STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LI**, Number 1, 2006

AUTOMATA-BASED COMPONENT COMPOSITION ANALYSIS

ANDREEA FANEA, SIMONA MOTOGNA AND LAURA DIOŞAN

Abstract. Formal specification of software components enables automatic composition and checking of component-based systems. The component system is modeled as a finite automaton. We propose an algorithm that builds all syntactically correct finite automata-based models of a component-based system. The result systems are checked against the properties: lost data and number of provider/inport.

1. Introduction

Component-based software engineering (CBSE) is the emerging discipline of the development of software components and the development of systems incorporating such components. In order to construct a correct system, these components should be assembled in an unified model and we would like to be able to obtain properties of the model that could contribute to its correctness.

A formal model for component-based software is of critical importance because it provides a basis for the understanding of the underlying concepts of component models, component certification techniques, component testing. The general definition of a software component is given in [5].

There are two issues which need to be addressed [1], [6] where a software system is to be constructed from a collection of components:

- Component integration - the mechanical process of wiring components together. There has to be a way to connect the components together.
- (Behaviour) Component composition - we have to get the components to do what we want. We need to ensure that the assembled system does what is required. Component integration is taken one step further to ensure that assemblies can be used as components in larger assemblies.

To achieve integration, syntactical composition is studied. It offers the necessary tool to meet for the requirements for wiring components together. Component integration is a more complex process which will need to assign also semantic

Received by the editors: 9.01.2006.

2000 *Mathematics Subject Classification.* 68N30, 68Q45.

1998 *CR Categories and Descriptors.* code I.6.4 [**Simulation and modeling**]: – *Model Validation and Analysis*; code I.6.5 [**Model Development**]: – *Modeling methodologies*.

information regarding behaviour to the syntactic entities. This will be the next step in our study and it is not treated here.

We have developed in [2] an algorithm that computes all the possibilities of constructing a system from a given set of components, checking only the syntactical part when wiring together the components. This computation is the first step from the construction of a component-based system. The next step consist of the behaviour composition of components.

In [4] a model of a component-based software system is proposed, which uses a finite automata-based method, enabling compositional reachability analysis. The following checks were performed:

- the system is consistent: starting from a given input, all components can be added to the model and the execution eventually terminates;
- there are no potential deadlocks in the model.

This paper proposes a new algorithm to construct all the component-based software systems as finite automata-based models. The resulting models are syntactically correct. By syntactically correct model we mean no semantic involvement in the models, but just the way to connect the components together, the mechanical process of "wiring" components together (component integration).

A syntactically correct model has the following properties:

- all inputs are provided for a component to be executed; a component is added into the model if and only if all its necessary input data are provided;
- there are no cyclic dependencies;
- no duplicate components.

The algorithm checks model consistency during its construction from a given set of components. The result systems are checked against the properties: if the execution of the component system is terminated and even if the system behaves properly, there is some lost data, and a component is not allowed to receive the value for an inport from more than one component - one provider/inport. A comparison analysis of three solutions (with different properties) are presented and some examples are given.

2. Previous results

In [4] the component system is modeled as a finite automaton, where components are represented as states and information flows as transitions.

Definition 1. *A source component, i.e. a component without inports, is a component that generates data provided as outports in order to be processed by other components.*

Definition 2. *A destination component, i.e. a component without outports, is a component that receives data from the system as its inports and usually displays it, but it does not produce any output.*

Definition 3. *A system of components is defined as a finite automaton* $A = (Q, \Sigma, \delta, q_0, F)$*, where:*

- Q *is the set of states, each* $q \in Q$ *representing a component;*
- Σ *is the input alphabet; in the proposed model,* Σ *is the union of the outports (of the components) already included in the models, in fact, the data set;*
- $\delta : Q \times \Sigma \to P(Q)$ *is the transition function;* δ *members have the form* $((C_1, d) \to C_2)$*, where* C_1,$C_2 \in Q$ *and* $d \in outports(C_1) \bigcap inports(C_2)$*;*
- $q_0 \in Q$ *is the initial state - the source component in the component system;*
- $F \subset Q$ *is the set of final states - the destination components from the component system.*

In [4] the *MakeModel* algorithm has as input a component system specification and builds the model, a nondeterministic finite automaton. The algorithm generates such a model from a given component system specification, checking the following properties:

- all inputs are provided for the tasks of the C component to be executed, i.e. $inports(C) \in \Sigma$;
- there are no "cyclic" component dependencies: C_1 expects data d_1 as inport and provides d_2 as outport, and component C_2 needs d_2 as inport and provides d_1 as outport.

