STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLV**, Number 2, 2000

FORMAL MODEL FOR SOFTWARE SYSTEMS COMPOSITION

IUGA MARIN

Abstract. In this paper we have provided a formal model for software systems specification and for the software systems composition operation. Using the notion of information system as a basis, we can model any information system using both software services and software interfaces. Doing this, we can develop a formal model for software systems composition. This formal model may be used both in formal specification of software systems (structure, functionality, requirements) and in software systems composition expressions.

1. An overview of software system notion

The history of "software system" notion is full of controversies and debates over what is central in the process of defining a software system. At first, a software system was identified with an executable program, but this definition has been enlarged later when a software system was associated with an executable program and its modules. Sooner, this definition has proven to be incomplete because the notion of software system has a larger range than that given by any program, no matters how large or complex this program is.

As a consequence, the definition of a software system has changed its center from the notion of executable programs and modules to the notion of software services and software systems inter-relations.

A radical change of perspective over the software systems is presented in [9]:

"Large software systems are non-algorithmic, open and distributed:

non-algorithmic: they model temporal evolution by systems of interacting components

open: they manage incremental change by local changes of accessible open interfaces

distributed: requirements as well as components are locally autonomous."

A system is generally considered to be a collection of components organized to fulfill a certain function or a certain set of functions. A software system is viewed as an entity that requests software services from the external environment and exports other software services to this environment. We will try to describe a software system without any need of information about its internal construction.

2000 *Mathematics Subject Classification.* 03B70,68N30.

1998 *CR Categories and Descriptors.* C.0. [**Computer Systems Organization**] : General.

It suffices to say that a software system has an internal state, represented by a set of abstract values, but we don't need to know how the state and the mechanism of state changing is implemented inside the system.

The classical software system concept is now replaced by the concept of *extensible system* (see [10]). An extensible system is considered to be a kind of software system whose functionality may be freely extended by replacing existing components with new ones. Smalltalk is an extensible language/system, and new additions to Java make it possible to create extensible systems in Java. Extensible systems cannot be created in more traditional languages such as Simula and C++. However, Active X from Microsoft, allows programming of extensible systems in C++, Visual Basic and other languages.

2. Modeling software systems using software services

We can observe now the fact that the definition of a software system is centered over the notion of software service, thus making the definition of software service the key to define the notion of software system. We will define the *software service* as a set of *operations* grouped under the same identifier. This identifier is the software service's identifier.

An *operation* is defined by a name, and a textual, rather informal, description of it.

Considering this, an *operation* could be represented as:

$$operation = (operation\_signature, operation\_description)$$

where:

operation_signature: is the operation's signature;
operation_description: is the operation's description.

We will provide a formal representation for an operation in this paper.

Once we can specify an operation, we are able to represent a *software service* as:

$$service = (service\_name, \{service\_operation_i, i \in 1, \ldots, num\_operations\})$$

where:

service_name: is the name of the software service,
service_operation$_i$: is the i-th operation of the software service
num_operations: is the number of operations associated with this software service.

A software service could be easily identified as a contract between a provider and a client. It specifies the terms of information exchange between the provider and the client, it specifies a protocol that makes the service provider and the client to understand each other and it specifies the conditions that must be met for the information exchange process.

As an example let's consider the process of a COM object serialization. The serialization is defined as "the ability of an object to write its state to a persistent storage" [6]. So, if we want a persistent COM object then this object must implement the service specified by *IPersistStorage* (at least). We will call this service as the *Persist_Storage* service, and it is characterized by the following operations:

IsDirty: indicates whether the object has changed since it was last saved to its current storage;

InitNew: initializes a new object, providing a handler to the storage to be used for the object;

Save: saves an object, and any nested objects that it contains, into storage;

SaveCompleted: notifies the object that it can revert from NoScribble or HandsOff mode, in which it must not write to its storage object, to Normal mode, in which it can;

HandsOffStorage: instructs the object to release all storage objects that have been passed to it by its container and to enter HandsOff mode, in which the object cannot do anything and the only operation that works is a close operation.

