I}} Q_i$, such that for all $a \in (\Sigma_{inp} \cup \Sigma_{out})$, δ restricted to a is a subset of $\Delta_a(\mathcal{S})$, and for all $a \in \Sigma_{int}$, δ restricted to a is equal to $\Delta_a(\mathcal{S})$.

The important fact to mention is that 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, when composing automata, we can loose some information about the behaviour of the system. For example, let us consider the component automata $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ from Figure 2 where $\mathcal{A}_1$ has one transition over *output* action a and both $\mathcal{A}_2$ and $\mathcal{A}_3$ have one transition over *input* action a. After composing these three automata to the automaton $\mathcal{A}$ over $\{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3\}$ (with one transition over *output* action a), we cannot differentiate between synchronization of the input of $\mathcal{A}_2$ with the output of $\mathcal{A}_1$ and synchronization of $\mathcal{A}_3$ with $\mathcal{A}_1$. This can be quite restrictive for verification of properties capturing which components participated in the computation. Moreover, if we would need to express that an automaton $\mathcal{A}_1$ in a state q_0 can synchronize with $\mathcal{A}_2$ only, we cannot include this information in the composition withou renaming.

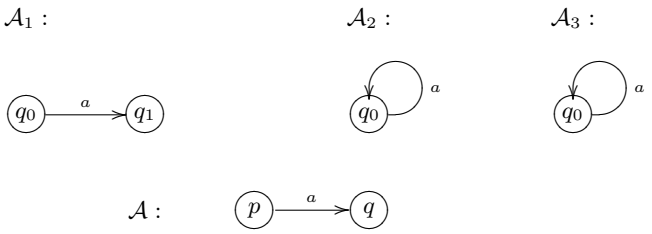

Figure 2: Composition of automata (p states for (q_0, q_0, q_0), q states for (q_1, q_0, q_0))

3.4 Summary

The characteristics of the current models described in this section make their applicability for the full description of interactions in component-based systems difficult. It is natural because studied models were often designed for a slightly different purpose (I/O automata, Team automata) and usually are limited to one strict type of synchronization (I/O automata, Interface automata) which we do not want to limit to. In some cases, relabelling and transformation of the component automata before each composition would be sufficient to express desired properties. But the price we would have to pay for it is in considerable state expanding, untransparency and uncomfortable use of the model. Moreover there are features (like strict synchronization at Interface automata or input enableness at I/O automata) which would be nontrivial to overcome.

4. COMPONENT-INTERACTION AUTOMATA

The issues mentioned in the previous section has motivated us to evolve a new verification-oriented automata-based formal model, *Component-Interaction automata*, designed for specification of interacting components in component-based systems with respect to several aspects of the systems (ADL interconnection structure, way of communication among components). The Component-Interaction automata make it possible to model all interesting aspects of component interaction in hierarchical component-based software systems without loosing any behaviours, and verify interaction behaviour of the systems as well. The model respects current ADLs to enable direct transformation of the ADL description to Component-Interaction automata, and current verification tools as Component-Interaction automata can be translated into their specification languages.

The Component-Interaction automata model is inspired by the Team automata model, mainly in freedom of choosing the transition set of the composed automaton what enables it to be architecture and synchronization configurable. However, Component-Interaction automata differ from Team automata in many aspects to be more comfortable in use for component-based systems, and to preserve important information about the interaction among synchronized components and hierarchical structure of the composed system.

Component-interaction automaton over a set of components S is a nondeterministic automaton where every transition is labelled as an input, output, or internal. Sets of input, output and internal actions need not to be pairwise disjoint. Input (output) action is associated with the name of a component which receives (sends) the action. Internal action is associated with a tuple of components which synchronize on the action. In composition of component-interaction automata, only two components can synchronize and the information about their communication is preserved. If a component has an input (output) action a, the composition also can have a as its input (output) action even after the linking of the action.

Notation: Let $\mathcal{I} \subseteq \mathbb{N}$ be a finite nonempty set with cardinality k, and let $\{S_i\}_{i\in\mathcal{I}}$ be a set. Then $(S_i)_{i\in\mathcal{I}}$ denotes the tuple $(S_{i_1}, S_{i_2}, \ldots, S_{i_k})$, where $\{i_1, i_2, \ldots, i_k\} = \mathcal{I}$ and for all $j_1, j_2 \in \{1, 2, \ldots, k\}$ if $j_1 < j_2$ then $i_{j_1} < i_{j_2}$.

