

COMPONENT COMPOSITION VALIDATION

ANDREAS SPECK*, ELKE PULVERMÜLLER**, MICHEAL JERGER***, BOGDAN FRAN CZYK****

* Intershop Research, Intershop Tower
D–07740 Jena, Germany
e-mail: a.speck@intershop.com

** Fakultät für Informatik, IPD, Universität Karlsruhe
D–76128 Karlsruhe, Germany
e-mail: pulvermueller@acm.org

*** sLAB oHG, Otto-Lilienthal Str. 36
D–71034 Böblingen, Germany
e-mail: jerger@jerger.org

**** Institute of Technical and Computer Education
University of Zielona Góra
65–762 Zielona Góra, Poland, and
Universität Leipzig
Lehrstuhl für Informationsmanagement
D–04109 Leipzig, Germany
e-mail: franczyk@wifa.uni-leipzig.de

Many approaches such as component technologies have been invented in order to support software reuse. Based on these technologies a large variety of techniques have been introduced to connect components. However, there is little experience concerning the validation of component systems. We know how to plug components together, but we do need ways to check whether that works. In this paper we introduce an approach to validating component compositions and showing how such a process can be supported by tools. We introduce a way to compare the interface specification of components automatically against the code. Furthermore, we demonstrate how compositions of components can be specified by logical formulas, allowing us to automatically validate these compositions.

Keywords: composition verification, component composition, feature interaction, model checking

1. Introduction

Component-driven approaches to software development are becoming more and more important. As with objects, there exist various definitions of components, and the one best known is given in (Szyperski, 1997). Components and their compositions are supported by various technologies such as CORBA, COM/DCOM (COM+) or Java Beans. Components are more than an issue of research. At the moment various commercial vendors provide components as building blocks (e.g., Enterprise Beans) for industrial systems, trying to establish component marketplaces (Componentsource, 2001).

Building systems from components allows us to reuse software units on an industrial scale. It is possible to buy, instead of making, parts of the whole system, and

to compose these parts which may come from different vendors. Reusing components has the potential to reduce time-to-market ratios and to improve quality. The composing can be realised by structuring sets of functionality (sub-components) to form a larger component (super-component) forming a hierarchy, or by inserting communication between components.

A major problem regarding components is that the mechanisms of component combination on the code-level are well known, while there is little experience concerning the modeling of relationships between components and their communication.

In this paper, we present an approach to modeling components and component compositions. Moreover, we provide a technique for defining the communication se-

quences between and inside the components. The composition and its dynamic interactions can be validated by a verification tool based on symbolic model checking.

Section 2 presents the basic component model of the approach and demonstrates how single components can be automatically verified against their specification. In Section 3 general issues of composition validation are discussed, and our composition model and validation process are introduced. Some related work can be found in Section 4.

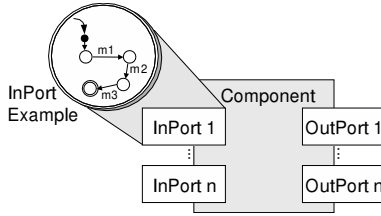


Fig. 1. Component with *InPorts* and *OutPorts*.

2. Component Validation

Before we start validating communication between components, first we have to check each component by comparing its specification with implementation.

To describe the capabilities of a component, we distinguish the component’s input (represented by *InPorts*), its output (*OutPorts*) and its internal behavior (c.f., Fig. 1). The concept of the interface specification with *InPorts* and *OutPorts* is based on the module interface concept (Gauthier and Pont, 1970). Its application is described in various papers (Lauder and Kent, 1999). *InPorts* and *OutPorts* are finite state machines which represent the protocol of the input and output communication of a component.

2.1. Description of the Internal Behavior

The internal behavior of components is triggered by external requests according to the protocol defined by the component’s *InPorts*, and has to result in responses of the component defined by the *OutPorts*.

There are several ways of describing the internal control flow between ports: finite state automata like in *Co-CoNut* (Heuzeroth and Reussner, 1999), which is based on the experiences of the application of finite state automata in the design and verification of network communication protocols (Holzmann, 1990), or UML sequence diagrams (Vanderperren and Wydaeghe, 2001). Since the automatic generation of UML sequence charts is supported by various tools, the second alternative is advantageous; hence we concentrate on this alternative.

