CARPATHIAN J. MATH.
20 (2004), No. 1, 197 - 203

Automata-Based Compositional Analysis of Component Systems. Design and Implementation Issues

Bazil Pârv, Simona Motogna and Dragoş Petraşcu

Abstract. Specifying a real-world component system is a complex manual process. It is essential to be able to verify the correctness and robustness of its behavior, before it becomes operational. We propose a new way of specifying a component system, based on the concept of interface, that can be seen as a tool from analysis and design to programming and for type specification. We construct an algorithm that constructs the model, and can be used to deduct properties about the system: correctness, termination, deadlock free.

1. Introduction

Component based systems offer strong tools for developing large applications, that usually are developed in teams. In order to construct a correct system these components should be assembled in an unfied model, and, desirable, we would like to be able to obtain properties about the model, that could contribute to its correctness. The development of a formal model for components provides an exact analysis of component characteristics and qualities, such that we can reason about the correctness, efficiency and and interoperability of the system. Such a model should be designed at an early stage, enabling the software engineers and programmers to make correct and useful decisions.

Starting from an intuitive model of components (proposed in [4]), we construct a more exhaustive specification, that will better describe component structure and interactions, based on the observation that the most important part of a component is its interface – what is the component providing.

We will discuss in brief the first attempt of constructing a component model, then we will propose a more sophisticated way to built it, and in the end we will discuss the benefits of this improved algorithm.

2. Definition of the model

We start by presenting the initial system specification.
The basic notions involved in a component definition are [1]:

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

Received: 26.09.2004; In revised form: 15.10.2004
2000 *Mathematics Subject Classification.* 68Q45, 68T10.
Key words and phrases. *Component-based systems, formal models, finite automaton.*

Considering X a component over the set A of attributes, we will use the following notations [1]:

- **inports**$(X) \subseteq A$ - represents the set of input ports (attributes) of the component X
- **outports**$(X) \subseteq A$ - represents the set of output ports (attributes) of the component X
- **attributes**$(X) \subseteq A$ - represents the set of attributes of component X

$$\mathbf{inports}(X) \cap \mathbf{outports}(X) = \emptyset$$

Thus, we may describe the behavior of the component as producing results to the output ports as an answer to the input received from the input ports; the ports represent the interface through which the component interact with other components. The behavior is incorporated in the component functionality, described as a set of tasks.

Definition 2.1. A *component* over a set A of attributes can be:

- a ***source component***: is a component without inports, that generates data provided as outports in order to be processed by other components; if X is a source then:

  $$\mathbf{attributes}(X) = \mathbf{outports}(X) = \{X\} \qquad \mathbf{inports}(X) = \emptyset$$

- a ***destination component***: is a component without outports, that receives data from the system as its inports and usually displays it, but it doesn't produce any output; if X is a destination then:

  $$\mathbf{attributes}(X) = \mathbf{inports}(X) = \{X\} \qquad \mathbf{outports}(X) = \emptyset$$

- a ***simple component***, performs a single task;
- a ***compound component***: a group of connected components, in which the output of a component is used as input by another component from this group.

Definition 2.2. We suppose that the specification of the components from the system is given in the following form:

Component_id:
Inport: . . .
Outport: . . .
Functionality: . . .

Using this definition, the algorithm [4], generates a finite-automata based model, that also verifies the consistency of the system during construction. The input is given by a set of component descriptions (as in definition 2.2), and the algorithm works as follows: detect the source (initial component, $outports(C_{init}) = \emptyset$), execute all tasks, mark the component and establish the transitions corresponding to data flows, and push outport data into the stack; find a component such that its inports are available into the stack, and continue until all components are markes, or there is an interruption. The algorithm breaks if either a component C needs some data as inport that are not provided by any other component, or there exists a cycle (a mutual dependence between the inports and outports of two or more components).