We can define a software system by the following quadruple:

$$(IN\_STATUS, OUT\_STATUS, IN, OUT)$$

where:

IN_STATUS: represents the system's internal status;

OUT_STATUS: represents the external environment's status that is accessed or modified by the system's services;

IN: represents the set of imported services that are needed by the system in order to fulfill its functionality;

OUT: represents the set of the exported services that are used by the software system to express its functionality.

As a synthesis of what we have exposed until now, we will consider a software system to be characterized by the following features:

- a series of software services exported to an external software environment;
- a series of software services imported from an external software environment;
- an internal state which could be changed as a result of a software service fulfillment;
- a software service execution could change the status of the external environment.

We are close to the model for a software component, introduced in [7], where the component is characterized by a service interface, a client interface and an implementation. Since the black-box model is adopted for a software component (excluding any information about internal implementation and imported services), we find the essence of this model applicable to software systems.

We denote by $OUT_1, \ldots, OUT_n$ the exported software services, where n is the number of exported services and we denote by $IN_1, \ldots, IN_m$ the imported software services where m is the number of the imported software services. Also we will denote by:

$$IN\_STATUS = \{IN\_STATE_1, \ldots, IN\_STATE_p\}$$

the set of the values of the software system status affected by the software services execution, and by:

$$OUT\_STATUS = \{OUT\_STATE\_1, \ldots, OUT\_STATE_q\}$$

the set of the values of the external software environment status affected by the software services execution.

We consider the external software environment to be divided into two parts, the first part denoted by IN exports software services to the software system, denoted by $SYSTEM$, and the second denoted by OUT is the part which imports the software services exported by $SYSTEM$. Both parts could be identified as a standalone software system. The first representation of the interaction of a software system with the external software environment, using software services, is given in Figure 1:

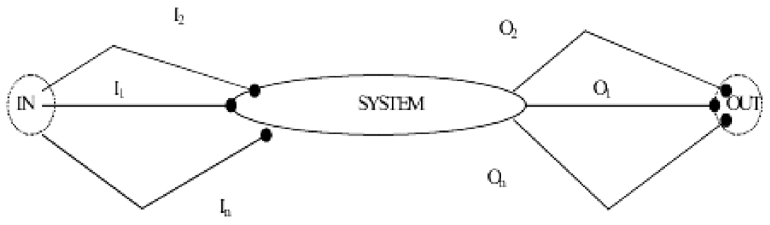

FIGURE 1. Representation of the interaction between a software system and its external software environment using software services

Let's consider, as an example, a software system, called *DataProcessor*, which receives data from an external data source, process it, and displays it to a display.

The imported services for this system are *DataProvider* service (imported form a data source system) and *DisplayRenderer* (imported from a graphical device system).

DataProvider service is characterized by the following operations:

OpenConnection: opens a connection with the data source;
CloseConnection: closes the connection with the data source;
GetData: obtains the raw input data.

DisplayRender service is characterized by the following operations:

ClearDevice: clears the content of the graphical device;
RenderImage: renders a graphical image.

The *DataProcessor* system exports the *DataProcessing* service, which is characterized by the following operations:

CheckValidity: checks the validity of input data;

ProcessData: processes the input data.

The internal status for the system is:

$$IN\_STATUS = \{idle, operation\_completed, operation\_ready\}$$

where:

idle: DataProcessor system is idling;
operation_completed: DataProcessor system has just finished an operation and is ready to provide output data;
operation_ready: DataProcessor system has received valid input data and is ready to begin a processing session.

The external status for the system's external environment is:

$$OUT\_STATUS = \{(connected), (not\_connected)\}$$

where:

connected: the data source has accepted connection and is ready to provide input data;
not_connected: the data source has not accepted a connection, the connection is closed or it is not ready to provide any input data.

As we may see from this example, the software service is only a feature that characterizes a software system and a software system could be viewed as a node that imports some services and exports other services.

3. Modeling software systems using software interfaces

By using a formal specification for a software service (as a interface implementation) we can obtain a formal representation for a software systems (as a set of interface implementations). In this kind of specification we must represent how the status of the external software environment and the status of the software system are affected by the software services execution.