The internal activities of components are described as follows: We start from each state in the component’s *InPorts* and connect them with their resulting *OutPort* states:

$M := \{m\}$, m is a symbol of the input alphabet of the *InPort* automata; we consider m as a message;

$R := \{r\}$, r is one symbol of the output alphabet of all *OutPort* automata of the respective component;

$C := \{c\}$, c is a Boolean expression.

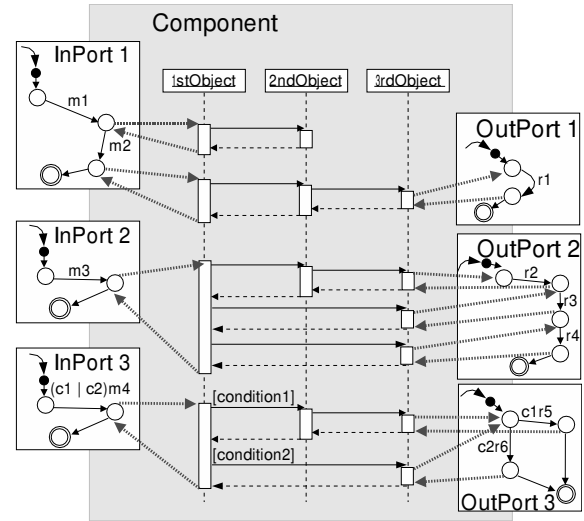


Fig. 2. Communication sequences between *InPorts* and *OutPorts*.

In general, we can distinguish four different cases, which are depicted in Fig. 2 (the figure presents these activities in an arbitrary temporal order):

1. **Empty output** (no resulting response):

$$f: M \longrightarrow \varepsilon. \quad (1)$$

For example, the message $m1$ causes no response:

$$m1 \mapsto \varepsilon.$$

2. **Exactly one response:**

$$f: M \longrightarrow R. \quad (2)$$

The message $m2$ results in one response:

$$m2 \mapsto r1.$$

3. **Sequence of responses:**

$$f: M \longrightarrow R^*, \quad (3)$$

$$R^* = \{r_{k_0} r_{k_1} \dots r_{k_n} \mid r_{k_i} \in R \cup \{\varepsilon\};$$

$$k_i \in \{0, 1, 2, 3, \dots\}\}.$$

If $n = 0$, Cases 1 and 2 are covered by the above equation.

A sample reaction to message $m3$ can be

$$m3 \mapsto r2r3r4.$$

4. Branched response:

$$f: C \circ M \longrightarrow C \circ R^*. \quad (4)$$

An extension of (4) results in a more generic equation, which covers Cases 1, 2 and 3:

$$f: C' \circ M \longrightarrow C' \circ R^*, \quad (5)$$

$$C' := C \cup \{\varepsilon\}.$$

The reaction to the received message $m4$ may result in any sequence of responses depending on the condition $[c]$. An example can be

$$c1m4 \mapsto c1r5,$$

$$c2m4 \mapsto c2r6.$$

In addition, conditions may be the means to express and thus check the synchronisation of component execution.

In order to compress the internal communication of a component (c.f., Fig. 3), *InPorts* and *OutPorts* may be connected directly, considering only the order of state transitions (in *InPorts* and *OutPorts*) triggered by incoming messages. Such a compression of internal information¹ leads to better comprehension of the component capabilities and an overview of the components' capabilities without being overwhelmed with internal details. The possibility of hiding the internal details is crucial for the concept of super-components as introduced in Section 3.2.

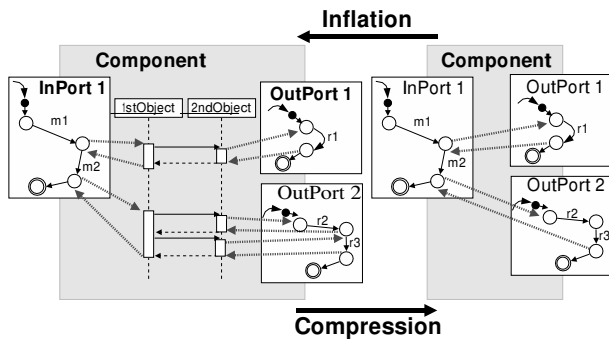


Fig. 3. Compression of internal communication.