4.1 Definition

Definition: A *component-interaction automaton* is a tuple $\mathcal{C} = (Q, Act, \delta, I, S)$ where

- Q is a finite set of *states*,
- Act is a finite set of *actions*, $\Sigma = ((X \cup \{-\}) \times Act \times (X \cup \{-\})) \setminus (\{-\} \times Act \times \{-\})$ where $X = \{n \mid n \in \mathbb{N},\ n \text{ occurs in } S\}$, is a set of *symbols* called an *alphabet*,
- $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of *labelled transitions*,
- $I \subseteq Q$ is a nonempty set of *initial states* and
- S is a tuple corresponding to a hierarchy of component names (from $\mathbb{N}$) whose composition $\mathcal{C}$ represents.

Symbols $(-, a, B)$, $(A, a, -)$, $(A, a, B) \in \Sigma$ are called *input*, *output* and *internal symbols* of the alphabet Σ, respectively. Accordingly, transitions are called input, output, and internal.

- The input symbol $(-, a, B)$ represents that the component B receives an action a as an input.
- The output symbol $(A, a, -)$ represents that the component A sends an action a as an output.
- The internal symbol (A, a, B) represents that the component A sends an action a as an output, and synchronously the component B receives the action a as an input.

Remark, that component-interaction automaton need not have disjoint sets of input actions (those involved in input transitions), output actions (involved in output transitions), and internal actions (involved in internal transitions).

As it can be seen from the structure of symbols, only two components can synchronize on the same action. It is a natural way of component communication according to a client–server principle. If we would like to address multi-way synchronization, the model could be naturally extended to *Multi Component-Interaction automata*, where the symbols would be represented as tuples $(\mathsf{A}, a, \mathsf{B})$ where A stands for a set of sending components and B for a set of receiving components.

Example 4.1.: Let us consider the system from Figure 3 (modelled in UML 2.0). The component C_1 sends an action a through the interface I_2 and sends an action b through the interface I_1. The component C_2 receives an action a through the interface I_3. C_3 sends a through I_6, C_4 receives a through I_4 and sends b through I_5. Finally, C_5 sends b through I_7.

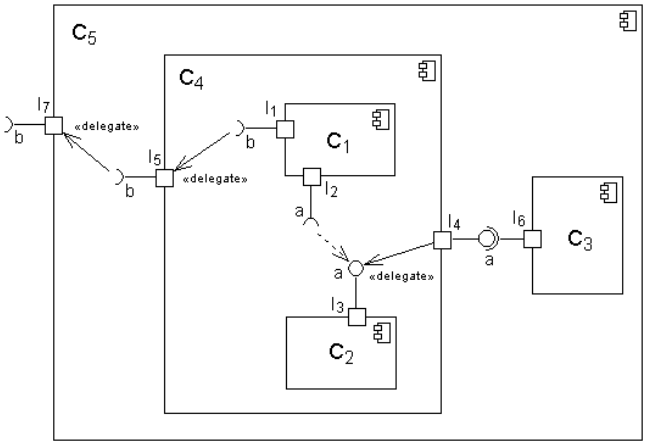

Figure 3: Component model of a simple system

Component-interaction automata $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$, modelling components C_1, C_2, and C_3 from Figure 3, respectively, follows (their graphical representation is in Figure 4).

$\mathcal{A}_1 = (\{q_0, q_1\}, \{a, b\}, \{(q_0, (1, a, -), q_1),$
$(q_1, (1, b, -), q_1)\}, \{q_0\}, (1))$

$\mathcal{A}_2 = (\{q_0\}, \{a\}, \{(q_0, (-, a, 2), q_0)\}, \{q_0\}, (2))$

$\mathcal{A}_3 = (\{q_0\}, \{a\}, \{(q_0, (3, a, -), q_0)\}, \{q_0\}, (3))$

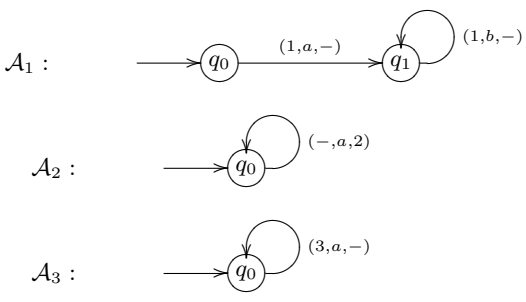

Figure 4: Automata $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ modelling components C_1, C_2, and C_3.