In [4] the procedure *Search(compList, cond, component, flag)* searches into the list of components *compList* for the first component satisfying a given condition *cond*. The output parameter *flag* is set to true if the search is successful. In this case, the component is also provided. If no component matching *cond* is found then *flag* is set to false. Because of the "first" criterion, only one nondeterministic finite automaton is constructed.

In [3] the following definition was introduced:

Definition 4. *a. A component C is reachable iff there exists a path from the source component to C. We say that* C' *is reachable from C through d if* $\delta(C, d) = C'$*;*
b. A data d is live iff for a reachable component C there exists a component C' reachable from C through d: $d \in outports(C) \cap inports(C')$*.*

We have modified the algorithm [4] in order to generate all the nondeterministic finite automata. Also, the final constructed system have only live data. This property is checked after building the consistent system (starting from a given input, all components are added to the model and the execution eventually terminates). The construction of the model is described in the following section.

3. Model building

We must first establish our entities involved in the component system definition:

- domain D - a set that does not contain the null element;
- set of attributes A - an infinite fixed and arbitrary set; the atributes signify variables or fields;
- type of an attribute $x \in A : Type(x) \in D$ represents the set of possible values for the attribute x.

Consider the component system $CS = \{C_1, C_2, ..., C_n\}$, in which every component C_k is specified as: $C_k = (compID_k, inports_k, outports_k, functs_k)$, where:

- $compID_k$ is the component identifier, unique;
- $inports_k \subseteq A$ the set of input ports;
- $outports_k \subseteq A$ the set of output ports;
- $functs_k$ the set of tasks the C_k component performs.

3.1. **Algorithm specification.** The specification of the **MakeAllModels** algorithm is as follows:

Begin

Input: the component system CS;

Output: all the nondeterministic finite automata A = (Q, Σ, δ, q_0, F).

End.

3.2. **Algorithm description.** We use a recursive backtracking algorithm to generate all the component-systems from the existing specified components.

The first component that is used from the component system is a source component. A component is added to solution (the intern conditions, specified in ***valid(i)***) if the component was not already used before and all the inputs of the component are provided for the tasks to be executed. A component-based system is found (the conditions for the complete solution, specified in ***solution(i)***) when the last component added to solution is a destination component.

The lost data property is checked only after a solution is generated, because when integrating a component into the systems we do not check if all the outputs are consumed (only some outputs are used for the transition to the current component). It is obvious that for a component in the solution all the inputs are consumed, because we used the condition that all the inputs are available. The property checks if all the outputs of all the components involved into the computation are consumed.

The necessity to provide all inputs of the component to be executed generates another condition to be checked after a solution is generated: the inputs of a component could be provided by more than one component and the choice is made in the algorithm. The following situation is not desired: the current component

Algorithm 1 BuildingAllModels Algorithm

```
1: for each component in the system do
2:   add the component to the solution on position i;
3:   mark the new set of available inputs;
4:   if valid(i) then
5:     for each mC ∈ CS do
6:       for each d ∈ inports(Component_i) do
7:         if d ∈ outports(mC) then
8:           δ := δ ⋃ {(mC, d) → Component_i};
9:         end if
10:      end for
11:    end for
12:    if not solution(i) then
13:      BuildingAllModels(i+1,...)
14:    else
15:      WriteSolution(i,...);
16:    end if
17:  end if
18: end for
```

receives the value for one of its inports from two different components. The algorithm will check at the end if a solution contains such situations. More precise explanations are presented in the next section.