There are several definitions for components, and we consider that all of them highlight the following basic characteristics:

- is an independent software module
- provides a functionality, but is not a complete system
- can be accessed only through its interface
- "black box" principle: that is, a component can be incorporated in a software system without regard to how it is implemented [1].

Based on this remark, we have focused on giving a more exact description to a component, especially on its interface. The interface of a component should specify all the services and ADT that the component provides to the rest of the system. The encapsulation principle has an important role in a component system. There are several levels of encapsulation and data hiding that should be taken into consideration:

- Code encapsulation: a service that is provided is specified through is signature, and the code is hidden inside the component. From outside one just know that a service performs a certain task, but not how it does it.
- Data encapsulation and hiding: again we adopt the view from OOP, where in a class the instance variables are hidden and can be accessed from outside the class only through methods. That's why, the interface specification will consist only class names that component is providing, and no variables;

Especially because of the interface role in a component, we have reviewed component definition.

Definition 2.3. A *component* will be specified by the following characteristics:
Component_id:
Interface:

A component may contain one or several class definition, services, modules, even code, in a structural programming way (procedures, functions) a.s.o. The interface of the component should therefor, describe all the exported features of the component. We will not detail here how this should be achieved. We just assume that the interface will contain these characteristics, and we will also assume that we have an "intelligent" searching routine, that knows exactly what we are looking for (service, function, procedure, method of a class), and return the corresponding result.

The services provided by the component should be seen as functions, so the interface will specify a list of function signatures. We will conclude by saying that a component interface has the general form:

$$Interface = [Domain, List\_of\_signatures]$$

where the *Domain* describe the classes, modules and other data structures definitions provided by the component.

We adopt the signature definition from OOP: a signature of an operation is provided as: *name : Domain* $\rightarrow$ *CoDomain*

3. Constructing the algorithm

Algorithm 1:
Input:

- A set of component descriptions (as in definition 2.3)
- A sequence of tasks to be executed; let it be: $[f_1, f_2, ..., f_n]$
- initial component name: C_{init}

Output: Model of the component-based system, if it can be built, otherwise corresponding messages.

```
Begin
  i:=1;
  Search if $f_i \in C_{init}.interface$
      if yes then
              begin
                  current := $C_{init}$;
                  mark($C_{init}$); add_state($C_{init}$);
                  push $f_i$ results in stack;
                  i:=i+1
              end
      else break; {$model\ cannot\ be\ constructed$; }
  for i := 2 to $n$ do
      Search if $f_i \in current.interface$
          if yes then
                  Search if $f_i.input \in stack$
                      if yes then
                              begin i:=i+1; push $f_i$ results in stack;
                                    add corresponding transition; end
                      else break {$no\ data\ provided\ to\ execute\ current\ task$}
          else
          Search if $f_i \in$ marked component interface
              if yes then
                  Search if $f_i.input \in stack$
                      if yes then
                          begin i:=i+1; push $f_i$ results in stack;
                              current := C; add corresponding transition; end
                      else break {$no\ data\ provided\ to\ execute\ current\ task$}
              else
              Search if $f_i \in$ unmarked component interface
                  if yes then
                      Search if $f_i.input \in stack$
                          if yes then
                              begin i:=i+1; push $f_i$ results in stack;
                                  mark(C); add_state(C);
                                  current := C; add corresponding transition; end
                          else break {$no\ data\ provided\ to\ execute\ current\ task$}
                  else break {$task\ not\ found$}
End.
```

The algorithm works in the following way: we start from component descriptions, given in the form specified by the definition 2.3, a sequence o tasks, that should be executed in the given order, without taking into consideration simultaneous execution. We also need to provide the initial component (or source), because we base our model on the fact that this component will provide services for input operations. Consequently, the first task in the list should perform some reading tasks.

During construction of the model, at each moment we will have a current component. We also use a marking routine in order to include only the components that are actually used (it's no use to construct a huge model, containing unseed components, even if their definitions are given). These two temporary information (current component and marked components) offer a way to optimize the construction: when a new task from the list is searched, instead of parsing the entire set of component definitions, the lookup method will search first in the current component; if it isn't found then the search will be performed in the marked components, and only after that in the rest of the definitions.