¹ We would like to call it compression according to (Coplien, 2000) since compressing implies that the information is not lost but may be restored when required.

2.2. Verification of Internal Communication Components

Before validating a component system, the components themselves have to be verified, which means comparing the components' inputs and outputs with the *InPort* and *OutPort* specifications. The internal communication of the components is implicitly checked here.

The concrete steps are as follows: The input interface relying on the *InPort* documentation is verified. This can be done simply by comparing the real interface in the code with the specification. A specific order between the method calls which correspond to an *InPort* automaton has to be indicated by annotations in the code.

In the second step the *InPort* automaton is extended by the internal communication sequences (c.f. the sequence chart in Fig. 2). The resulting extended *InPort* automaton is then compared with the corresponding *OutPort* automaton. Such a comparison can be automatically performed with a model checking tool.

We apply *Mealy* automata (Kohavi, 1978). However, *Moore* automata (Moore, 1956) may be applied as well, since they are equivalent to the *Mealy* automaton (Hopcroft and Ullman, 1979). A *Mealy* automaton M is a quintuple $M = (I, O, S, \delta, \lambda)$ with I, O, S as non-empty finite sets of input and output symbols as well as states, respectively:

input function $\delta: I \times S \rightarrow S$ (state transition),

output function $\lambda: I \times S \rightarrow O$,

while $I := C' \cup M$ and $O := C' \cup R^*$ is valid.

The extensions of the *InPort* automaton follow the rules given in Section 2.1. Extensions of the examples are:

$$m1 \mapsto \varepsilon,$$

$$m2 \mapsto r1,$$

$$m3 \mapsto r2r3r4,$$

$$c1m4 \mapsto c1r5,$$

$$c2m4 \mapsto c2r6.$$

The extension of the *InPort* automaton M_I is performed by projection Π_{O_i} . This projection is defined as follows: If $M = (I, O, S, \delta, \lambda)$ is a *Mealy* automaton and B is the set of symbols, the projection Π_B on the automaton M is

$$\Pi_B: M \longrightarrow M',$$

$M' := (I, O', S, \delta, \lambda')$ with $O' := O \cap B$ and $\lambda' : (i, s) \rightarrow o$, where $i \in I, s \in S, o \in O$ and $o = \varepsilon$ if $o \notin B$.

The application of this projection on M_I results in

$$\Pi_{O_1}(M_I) := M'_I \text{ with } O' = \{r1\} \text{ and } \lambda' := \{(m2, s_0) \mapsto r1, (i, s) \mapsto \varepsilon\},$$

$$\Pi_{O_2}(M_I) := M''_I \text{ with } O'' = \{r2, r3, r4, r5, r6, c1, c2\} \text{ and } \lambda'' := \{(m3, s_1) \mapsto r2r3r4, (c1m4, s_2) \mapsto c1r5, (c2m4, s_2) \mapsto c2r6, (i, s) \mapsto \varepsilon\}.$$

Various alternatives exist for the comparison of the extended *InPort* automaton with the *OutPort* automaton; an example can be state equivalence, which is applied by symbolic model checking.

The result of these approaches is the same. Since we applied the model checking tool *RAVEN* (Ruf, 2001), we use state equivalence. In (McMillan, 1992) state equivalence is defined as: "... the greatest relation between states such that if x is equivalent to y , then for all inputs, the output in state x is equal to the output in state y , and the successor state of x is equivalent to the successor state of y ."

In order to achieve state equivalence, we reduce the definition area of automata M_{O_1} and M_{O_2} to the representative input sequence of the extended *InPort* automaton.²

Figure 4 depicts the *RAVEN* input module defining Case 4 of Section 2.1. The branched response $c1m4 \mapsto c1r5$; $c2m4 \mapsto c2r6$ is processed by a specific automaton. Additionally, another automaton is defined which generates random conditions $c1$ or $c2$ and triggers the branched response with either the message $c1m4$ or $c2m4$.

