Bul. Ştiinţ. Univ. Baia Mare, Ser. B,
Matematică-Informatică, Vol. XVIII(2002), Nr. 2, 269 - 274

A FORMAL MODEL FOR COMPONENTS

Simona MOTOGNA, Bazil PÂRV

Abstract. Component-based programming advantages have established it as an important paradigm. Developing large programs requires models in order to describe the system behaviour, offering a better understanding from analysis and design to programming and a tool for type specification. We propose a model to describe the behaviour of the components in a system and the relations between components. The system is specified as a finite automaton, where the states are the components and the transitions are the relations between components. Another important feature of the model is the fact that it provides two descriptions: an external (first level) and an internal (second level) point of view.

MSC: 68Q45, 68T10

Keywords:

1. Introduction

Due to the complexity and variety of current applications, software engineering has focused in the recent years on the design and development of component-based software development systems and methodologies. However, progressing from object oriented paradigm to component-based paradigm has risen a lot of issues concerning models and specifications of systems. There are many similar but not identical definitions of components, although the basic idea seems to be the same.

Definition 1.1: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.[3]

All definitions highlight the basic characteristics of a component:

- 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].

Various component-based software development systems are used nowadays, offering similar features, namely strong tools for developing large applications. An important disadvantage is that only very few are defined in clear and simple terms (as programming languages are), and their formal specification is difficult to be built. Several attemps have been made, each of them with their own shortcomings. Even, the proposed standard of

UML, could not be used in order to build the type system or to make any deductions regarding the correctness of model: "there is a huge abstraction gap between the "graphical" world of UML and all the mathematical models of describing dynamic semantics (such as transition systems, Petri nets, abstract state machines, process algebras, etc.). Consequently, systems engineers will require the backannotation of analysis results into the original UML formalism, as well as an easy–to–understand, visual specification of the dynamic semantics. [4]

A formal description of components would solve some of these issues, since it will offer the following features:

- provides a simple and precise description of a component–based system, including all types of possible components,
- provides a framework for building a type system and
- provides a framework for verifying the model.

The paper start with establishing the entities involved in a component definition, and describing the types of components that can appear in a component-based system. The domain and the set of attributes are defined such that a component interface can be specified by specializing these entities. We undertake a top-down approach in order to specify the relations between components. The first level model represents the initial description. Then, we will concentrate on the way compound components are created, depending on the interdependencies inside a compound component: serial, respectively parallel composition. This would result in a second level of specification. Both levels are described using finite automaton. Second level model will be defined in a constructive way, giving a straighforward solution to describe in detail each simple component of the system.

2 Definitions and notations

In order to specify the components we must first establish our entities involved in a component definition [1]:

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

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 $(X) \subset A$ - represents the set of attributes of the component X

$$inports(X) \cap outports(X) = \emptyset$$

Thus, we may describe the behaviour 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. Our specification is based on a top-down approach. We start by looking at a system that uses components, and the interaction between components in this type of system. First at all, the components that form the entities of this system are studied, and described. Then we take into consideration the actions that can occur in such a system.

Components

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:

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

- 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:

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

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

Remark 2.1: Why using compound components? There are several arguments concerning this problem. First at all, a compound component groups components that are consider to have a certain functionality within the global scope, and are the natural result of a top-down analysis of the system. A compound component can be seen as a subsystem, and also offers support for software reuse.

3 First Level Model

The system can be described at any moment by its state; this state depends on:
1. the components existing in the system; 2 the previous state; 3. the action that has been performed.

This remark has given the idea that a system involving components might be functioning like a push-down automaton, in which we have identified: the components behave as the states, and the transitions from one state to another are the result of some performaces of components.

Definition 3.1: A system of components will be defined as a finite automaton $M = (Q, \Sigma, q_0, T, F)$ where:

- Q is the set of states, each $q \in Q$ representing a component
- Σ, the input alphabet is, in fact, the set A of attributes

- q_0 is the initial state, represented by the source component[1]
- M is the transition function $T : Q \times \Sigma \to Q$
- $F \subseteq Q$ is the set of final states, represented by the destination components.

The transition function is defined in the following way: $T(q, a) = q'$, where $q, q' \in Q$, $a \in A$ and $a \in outports(q)$ and $a \in inports(q')$.

4 Second Level Model

We have described the system from an external point of view as a push-down automaton. We will preserve this description, but now from an internal point of view we possess more information about the components. We can think at this internal approach as if we have opened some black boxes containing simple and compound components. In order to extend our perception over an internal view on components we will reconsider and extend the definitions of simple and compound components.

Definition 4.1: *A simple component* X is defined as a 5-druple $X =$ (inports, outports, function, attributes, $\prec_X$), where:

- $inports(X) = \{a_1, a_2, ..., a_n\} \subseteq A, n \in N$; $outports(X) = \{b\} \subseteq A, b \notin inports(X)$;
- $function : Type(a_1) \times Type(a_2) \times ... \times Type(a_n) \to Type(b)$;
- $attributes(X) = inports(X) \cup outports(X) \cup var(X)$, where $var(X)$ represents the subset of attributes (different from the output) that can change their value due to some processing inside the component X.

$$var(X) = \{a | a \in A \setminus \{b\},\ a \text{ is variable}\}$$

- the binary relation $\prec_X \subseteq \mathbf{inports}(X) \cup \mathbf{outports}(X) \times \mathbf{outports}(X)$:[2]

• $a \prec_X b$ if the value b depends on the value a.