Component-interaction automata can be composed and form a hierarchical structure which is preserved as visible from the next definition.

Definition: Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, S_i)\}_{i\in\mathcal{I}}$, where $\mathcal{I} \subseteq \mathbb{N}$ is finite, be a system of component-interaction automata such that sets of components represented by the automata are pairwise disjoint. Then $\mathcal{C} = (\Pi_{i\in\mathcal{I}} Q_i, \cup_{i\in\mathcal{I}} Act_i, \delta, \Pi_{i\in\mathcal{I}} I_i, (S_i)_{i\in\mathcal{I}})$ is a component-interaction automaton *over* $\mathcal{S}$ iff
$\delta = \Delta_{OldInternal} \cup \delta_{NewInternal} \cup \delta_{Input} \cup \delta_{Output}$ where

$\Delta_{OldInternal} = \{(q, (A, a, B), q') \mid \exists i \in \mathcal{I} :$
$(proj_i(q), (A, a, B), proj_i(q')) \in \delta_i, \ \forall j \in \mathcal{I}, \ j \neq i :$
$(proj_j(q)) = proj_j(q')\}$

$\Delta_{NewInternal} = \{(q, (A, a, B), q') \mid \exists i_1, i_2 \in \mathcal{I}, i_1 \neq i_2 :$
$(proj_{i_1}(q), (A, a, -), proj_{i_1}(q')) \in \delta_{i_1} \ \wedge$
$\wedge \ (proj_{i_2}(q), (-, a, B), proj_{i_2}(q')) \in \delta_{i_2} \ \wedge$
$\wedge \ \forall j \in \mathcal{I} : i_1 \neq j \neq i_2 \ proj_j(q) = proj_j(q')\}$

$\delta_{NewInternal} \subseteq \Delta_{NewInternal}$

$\Delta_{Input} = \{(q, (-, a, B), q') \mid \exists i_1 \in \mathcal{I} :$
$(proj_{i_1}(q), (-, a, B), proj_{i_1}(q')) \in \delta_{i_1} \ \wedge \ \forall j \in \mathcal{I} : i_1 \neq j :$
$(proj_j(q) = proj_j(q'))\}$

$\delta_{Input} \subseteq \Delta_{Input}$

$\Delta_{Output} = \{(q, (A, a, -), q') \mid \exists i_2 \in \mathcal{I} :$
$(proj_{i_2}(q), (A, a, -), proj_{i_2}(q')) \in \delta_{i_2} \ \wedge \ \forall j \in \mathcal{I} : j \neq i_2 :$
$(proj_j(q) = proj_j(q'))\}$

$\delta_{Output} \subseteq \Delta_{Output}$

Transitions in $\Delta_{OldInternal}$ are internal transitions of the component automata. Transitions in $\delta_{NewInternal}$ arise from synchronization of two components. δ_{Input} and δ_{Output} are input and output transitions of the composed automaton provided by components of the composed automaton.

In the definition, we use auxiliary sets Δ_s, $s \in \{NewInternal, Input, Output\}$. Each of these sets represents all possible transitions (complete transition space) over a specific set of symbols determined by the index s. The architecture of the modelled component-based system and other advanced characteristics determine which transitions from the complete transition space are included in the composed automaton. The idea is that the final transition set δ is formed automatically according to the rules specifying the complete transition space, interconnection rules generated from the ADL description and other specified characteristics.

Example 4.2.: Let us illustrate the composition on automata from Example 4.1. Automaton $\mathcal{A}_4$ (modelling the component C_4 from Figure 3) is a component-interaction automaton over $\mathcal{A}_1$ and $\mathcal{A}_2$. Automaton $\mathcal{A}_5$ (modelling the component C_5 from Figure 3) is a component-interaction automaton over $\mathcal{A}_3$ and $\mathcal{A}_4$. The architecture of the composition is determined by the UML 2.0 description of the system in Figure 3.