The command line call and the execution result of *RAVEN* are shown in Fig. 5.

3. Component Interactions

The goal of our approach is not only to verify single components versus their specifications. Moreover, we would like to describe and validate the composition of components. In this section we first introduce the model of component composition, and then present our approach to validating such a composition.

3.1. Validation Issues

The description of a component architecture in logical formulas provides tool support for the validation of a composed system. Some important questions that may be automatically answered are by applying model checking:

² Alternatively, the input of M'_I can be simulated by another automaton M_{input} , with $I_{input} := \{\varepsilon\}$. The pair of automaton M_{input} connected with M'_I can be proven to be in state equivalence with the *OutPort* automaton.

```

MODULE input
  SIGNAL s : { s1 s2 s3 s4 s5 end }
  INPUT  c  := random.s
         m3 := inmessage.m3
         m4 := inmessage.m4

  DEFINE
    r2 := (s=s1)
    r3 := (s=s2)
    r5 := (s=s4)
    r6 := (s=s5)
    c1 := (s=s3) & c
    c2 := (s=s3) & !c

  INIT s=s1
  TRANS
    | - s=s1 -- (m3) --> s:=s2
    | - s=s2 -- TRUE --> s:=s3
    | - s=s3 -- (m4) & c --> s:=s4
    | - s=s3 -- (m4) & !c --> s:=s4
    | - s=s4 -- TRUE --> s:=end
    | - s=s5 -- TRUE --> s:=end
    | - s=end-- TRUE --> s:=end

  END

  SPEC
    eq := AG((output.r2=input.r2) &
              (output.r3=input.r3) &
              (output.r5=input.r4) &
              (output.r6=input.r6) &
              (output.c1=input.c1) &
              (output.c2=input.c2))

```

Fig. 4. Input for *RAVEN*.

```

examples/ril> raven raven_sample.txt
parse
init encodings
expand
compose
reduce
minimize
check eq:
  use time abstraction
  eq is true
RESOURCES:
time           = 0.06 sec
parsing time   = 0.03 sec
composition time = 0.02 sec
minimization    = 0.01 sec
-----
BDD nodes for trans : 29
before minimization : 29 (100%)
PROCESS-MEMORY    : 1.3 MB

```

Fig. 5. Execution of the *RAVEN* example.

- Is it possible that two specific components may interact?
- Which environment is needed for a specific component?
- Could specific paths (e.g., methods of component objects) or *InPort* and *OutPort* states be reached within a composition?

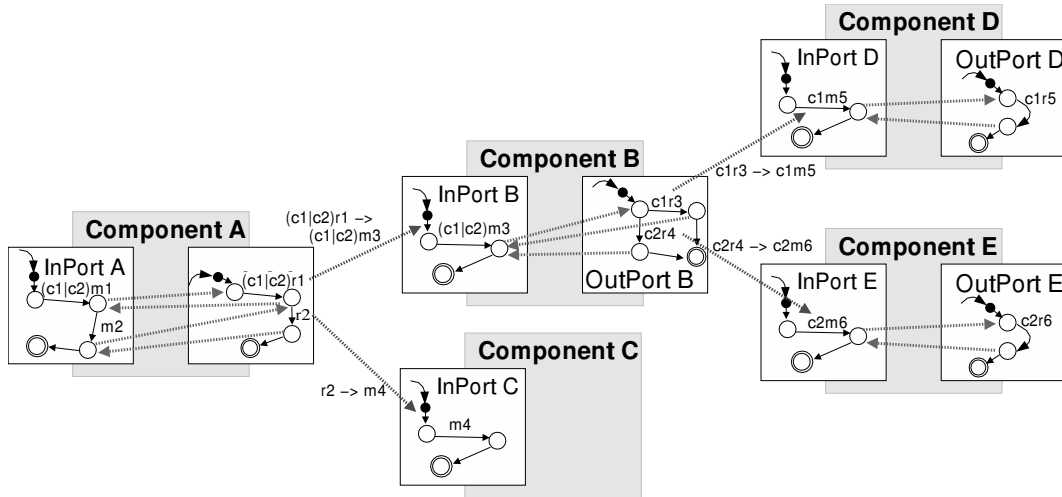


Fig. 6. Communication paths through components.