The tasks must the performed in the order specified by the sequence. The first task should belong to the initial component; if not then the construction is not possible. We consider that this is a logical supposition, and doesn't impose restrictions. If at one moment, a certain task is not found in any component definition, then the construction of the model will be stopped, with a corresponding message. The other possible interruption of the algorithm appears in case a certain task is found, but it's required input is not available (is not found in the stack), meaning that no previous task has this exact output. This is the reason why we have introduce this auxiliary stack: it will store all the intermediary results that are provided by tasks.

4. Comparison with the other model

There are several similarities and differences between the two models, and we will argue why we consider that this second approach is a more refined and exact method.

1) First at all, the first algorithm was though for constructing a correct system: when a component is inserted into the model all its functionality and data are available to the system; no restriction is imposed on which tasks are actually executed, and in what order. From this point of view, the second algorithm takes into consideration a concrete sequence of tasks to be executed, and assures that all these tasks are provided by the set of definitions. The resulting model, through the computed transitions, can generate work flows corresponding to these tasks.

We should also mention that this model can be used in extracting efficiency information about the components, and give an answer to the following issue stated by Szyperski: " Some of component's services may be less popular than others, but if none are popular and the particular combination of offered services is not either, the component has no market. In such a case, the overhead cost of casting a particular solution into a component form may not be justified“ [5].

2) Why interfaces instead of inports, outports, functionality? Certainly, there is a strong connection between the two approaches. But, interface-based description is more realistic, and the signatures of the services contain the information represented by inports and outports. In case we are working with interfaces, we can construct the description from definition 2.2 from the set of descriptions according to definition 2.3. In case a component comprises more than one functionality, the first model will not specify clearly which inports, respectively outports, are corresponding to a certain task (just a union of them). Thus, the second model description cannot be built from the first model component description, and we may conclude that the second model provides more information than the first one.

3) However, there is a small disadvantage of the second model: it does not provide a data flow. If we look at the model as a graph, then a path in this model will describe a control flow, but no information about data flow. On the other hand, the first model only provided data flow, and no information about the control (tasks to be executed, and in what order). So, depending on which aspect you want to concentrate, use one specification, or the other one, or both.

This model can be easily extended to include more detailed information: we can have a complete specification of the signatures from the interfaces, such that we can check in which condition the tasks can be executed, we can performed some type checking.

5. Example

Consider the following general set of components:
$C_1 = (\phi,\ \{d_1, d_2\},\ \{read\})$;
$C_2 = (\{d_1, d_3\},\ \{d_5, d_6\},\ \{task_1, task_2, task_3\})$;
$C_3 = (\{d_2\},\ \{d_3, d_7\},\ \{task_4\})$;
$C_4 = (\{d_5, d_7\},\ \{d_8\},\ \{task_5\})$;
$C_5 = (\{d_6, d_8\},\ \phi,\ \{write\})$;
$C_6 = (\{d_1, d_3\},\ \{d_4, d_5, d_6\},\ \{task_1, task_2, task_3\})$;
$C_7 = (\{d_1, d_5\},\ \{d_3\},\ \{task_1, task_2, task_3\})$;
$C_8 = (\{d_2, d_3\},\ \{d_4\},\ \{task_4\})$;
$C_9 = (\{d_4\},\ \{d_5, d_6\},\ \{task_5\})$;
$C_{10} = (\{d_6\},\ \phi,\ \{write\})$;
Applying the first algorithm [4], we have obtained the model from figure 1.