$\mathcal{A}_4 = (\{s_0, s_1\}, \{a, b\}, \{(s_0, (-, a, 2), s_0), (s_0, (1, a, 2)s_1),$
$(s_1, (1, b, -), s_1), (s_1, (-, a, 2), s_1)\}, \{s_0\}, ((1), (2)))$
$\Delta_{OldInternal} = \emptyset$
$\Delta_{NewInternal} = \{(s_0, (1, a, 2), s_1)\}$
$\delta_{NewInternal} = \{(s_0, (1, a, 2), s_1)\}$
$\Delta_{Input} = \{(s_0, (-, a, 2), s_0), (s_1, (-, a, 2), s_1)\}$
$\delta_{Input} = \{(s_0, (-, a, 2), s_0), (s_1, (-, a, 2), s_1)\}$
$\Delta_{Output} = \{(s_0, (1, a, -), s_1), (s_1, (1, b, -), s_1)\}$
$\delta_{Output} = \{(s_1, (1, b, -), s_1)\}$
Here s_0 and s_1 represent the states (q_0, q_0) and (q_1, q_0), respectively. For the graphical representation of $\mathcal{A}_4$ see Figure 5.

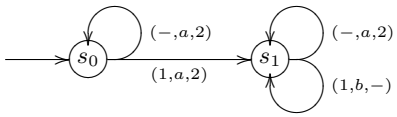

Figure 5: Automaton $\mathcal{A}_4$ modelling component C_4

$\mathcal{A}_5 = (\{p_0, p_1\}, \{a, b\}, \{(p_0, (3, a, 2), p_0), (p_0, (1, a, 2), p_1),$
$(p_1, (3, a, 2), p_1), (p_1, (1, b, -), p_1)\}, \{p_0\}, (((1), (2)), (3)))$
$\Delta_{OldInternal} = \{(p_0, (1, a, 2), p_1)\}$
$\Delta_{NewInternal} = \{(p_0, (3, a, 2), p_0), (p_1, (3, a, 2), p_1)\}$
$\delta_{NewInternal} = \{(p_0, (3, a, 2), p_0), (p_1, (3, a, 2), p_1)\}$
$\Delta_{Input} = \{(p_0, (-, a, 2), p_0), (p_1, (-, a, 2), p_1)\}$
$\delta_{Input} = \emptyset$
$\Delta_{Output} = \{(p_0, (3, a, -), p_0), (p_1, (1, b, -), p_1), (p_1, (3, a, -), p_1)\}$
$\delta_{Output} = \{(p_1, (1, b, -), p_1)\}$
Here p_0 and p_1 represent the states $((q_0, q_0), q_0)$ and $((q_1, q_0), q_0)$, respectively. For the graphical representation of $\mathcal{A}_5$ see Figure 6.

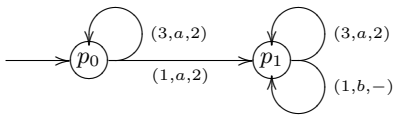

Figure 6: Automaton $\mathcal{A}_5$ modelling component C_5

The operation of composition let us model the hierarchical structure of component-based systems. The base of the composition are *primitive* component-interaction automata. A component-interaction automaton is primitive if it represents one individual component only. Automata $\mathcal{A}_1, \mathcal{A}_2$, and $\mathcal{A}_3$ from Example 4.1 are primitive. In the modelling and verification process we often need to consider only input and output transitions of a component-interaction automaton, which corresponds to the notion of primitiveness. Therefore we define a relation *primitive to* which enables us to transform any component-interaction automaton to a primitive one if we want to make the system less complex for further verification.

Definition: Let $\mathcal{C} = (Q, Act, \delta, I, S)$ be a component-interaction automaton. Then component-interaction automaton $\mathcal{C}' = (Q, Act, \delta', I, (n))$ is *primitive to* the component-interaction automaton $\mathcal{C}$ iff

- $n \in \mathbb{N}$ does not occur in S,
- $(q, (n, a, n), q') \in \delta'$ 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 4.3.: Automaton $\mathcal{B}_4$ (see Figure 7) is primitive to the automaton $\mathcal{A}_4$ from Example 4.2. (s_0 and s_1 represent the states (q_0, q_0) and (q_1, q_0), respectively).