The first question has already been addressed by the *CoCoNut* approach. In (Schmidt and Reussner 2000) the adaptability of components concerning their interfaces was investigated. The proposed mechanisms can be applied to our approach, too.

The second question focuses on the services a component requires from the other components of a system, or if there are any components in the system that do negatively interfere. This can be verified by checking the conditions directly related to the specific component.

In the remainder of this paper, we will focus on the verification of paths within a composed system, which is a very important problem. Figure 6 depicts a set of communicating components. In this example it might be of interest which condition (*c1* or *c2*) connected with message *m1* at *InPort* results in which response (either *c1r5* of *Component D* or *c2r6* of *Component E*).

Figure 6 shows a uni-directional communication. Renaming component *D* with *A* would show what a two-way interaction would look like. However, there is no need to treat component *A* as two separate components. That is to say, in this case, merging and mapping of component *A* and component *D* would be possible and useful. Potential cycles (in the sense of endless loops) have to be resolved by a fix-point analysis, or (if this is impossible) by time-out or other appropriate means.

3.2. Composition Model

The basis of composition is the validated specification of the components (cf., Sections 2.1 and 2.2). The composition can be described by logical formulas. These logical formulas of a composition can be automatically compared with the specific knowledge related to a component. For

example, a communication component may bear knowledge in which context it may be applied. The knowledge is expressed in logical operations. A detailed example of this approach can be found in (Klaeren *et al.*, 2000).

These logical operations constitute the basis for a more finely grained definition of the relation between components. The composition itself can be defined as the quintuple

$$CM := (IP, OP, A, \tau, C)$$

with a set of *InPorts* *IP*, a set of *OutPorts* *OP*, an *Actor* $A \in OP$, a set of transitions $\tau: OP \rightarrow IP$ and the conditions for the composition *C*.

The *Actors* are other components or human beings sending messages to the composed system triggering certain behavior. The transitions represent communication via the relationships between components. In each composition the output of the *OutPort* automaton of an *OutPort* o_l of a transition $\tau_m: o_l \mapsto i_k$ has to be accepted by an *InPort* automaton of the cooperating component's *InPorts* i_k . Composition conditions can be used to express rules of a composed system's validity (c.f., Section 3.2.2).

This composition concept has been derived from an approach to combining aspects with components described in (Pulvermüller *et al.*, 2001). Details of component combination can be found in (Speck and Pulvermüller, 2001).

3.2.1. Super-Components

In our approach we apply two abstract types of component composition representing real component composition mechanisms. First, components are hierarchically structured into super-components and sub-components. A well-known principle in software engineering of dealing

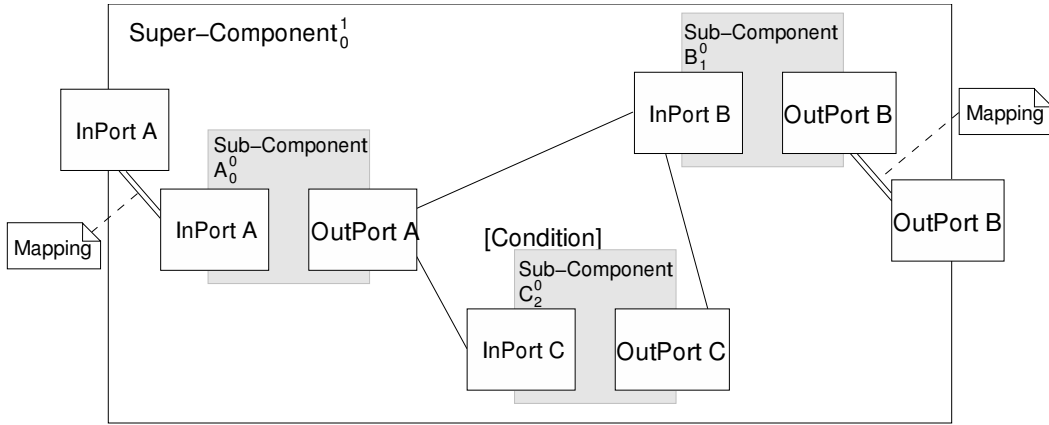


Fig. 7. Component composition example.

with complexity is modularizing and forming hierarchies. Our approach is based on these hierarchies independent of whether these hierarchies are on the level of objects or sub-systems. In the following, we abstract from these levels by using the terminology *sub-components* and *super-components*.