4. Examples and result analysis

4.1. **Example 1.** Consider the following general set of components:

$C_1 = (C1, \varnothing, \{d_1, d_2\}, \{read\})$;
$C_2 = (C2, \{d_1, d_3\}, \{d_5, d_6\}, \{task_1, task_2, task_3\})$;
$C_3 = (C3, \{d_2\}, \{d_3, d_7\}, \{task_4\})$;
$C_4 = (C4, \{d_5, d_7\}, \{d_8\}, \{task_5\})$;
$C_5 = (C5, \{d_6, d_8\}, \varnothing, \{write\})$;
$C_6 = (C6, \{d_1, d_3\}, \{d_4, d_5, d_6\}, \{task_1, task_2, task_3\})$;
$C_7 = (C7, \{d_1, d_5\}, \{d_3\}, \{task_1, task_2, task_3\})$;
$C_8 = (C8, \{d_2, d_3\}, \{d_4\}, \{task_4\})$;
$C_9 = (C9, \{d_4\}, \{d_5, d_6\}, \{task_5\})$;
$C_{10} = (C10, \{d_6\}, \varnothing, \{write\})$;

The results of building the models from existing components are presented in Table 1: the number of all the consistent solutions, the number of solutions without lost data and the number of solutions with only one provider/input for all involved components and the number of final solutions. The percentage shows

that only a small part of the solutions should be taken into consideration based on the efficiency criterion.

TABLE 1. The number of solutions for the components set from example 1

Algorithm MakeAllModels	All Solutions	Solutions without Lost Data	Solutions one provider/input	Final Solutions
Number	1323	40	64	1
Percent	100%	3.02%	4.83%	0.07%

The presented solution from Figure 1.a has lost data. The $C6$ component has two outports that are not "consumed".

The solution is $A = (Q, \Sigma, \delta, q_0, F)$, where:

- Q={C1,C3,C2,C4,C6,C5}
- Σ ={d1,d2,d3,d7,d5,d6,d8,d4}
- δ={(C1,d2) → C3,(C1,d1)→ C2,(C3,d3)→ C2, (C2,d5) → C4,(C3,d7)→ C4,(C1,d1)→ C6,(C3,d3)→ C6, (C2,d6)→ C5, (C4,d8)→ C5,(C6,d6)→ C5}
- q_0={C1}
- F={C5}

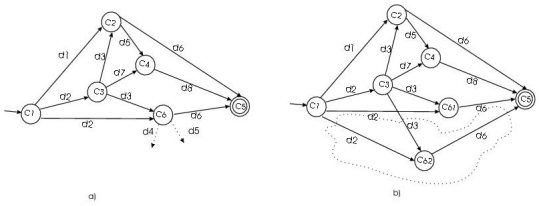

FIGURE 1. The nondeterministic finite automaton for the consistent solution a) with lost data d_4 and d_5 b) the correct model.

As Figure 1.a shows, component $C6$ outport contains data $d4$ and $d5$ which are lost (no other component from the system is using it). So we will split this component into two new components $C61$ and $C62$, clone its inports, data $d2$ and data $d3$, and then isolate the area containing component $C62$ and data $d2$, $d3$. The resulting model, as presented in Figure 1.b, is correct.

The presented solution is not included in the solution set with one provider/inport because the are two transitions to the same component $C5$ with the label $d6$ as in Figure 2. The input $d6$ of the $C5$ component must have only one provider on an

execution. The "reverse" propagation of data (the output data of a component is propagated to two or more components) is allowed. Component $C3$ distributes the data $d3$ to $C2$ and $C6$ component as in Figure 2.a. In Figure 2.b the final solution is presented: the solution is consistent, no lost data and each inport for each component has just one provider.