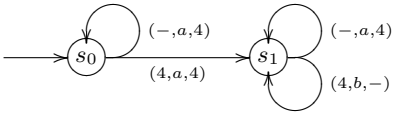

Figure 7: Automaton $\mathcal{B}_4$

4.2 Verification

For a given component-interaction automaton its behaviour can be defined through its *executions* and *traces*.

Definition: An *execution fragment* of a component-interaction automaton $\mathcal{C} = (Q, Act, \delta, I, S)$ is an infinite alternating sequence $q_0, x_0, q_1, x_1, \ldots$ of states and symbols of the alphabet Σ such that $(q_i, x_i, q_{i+1}) \in \delta$ for all $0 \leq i$. An *execution* of $\mathcal{C}$ is an execution fragment $q_0, x_0, q_1, x_1, \ldots$ such that $q_0 \in I$. An execution fragment is *closed* if all its symbols are internal. A *trace* of $\mathcal{C}$ is a sequence $x_0, x_1, \ldots$ of symbols for which there is an execution $q_0, x_0, q_1, x_1, \ldots$.

In real component-based systems one needs to verify various properties of system behaviour. If the system is modelled as a component-interaction automaton the behaviour capturing the interaction among components and architectural levels are the traces. Both linear and branching time temporal logics have proved to be useful for specifying properties of traces. There are several formal methods for checking that a model of the design satisfies a given specification. Among them those based on automata [6] are especially convenient for our model of Component-Interaction automata.

We have experimentally verified several specifications of a component-based systems modelled as a component-interaction automata with the help of DiVinE [1, 4]. DiVinE (Distributed Verification Environment) is a model checking tool that supports distributed verification of systems. The DiVinE native input language si based on finite automata and the component-interaction automata can be translated into the DiVinE language. The tool supports verification of LTL properties.

5. CONCLUSIONS AND FUTURE WORK

The paper presents a new formal verification-oriented component-based specification language named Component-Interaction automata. This model is defined with the aim to support specification and verification of component interactions according to the interconnection architecture and other aspects of modelled system. On the one hand, Component-Interaction automata are close to architecture description languages which can be (semi)automatically transformed into Component-Interaction automata without loosing important behavioural characteristics. On the other hand, the proposed model is close to Büchi automata model and this admits automata-based verification of temporal properties of component interactions.

The Component-Interaction automata model aims to provide a direct and desirable way of modelling component-based systems which is meant to be more transparent and understandable thanks to the primary purpose oriented to component-based systems and their specifics. The model is inspired by some features of previously discussed models and differs in many others. It allows the freedom of choosing the transition set what allows its configurability according to the architecture description (inspired by Team automata) and is based on synchronization on one input and one output action with the same name which becomes internal later on (inspired by Interface automata). The model is designed to preserve all important interaction properties to provide a rich base for further verification. As a distinct from the models discussed in Section 3, it naturally preserves information about the components which participated in the synchronization and about the hierarchical structure, directly without renaming that would make the model less readable and understandable. Even if some component-based systems could be modeled by previously discussed models (I/O automata, Interface automata, Team automata) with appropriate relabelling, it would be for a price of considerable state expanding, untransparency and uncomfortable use of the model.

Nowadays we are developing an automatic transformation from SOFA ADL specification to Component-Interaction automata and from Component-Interaction automata to DiVinE model checking tool native input language. In the future, we intent to study Component-Interaction automata model in a more detailed way, considering mathematical (expressiveness), verification (properties and algorithms) and software engineering (reusability and compositionality) point of view.

6. REFERENCES

[1] Divine – Distributed Verification Environment. http://anna.fi.muni.cz/divine.

[2] J. Adamek and F. Plasil. Behavior protocols capturing errors and updates. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE 2003), ETAPS*, pages 17–25, Warsaw, Poland, April 2003. University of Warsaw, Poland.

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

[4] J. Barnat, L. Brim, I. Černá, and P. Šimeček. Divine – The Distributed Verification Environment. In *Proceedings of the Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, July 2005.

[5] 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, 2003.

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

[7] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

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

[9] 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. ACM Press, New York, 1997.

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

[11] D. C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *Proceedings of DIMACS Partial Order Methods Workshop IV*, July 1996.

[12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[13] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of PODC*, pages 137–151, April 1987.

[14] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC'95)*, September 1995.

[16] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1)*, February 1999.

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

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