Second, components may be composed (or interact) on the same level (e.g., the sub-components A_0^0 , B_1^0 and C_2^0 in Fig. 7). In much the same way as for the hierarchical issue, we abstract from the technology realizing the communication.

The advantage of the abstract description of component relationships is that different mechanisms are captured which may be mapped on both abstract composition relationships. Specific mechanisms and levels may have the need to solve additional detailed problems. However, this will always be only a specialized refinement of the presented approach.

The application of super-components as super-sets containing one or more composed sub-components is quite a common approach used in software engineering to express the hierarchy. The distinction between these two types of components (super- and sub-component) allows us to define different levels of granularity.

A component can be defined labelled as C_i^k , where k represents the granularity (0 represents the lowest level of granularity) and i gives the index distinguishing between components on the same level of granularity within the same super-component (c.f., Fig. 7).

The advantage of the concept of a super-component containing a set of other components is that it allows us to minimize the number of interfaces of this set of components. The sum of all interfaces of the contained components does not have to be taken into account when the component set is considered; only the interfaces defined by the super-component do.

The concept of super-components containing sub-components and the relationships between components of the same level can be realized by the COM mechanisms *Containment* and *Aggregation*, see, e.g., (Szyper-ski, 1997).

3.2.2. Composition Conditions

In the same way as we used conditions to express decisions in branched responses within components (c.f., Section 2.1, Case 4.) we now use conditions to define the validity of a component within a given context. The composition conditions can be considered as a special case of more dynamic branched response conditions. For example, if there is no branch to a specific component within a super-component, the composition condition for this component can be that this component must not be part of the super-component.

Composition conditions are concepts defining different component versions within a system family (Pulvermüller et al., 2001). This leads to an inductive definition of the construction of components:

$$\text{Component } C_i^0 = 1 \wedge \text{Cond}_i^0,$$

where Cond_i^0 represents the condition³ that has to be true for one version C_i^0 on level 0.

We have $C_i^0 \in C^0$, where

$$C^0 = \left\{ \bigcup_{j=1}^{\infty} C_j^0 \right\}$$

is the set of all versions on level 0.

Similarly, we can define the induction step:

$$\text{Super-component } C_i^{(n+1)} = \bigwedge_{j=1}^m C_j^n \wedge \text{Cond}_i^{(n+1)}$$

³ Several conditions may be unified in one condition.

with

$$\text{Cond}_i^{(n+1)} \text{ is true, } 1 \leq l \leq m \leq |C^n|, \quad C_j^n \in C^n.$$

A super-component is a set of conditions (unification of the particular conditions of a certain super-component and all conditions of the sub-components contained in the super-component). Each condition is presented as a Boolean expression.

An example of a super-component can be

$$C_1^{(2)} = (C_1^{(1)} \wedge \neg \text{Cond}1) \wedge (C_1^{(1)} \wedge \neg \text{Cond}1) \vee \text{Cond}4.$$

3.2.3. Temporal Relationships

When the set of components within a super-component is defined, the temporal relationships among these components have to be taken into account. The temporal sequence of communication between the components are modeled with sequence charts. Since a super-component may be considered as a component on its own, the super-component's internal relations may be defined as depicted in Fig. 2.

Symbolic model checking applies temporal logic in order to express the temporal order of the sequences of communication between the components (McMillan, 1992).

3.3. Validation Procedure

In order to verify the composition of component systems, we apply the model checking tool *RAVEN*. The composition (c.f., Section 3.2) is expressed in *Computation Tree Logic* (CTL), which is defined in (McMillan, 1992).