The contract between a software entity and its external environment must be specified in a neutral language and there is needed a contract that will stipulate the terms and limits of the information transaction. In [5] we have specified the fact that the contract that supervises the information transaction should be based on the notion of software interface and the software interface must be specified in a programming language neutral manner.

However, other authors have different points of view about the neutrality of an element specification. They consider the specification of an element (type, interface, class, component, ...) as an abstract description of it, and a program (or module) as the concrete description of this element. In [2] it is requested that any software specification must be in an executable format, but it is hard to agree with this.

For a long time, a software service has been modeled as an interface. This kind of model ignores the fact that an interface can be identified only with the specification of a protocol for a set of operations (the syntactic part) and cannot capture the meaning of this operation (the semantic part). So, it is properly to discuss a service by the means of the implementation of an interface.

So, we will propose to use the interface implementations as a model for a software service, rather then using only interfaces. The interfaces are sets of method signatures and carry only the syntactic information, while the interface implementations are sets of methods and carry semantic information (behavioral specifications). There are many ways to specify a method by using predicate calculus, functional methods and non-functional methods.

We will propose here a specification model that is based on the predicate calculus. We will specify a method as:

$$(signature, precondition, postcondition)$$

where:

signature: is the method's signature;
precondition: is the method's precondition predicate;
postcondition: is the method's postcondition predicate.

The method's signature is represented as:

$$return\_typemethod\_name(in\_status, out\_status, [par\_rolepar\_name : par\_type])$$

where:

return_type: is the method's return type;
method_name: is the method's name;
in_status: represents the IN_STATUS for the software system to whom the method are bounded to, via its associated interface;
out_status: represents the OUT_STATUS for the software system to whom the method are bounded to, via its associated interface;
par_role: is the parameter's role (could be in, out, inout);
par_name: is the parameter's name;
par_type: is the parameter's type.

For a method m, we will consider the following sets:

- $IN(m) = \{$the set of all in or inout parameters$\} \cup \{$in_status, out_status$\}$;
- $OUT(m) = \{$the set of all parameters$\} \cup \{$in_status, out_status$\} \cup \{$result – the value returned by this method$\}$.

The precondition predicate is defined over values from $IN(m)$ and it is true if these values represents valid input data, and false otherwise.

The postcondition predicate connects the input data with the output data, and is true if the returned values are those expected (if valid input data is considered for the actual parameters of the method).

All that we have to remember is the fact that an interface implementation specification must consider the mechanism of state changing associated with the system that implements the interface. As a consequence of this thing, not all interface implementations could be attached to any software system. A software system that implements this interface must accept the values of the state changed by this interface implementation. We will denote by I_j the interface that has its implementation specified by the software service IN_j and with O_i the interface that has its implementation specified by the software service OUT_i. Using the name of the interface to designate the interface implementation associated with the service, we will have another representation of the interaction between a software system and its associated external environment, as can be seen in Figure 2.

The way an interface implementation is specified in has no critical importance. Thus, we have provided a functional specification, but it also can be non-functional (using message sending/receiving for example). This specification must take into consideration the modification of the state of the software system and its external environment.

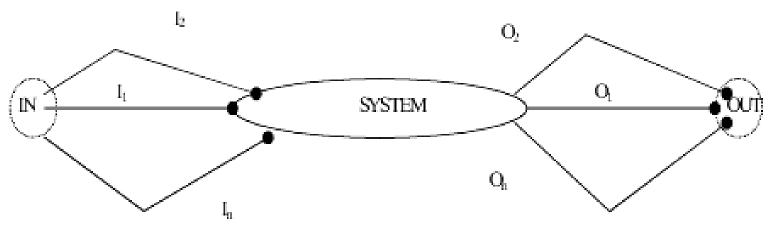

FIGURE 2. Representation of the interaction between a software system and its external software environment using interface implementation

Finally, we may synthetize the definition of a software system by using the following quadruple:

$$\begin{aligned} SYSTEM \; = \; & (IN\_STATUS, OUT\_STATUS, \\ & \{FI_k, 0 \leq k \leq ni, ni \in \mathbb{N}\}, \{FO_k, 0 \leq k \leq no, no \in \mathbb{N}\}) \end{aligned}$$

where the notions involved are:

SYSTEM: the software system to be defined;
IN_STATUS: the set of the values of the software system status affected by the software services execution;
OUT_STATUS: the set of the values of the external software environment status affected by the software services execution;
FI_l: the interface implementation associated with the software service IN_l;
FO_l: the interface implementation associated with the software service OUT_l;
ni: number of the imported services;
no: number of the exported services.