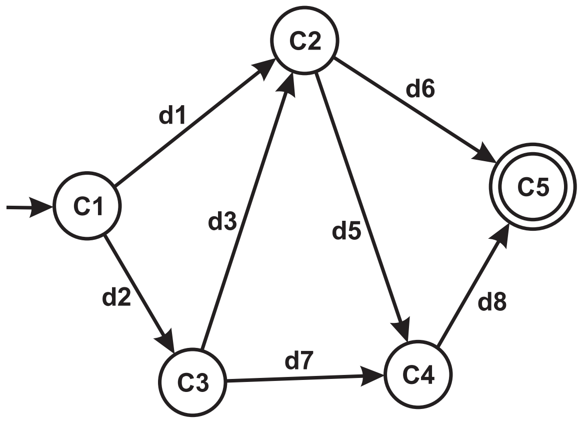

Figure 1. A valid model, based on definition 2.2.

Considering the same system, the system definition should be:
component1 = (C1, $[\phi, (read : \phi \to d_1, d_2)]$);
component2 = (C2, $[\phi, (task_1 : d_1 \to d_9, task_2 : d_9 \to d_5, task_3 : d_3, d_9 \to d_6)]$);
component3 = (C3, [class3,ϕ]);,
where $class3$ contains a method $task_4 : d_2 \to d_3, d_7$
component4 = (C4, $[\phi, (task_5 : d_5, d_7 \to d_8)]$);
component5 = (C5, $[\phi, (write : d_6, d_8 \to \phi)]$);

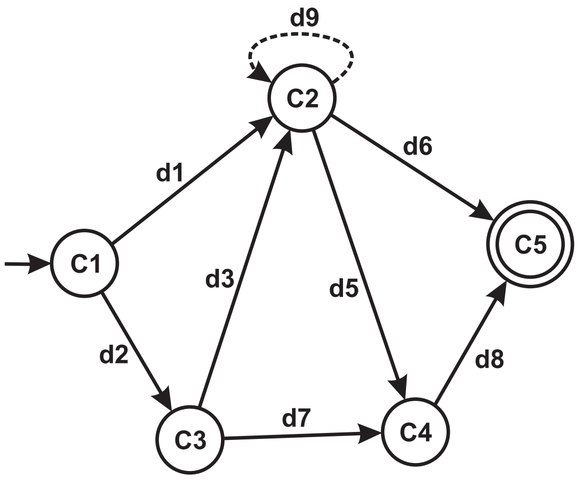

FIGURE 2. A valid model, based on definition 2.3. and algorithm 1

REFERENCES

[1] Cox, Philip T., Baoming, S., *A Formal Model for Component-Based Software*. In Proceedings of 2001 IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering, Stresa, Italy, September 2001
http://www.cs.dal.ca/ pcox/publications/Components-HCC01.pdf

[2] Henzinger, T. A., Masaccio - *A Formal Model for Embedded Components*, Proceedings of the First IFIP International Conference on Theoretical Computer Science, Lecture Notes in Computer Science 1872, Springer-Verlag, 2000, 549-563
http://sec.eecs.berkeley.edu/papers/00/masaccio/masaccio.pdf

[3] Motogna, S., Parv, B., *A Formal Model for Components*, Bul. Ştiinţ. Univ. Baia Mare, Ser. B, Matematica-Informatica, XVIII, **2** (2002), 269-274

[4] Parv, B., Motogna, S., Petraşcu, D., *Component System Checking Using Compositional Analysis*, Proc. ICCC 2004, Băile Felix, 27-29 Mai 2004

[5] Szyperski, C., Component Software, *Beyond Object-Oriented Programming*, ACM Press, Addison-Wesley, NJ, 1998

[6] Varr, D., *A Formal Semantics of UML Statecharts by Model Transition Systems*, to appear in Proceedings of ICGT 2002: International Conference on Graph Transformation, Barcelona, Spain, 2002
http://www.inf.mit.bme.hu/FTSRG/Publications/icgt2002_sc.pdf

BABEŞ BOLYAI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
STR KOGALNICEANU NO 1, 400084, CLUJ-NAPOCA, ROMANIA
E-mail address: {bparv,motogna,petrascu}@cs.ubbcluj.ro