Some examples of requirements described with CTL may be:

- “If Actor *A* sends the message *B.m1* it can trigger the function *C.r1* in the component *C*.”

$$\text{CTL : } B.m1 \rightarrow E(\text{true } U \text{ } C.r1),$$

- “If Actor *A* sends message *B.m2* it must trigger the function *C.r2* in component *C*.”

$$\text{CTL : } B.m2 \rightarrow A(\text{true } U \text{ } C.r2).$$

According to the composition model of a component system, the relations and transitions between the components (c.f., Section 3.2) have to be modeled in CTL. In general, this can be done as with the communication sequences between *InPorts* and *OutPorts*. We have already introduced this procedure in Section 2.2. The rules applied to the internal communication of components have

to be extended in order to describe inter-component communication to avoid name clashes. Additionally, the *InPort* name and component ID of each message are to be attached to the message name. Therefore the *InPort* name and component ID specify a name space for each message which allows us to identify it explicitly. It separates messages with the same names by adding the corresponding *InPort*.

4. Related Work

CoCoNut (Heuzeroth and Reussner, 1999) is an approach which may be considered as complementary. *CoCoNut* uses *InPort* and *OutPort* automata (Lauder and Kent, 1999) in the same way to express the external communication of components. In contrast to our approach, the internal activities of components are modeled with finite state automata. This concept of applying finite state automata to modeling and verifying has already been used in the domain of communication protocols and networking (Holzmann, 1990).

The *OutPorts* are developed by inserting the finite state machines describing the internal behavior in the *InPort* automata which may lead to a large number of states (instead of projection). Compositions are modeled by merging the automata of the components. If components within a composed system are exchanged, such large finite state machines are used to determine adaptors for new components (Schmidt and Reussner, 2000).

A similar approach is presented in (Vanderperren and Wydaeghe, 2001). However, here all interactions between and within the components are specified by UML sequence diagrams.

Our concept of modeling and validating component compositions supports design processes such as those discussed in (Tekinerdogan, 2000). It gives a comparatively fine-grained model for component relationships and provides tool support for validating design decisions.

Component composition can be implemented using various approaches. Generators as they are proposed in (Czarnecki and Eisenecker, 2000) can be applied to combine the components. Focused on the GenVoca architecture, in (Batory and Geraci, 1997) an approach to composition verification is presented. However, this approach is more coarsely grained and mainly concentrates on layered systems. Moreover, the dynamic issues are not considered.

Aspect-Oriented Programming AOP (Kiczales *et al.*, 1997) and *Subject-Oriented Programming* SOP (Ossher and Tarr, 1999) provide new concepts of the modularisation and structuring of component systems. A concept of model component-based systems according to these new approaches is presented in (Pulvermüller *et al.*, 2001).

The approach proposed in this paper can be implemented by CORBA or COM/DCOM (COM+). However, the usage of the COM *IUnknown* interface cannot be applied to the implicit delegation of calls since only explicit communication can be verified.

5. Conclusion

The paper discusses an approach to defining and validating component compositions. The approach we propose allows us to reuse the already existing components in order to construct new systems. The process of constructing such component systems is as follows:

1. We identify the components we wish to use in component systems. Therefore we examine the existing components by comparing them with their *InPort* and *OutPort* specifications. This is supported by tools.
2. The information gained in the first step is then used for the design of a component system. Such a design (like most design processes) has to be done manually. However, we provide a specific notation based on logical equations to express the system design. The structure of the component system is given by a super-component mechanism and dynamic communication sequences.
3. The composition can then be validated by verifying specific properties. Examples can be: Are two (or more) components properly interacting? What is the environment needed for a component? Are the specific paths or states in *InPorts* or *OutPorts* reached within a composition with a given set of conditions? These checks are tool-supported.

This concept intensively supports the development of component-based systems. The existing components can be reused. Components are grouped within super-components with a limited and defined set of interfaces which increases system modularity and therefore reduces system complexity.

Additional work in the future can be done by applying temporal logic as extensions of sequence diagrams. The temporal logic can be a base for additional verification of dynamic behavior. Another important issue which has to be addressed is the fact that it is not always possible to express component relationships with crisp logical formulas. A possible solution can be the application of fuzzy techniques in order to deal with only partially specified components.

Our first prototype implementations work with Java Bean components of a Jini system. Prior to their composition with Jini, the components were verified. Based

on the components' verification the intended composition was validated.