This quadruple can capture the entire description of a software service. It is an open model though, because of the way an interface implementation is specified in (but is not fixed because one can choose an alternate way to specify a software interface implementation). Any kind of specification (predicative, functional or non- functional) can be used, the only restriction is that the specification must consider the mechanism of status changing for a software system and its associated environment.

4. Software systems composition

The idea that a software system must be decomposed in smaller subsystems, for the purpose of a better handling, is an old idea and it is frequently argued in [1]. But building a software system form simpler subsystems is an idea embraced from the beginning of 90s, and the advantages of this method is presented in papers like [8, 4, 3]. We will specify a formal model, based on our software system specification, for the operation of software systems composition.

In the previous paragraphs we have provided a formal model for software systems, model based on services and interfaces. Using this model we will propose a formal model for the operation of composition of two software systems. In an informal manner, we will consider the result of the composition of two software systems S_1 and S_2 as a new software system that follow these rules:

- the group of IN services for the result system is obtained by putting together the IN services of both software systems. From this group we will eliminate all those services that are IN services for one system and OUT services for the other system;
- the group of OUT services for the result system is obtained by putting together the OUT services of both software systems. From this group we will eliminate all those services that are OUT services for one system and IN services for the other system;
- the IN_STATUS is the set of all values of the software system status which appear in all of the service descriptions from IN and OUT groups;
- the OUT_STATUS is the set of all values of the external environment status which appear in all of the service descriptions from IN and OUT groups.

For a software system S, we will consider the following functions:

- the $IN(S)$ function as the function that returns all the interface implementations associated with the imported services of this system;
- the $OUT(S)$ function as the function that returns all the interface implementations associated with the exported services of this system;
- the $SpecStatus_{IN}(spec)$ function as the function which returns all the system's status values which appear in the interface implementation from specification set *spec*;
- the $SpecStatus_{OUT}(spec)$ function as the function which returns all the external environment's status values which appear in the interface implementation from specification set *spec*.

By using the interface-based model, we can define the software systems composition by considering the set named $SYSTEMS$ as the set of all software systems.

The operation of composition, denoted by "+":

$$+ : SYSTEMS \times SYSTEMS \to SYSTEMS$$

will be defined for any software systems:

$$\begin{array}{ll} S_1 = (in\_status^1, out\_status^1, & \{(i_l^1, fi_l^1), 0 \le l \le ni^1, ni^1 \in \mathbb{N}\}, \\ & \{(o_l^1, fo_l^1), 0 \le l \le no^1, no^1 \in \mathbb{N}\}), \\ S_2 = (in\_status^2, out\_status^2, & \{(i_l^2, fi_l^2), 0 \le l \le ni^2, ni^2 \in \mathbb{N}\}, \\ & \{(o_l^2, fo_l^2), 0 \le l \le no^2, no^2 \in \mathbb{N}\}). \end{array}$$

as:

$$\begin{array}{lll} S_1 + S_2 = & (& SpecStatus((IN(S_1) \backslash S_2 \cap S_1) \cup (IN(S_2) \backslash S_1 \cap S_2)), \\ & & SpecStatus((OUT(S_1) \backslash S_1 \cap S_2) \cup (OUT(S_2) \backslash S_2 \cap S_1)), \\ & & (IN(S_1) \backslash S_2 \cap S_1) \cup (IN(S_2) \backslash S_1 \cap S_2), \\ & & (OUT(S_1) \backslash S_1 \cap S_2) \cup (OUT(S_2) \backslash S2 \cap S_1)) \end{array}$$

This formal definition of the software systems composition captures the entire meaning of the informal definition, previously presented. The expression:

$$(IN(S_1) \backslash OUT(S_2)) \cup (IN(S_2) \backslash OUT(S_1))$$

is the formal expression of the imported services, and the expression:

$$(OUT(S_1) \backslash IN(S_2)) \cup (OUT(S_2) \backslash IN(S_1))$$

is the formal expression of the exported services of the $(S1 + S2)$ information system.