A compound component is a component that uses the same attributes in order to perform some operations on them. If we consider that a compound component is formed from 2 simple components then there are two basic ways in which these simple components can depend on each other:

- a parallel composition, denoted $A \| B$, in which the operations performed on data are independent and there is not dependency between outports(A) and outports(B);
- a serial composition, denoted $A + B$ in which the B component expects some results from component A.

Any compound component can be described with these two basic operations, no matter how many simple components it contains.

[1] We have consider that the system has only one source component. Of course, it is possible in real life applications to have more sources, but our approach does not loose generality, since these sources can be grouped together in a compound source

Definition 4.2: Parallel composition of the components A and B can be done if the following condition is satisfied:[2]

$$\forall x \in \mathbf{outports}(A) \vee y \subset \mathbf{outports}(B) :$$
$$\text{if } x \prec_B^* y \text{ then NOT } (y \prec_A^* x)$$
$$\text{and}$$
$$\text{if } y \prec_A^* x \text{ then NOT } (x \prec_B^* y)$$

and the result component is defined as follows:
- $inports(A \| B) = inports(A) \cup inports(B)$
- $outports(A \| B) = outports(A) \cup outports(B)$
- $\prec_{A\|B} = \prec_A \cup \prec_B$

Definition 4.3: Serial composition of the components A and B can be done if the following condition is satisfied: $outports(A) \cap inports(B) \neq \emptyset$ and the result component is defined as follows:
- $inports(A + B) = inports(A) \cup (inports(B) \setminus outports(A))$
- $outports(A + B) = (outports(A) \setminus inports(B)) \cup outports(B)$
- $\prec_{A+B} = \prec_A \cup \prec_B$

In these conditions, we expect that the transition function of the automaton to be a smoother description of the relation between components. Also, the states of the automaton carry significantly more information, but will represent only simple components, so we must redefine the automaton:

Definition 4.4: The finite automaton $M' = (Q', A, q_0, T', F')$, corresponding to a system with simple components is defined in a constructive way, from M:

- Q' is obtained from Q, where all the compound states q are replaced by two new states q_A and q_B;
- F' is obtained from F, where all the compound states q are replaced by two new states q_A and q_B;
- T' is obtained from T in the following way:

$$T'(q,a) = \begin{cases} p, & \text{if } T(q,a) = p,\ p,q \text{ are NOT compound components} \\ q_A, & \text{if } T(q,a) = p, p \text{ is } A \parallel B, a \in \mathbf{inports}(A) \\ q_B, & \text{if } T(q,a) = p, p \text{ is } A \parallel B, a \in \mathbf{inports}(B) \\ q_A, & \text{if } T(q,a) = p, p \text{ is } A + B, a \in \mathbf{inports}(A) \\ q_B, & \text{if } T(q,a) = p, p \text{ is } A + B, a \in \mathbf{inports}(B) \setminus \mathbf{outports}(A) \end{cases}$$

$$T'(q_A,a) = \begin{cases} p, & \text{if } T(q,a) = p,\ q \text{ is } A \parallel B, a \in \mathbf{outports}(A) \\ p, & \text{if } T(q,a) = p, q \text{ is } B \parallel A, a \in \mathbf{outports}(B) \\ p, & \text{if } T(q,a) = p, q \text{ is } A + B, a \in \mathbf{outports(A)} \setminus \mathbf{inports}(B) \\ p, & \text{if } T(q,a) = p, q \text{ is } B + A, a \in \mathbf{outports}(B) \\ q_B, & \text{if } \exists A + B, a \in \mathbf{outports}(A) \cap \mathbf{inports}(B) \end{cases}$$

[2] $\prec^*$ denoted the transitive closure of the relation $\prec$

The states that represented source component, destination components and simple components in the first level model are not affected by the transformation of the system. In case a state represented a compound component then the transition function from it and to it must be redefined in terms of its simple components. In addition, a transition inside a serial compound component is defined.

5 Conclusions and future work

The model we have presented is simple and offers a precise description of a component-based system, including all types of possible components. It is based on a top-down approach, decomposing the system into simple entities. It offers also the advantage that is easy to be extended, based on the following remarks:

- the domain D and the set of attributes A can be further specialized in order to specify the interfaces of components;
- the binary relation $\prec_X$ can impose conditions for extensions;
- other component composing methods can be studied within this framework.

This paper describe our first attemp to component specification, and will serve as basis for our further investigation. We intend to specialize this model, adding new composition operators, and to introduce a type system (provable as sound) for a component-based system.

REFERENCES

[1] Cox, Philip T, Baoming Song, 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] Thomas A. Henzinger, 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, pp. 549-563.
http://sec.eecs.berkeley.edu/papers/00/masaccio/masaccio.pdf

[3] Szyperski C., (1998). Component Software, Beyond Object-Oriented Programming, ACM Press, Addison-Wesley, NJ.

[4] Daniel Varró, 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

Received: 11.09.2002

Babeş-Bolyai University,
Faculty of Mathematics and Computer Science,
RO 3400 Cluj-Napoca, Str. Kogălniceanu 1, Romania
E-mail:bparv|motogna@cs.ubbcluj.ro