Currently we assess the proposed approach by applying it to the area of e-business systems. In this domain there is a high demand for flexible and reliable systems built of software components.

References

- Batory D. and Geraci B. (1997): *Composition validation and subjectivity in GenVoca generators*. — IEEE Trans. Softw. Eng., Vol. 23, No. 2, pp. 67–82.
- Componentsource (2001): *Marketplace and community for software components*. — Available at: <http://www.componentsource.com/>.
- Coplien J. (2000): *Data Compression versus abstraction*. — Private communication.
- Czarnecki K. and Eisenecker U. (2000): *Generative Programming—Methods, Tools, and Applications*. — New York: Addison-Wesley.
- Gouthier P. and Pont S. (1970): *Designing Systems Programs*. — Englewood Cliffs: Prentice Hall.
- Heuzeroth D. and Reussner R. (1999): *Dynamic coupling of binary components and its technical support*. — Proc. GCSE'99 Young Researchers Workshop, Erfurt, pp. 30–31.
- Holzmann G. (1990): *Design and Validation of Computer Protocols*. — Englewood Cliffs: Prentice Hall.
- Hopcroft J. and Ullman J. (1979): *Introduction to Automata Theory, Languages and Computation*. — New York: Addison-Wesley.
- Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M. and Irwin J. (1997): *Aspect-oriented programming*. — Proc. European Conference on Object-Oriented Programming, ECOOP'97, Jyväskylä, Finland, Berlin: Springer, LNCS 1241, pp. 220–241.
- Klaeren H., Pulvermüller E., Rashid A. and Speck A. (2000): *Aspect composition applying the design by contract principle*. — Proc. 2nd Int. Symp. Generative and Component-based Software Engineering (GCSE 2000), Erfurt, Germany, Berlin, Springer, LNCS 2177, pp. 57–69.
- Kohavi Z. (1978): *Switching and Finite Automata Theory*. — New York: McGraw-Hill, 2nd Edition.
- Lauder A. and Kent S. (1999): *EventPorts: Preventing legacy componentware*. — Proc. 3rd Int. Conf. Enterprise Distributed Object Computing Conference (EDOC 99), Mannheim, Germany, IEEE Publishing, pp. 224–232.
- McMillan K. (1992): *Symbolic Model Checking*. — Ph.D. Thesis, Carnegie Mellon University.
- Moore E. (1956): *Gedanken-experiments on sequential machines*. — Ann. Math. Stud., Vol. 15, No. 4, pp. 129–153.

- Ossher H. and Tarr P. (1999): *Using subject-oriented programming to overcome common problems in object-oriented software development/evolution.* — Proc. 1999 Int. Conf. Software Engineering, ICSE, Los Angeles CA, ACM Press, pp. 687–688.
- Pulvermüller E., Speck A. and Coplien J. (2001): *A version model for aspect dependency management.* — Proc. 3rd Int. Conf. Generative and Component-based Software Engineering (GCSE 2001), Erfurt, Germany, Berlin: Springer, LNCS 2186, pp. 70–79.
- Ruf J. (2001): *RAVEN: Real-time analyzing and verification environment.* — J. Univ. Comp. Sci., Vol. 7, No. 1, pp. 89–104.
- Schmidt H. and Reussner R. (2000): *Automatic component adaption by concurrent state machine retrofitting.* — Tech. Rep., No. 2000/81, School of Computer Science and Software Engineering, Monash University, Melbourne.
- Speck A. and Pulvermüller E. (2001): *Versioning in software engineering.* — Proc. 27th Ann. Conf. IEEE Industrial Electronics Society, IECON'01, Denver, CO. — IEEE Computer Society Press, pp. 1856–1861.
- Szyperski C. (1997) *Component Software.* — New York: Addison-Wesley, ACM-Press.
- Tekinerdogan B. (2000) *Synthesis-based software architecture design.* — Ph.D. Thesis, Dept. Computer Science, University of Twente, Enschede, the Netherlands.
- Vanderperren W. and Wydaeghe B. (2001): *Towards a new component composition process.* — Proc. 8th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS), Washington DC, IEEE Press, pp. 322–329

Received: 10 October 2001

Revised: 24 April 2002