The expressions:

$$\begin{array}{ll} SpecStatus_{IN} & ((IN(S_1) \backslash OUT(S_2)) \cup (IN(S_2) \backslash OUT(S_1)) \cup \\ & \cup (OUT(S_1) \backslash IN(S_2)) \cup (OUT(S_2) \backslash IN(S_1))) \\ SpecStatus_O UT & ((IN(S_1) \backslash OUT(S_2)) \cup (IN(S_2) \backslash OUT(S_1)) \cup \\ & \cup (OUT(S_1) \backslash IN(S_2)) \cup (OUT(S_2) \backslash IN(S_1))) \end{array}$$

defines the IN_STATUS and, respectively, OUT_STATUS attributes of the result system.

The composition operation for two software systems models the process of the tight coupling between these systems. All the similar services exported by one system and imported by the other system are hidden in the obtained system, along with the corresponding status values. One can use this operator if he wishes to obtain an expression for a tight interaction between two software systems. The composition operation is characterized by the following proprieties:

- the composition operation is commutative;
- the system $\theta = (\emptyset, \emptyset, \emptyset, \emptyset)$ is the neutral element for the composition operation;
- if we consider the software system:

$$\begin{array}{lll} S & = & (IN\_STATUS, OUT\_STATUS, \\ & & \{I_k, 0 \le k \le ni, ni \in \mathbb{N}\}, \{O_k, 0 \le k \le no, no \in \mathbb{N}\}) \end{array}$$

then the following system:

$$\begin{aligned} CLOSE(S) &= (OUT\_STATUS, IN\_STATUS, \\ &\quad \{O_k, 0 \leq k \leq no, no \in \mathbb{N}\}, \{I_k, 0 \leq k \leq ni, ni \in \mathbb{N}\}) \end{aligned}$$

is the inverse element of S for the composition operation;

- the composition operation is not generally associative.

The proof of these proprieties, due to its extent, it is not discussed here. We have only wished to enumerate them.

The software systems specification and composition may be used for many purposes, ranging from checking of software systems compatibility to methods for software applications design and generation. CASE tools can use them as a support for software systems representation and interaction models. They might also be the basis for other different formal models in programming.

References

[1] Dahl O.J., Dijkstra E. W., Hoare C.A.R., Structured Programming, Academic Press, 1972

[2] Fucs N. E., "Specifications Are (Preferably) Executable", Software Engineering Journal, September, 1992

[3] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

[4] Hölzle U., Integrating Independently-Developed Components in Object-Oriented Languages in LNCS 707, pp. 36–56, 1993

[5] Iuga Marin, A Graphical Representation for Software Component Systems, Faculty of mathematics and Computer Science, Research Seminars, pp. 107–110, 1999

[6] MSDN Library Visual Studio 6.0, Visual C++ Programmers guide, Serialization (Object Persistence)

[7] Allen Parrish, Component Based Software Engineering: A Broad Based Model is Needed, Brandon Dixon, David Hale in International Workshop on Component-Based Software Engineering proceedings, pp. 43–46, 1999

[8] Jan Udell, ComponentWare, Byte Magazine, pp. 46–56, 1994

[9] Wegner Peter, Models and Paradigms of Interaction, in Object-Based Distributed Programming, ECOOP'93 Workshop, Vol. 791, pp. 1–32, Springer-Verlag, 1994

[10] Szyperski Clemens, Pountain Dick, Extensible Software Systems, in BYTE May 1994, pp. 57–62, 1994

Babeş-Bolyai University, Faculty of Mathematics and Computer Science
E-mail address: marin@cs.ubbcluj.ro, iuga_marin@yahoo.com