4.2. **Example 2.** Consider the following general set of components:
$C_1 = (C1, \varnothing, \{d_1, d_2, d_3\}, \{read\})$;
$C_2 = (C2, \{d_3\}, \{d_1\}, \{task_1, task_2, task_3\})$;
$C_3 = (C3, \{d_3, d_4\}, \{d_5\}, \{task_4\})$;
$C_4 = (C4, \{d_2\}, \{d_4\}, \{task_5\})$;
$C_5 = (C5, \{d_5\}, \varnothing, \{write\})$;
$C_6 = (C6, \{d_1\}, \{d_3\}, \{task_1, task_2, task_3\})$;

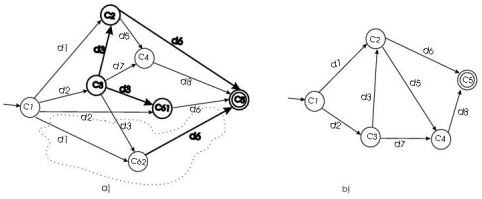

FIGURE 2. The finite automaton a) with more than one provider/inport; b) the corresponding final consistent solution.

The results of building the models from existing components are presented in Table 2.

TABLE 2. The number of solutions for the component set of Example 2

Algorithm MakeAllModels	All Solutions	Solutions without Lost Data	Solutions one provider/input	Final Solutions
Number	19	5	5	0
Percent	100%	26.31%	26.31%	0%

Figure 3 presents two solutions, one from the solution set without lost data figure 3a) and the other from the one provider/input solution set figure 3b). The consistent system $CS = \{C1, C4, C6, C3, C5\}$ from the 1a) side contains an output that is not "consumed": the data $d3$ is lost. On the right hand side there is a consistent system that has a component with more than one provider for an input: component $C3$ receives the data $d3$ from component $C1$ and from $C6$.

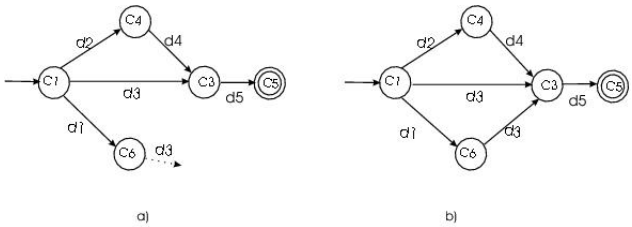

FIGURE 3. A finite automaton for the consistent solution a) without lost data; b) with more than one provider/inport.

5. Conclusions and future work

In this paper we proposed a new algorithm for computing all the component-based systems as automata-based models from a set of specified components. We analyse the final models from distinctive perspective, checking the existence of the properties lost data and one provider/port.

Using the same component model we intend to extend the algorithm and to address the following topics in the future: checking if the model supports a given sequence of tasks and building a component-based system that contains a given sequence of tasks. Also checking the behaviour of components (to ensure that the assembled system does what is required) after syntactic composition is intended to be studied.

References

[1] Ivica Crnkovic and Magnus Larsson, *Building Reliable Component-Based Software Systems*, Artech House publisher, 2002

[2] A. Fanea, S. Motogna, *A Formal Model for Component Composition*, Proceedings of the Symposium "Zilele Academice Clujene", 2004, pp. 160-167

[3] S. Motogna, B. Parv, D. Petrascu, Finding Errors in a Component Model Using Automata Techniques, 5th Joint Conference on Mathematics and Computer Science, Debrecen, Hungary, 2004, pp. 69

[4] B. Parv, S. Motogna, D. Petrascu, Component System Checking Using Compositional Analysis, Proceedings of the International Conference on Computers and Communications, 2004, Baile Felix Spa-Oradea, Romania, 2004, pp. 325-329

[5] Szyperski C. et al., Component Software, Beyond Object-Oriented Programming, 2nd ed., ACM Press, Addison-Wesley, NJ, 2002.

[6] Robert John Walters, *A Graphically based language for constructing, executing and analysing models of software systems*, University of Southampton, Faculty of Engineering and Applied Science Electronics and Computer Science, PhD Thesis, 2002

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

E-mail address: {afanea, motogna, lauras}@cs.ubbcluj.ro