STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LII**, Number 2, 2007

# COMDEVALCO — A FRAMEWORK FOR SOFTWARE COMPONENT DEFINITION, VALIDATION, AND COMPOSITION

BAZIL PÂRV, SIMONA MOTOGNA, IOAN LAZĂR, ISTVAN CZIBULA, AND LUCIAN LAZĂR

Abstract. This paper introduces **ComDeValCo** - a framework for Software **Com**ponent **De**finition, **Val**idation, and **Co**mposition. This is the first paper in a series describing current and further developments of this framework, which includes a modeling language, a component repository and a set of tools. The object-oriented modeling language contains fine-grained constructions, aimed to give a precise description of software components. Component repository is storing valid components, ready to be composed in order to build more complex components or systems. The toolset contains tools dedicated to component definition, validation, and composition, as well as the management of component repository.

## 1. INTRODUCTION

Software systems become more and more complex. In most situations, where the complexity of the problem to be solved is an important issue, the decomposition is used - the initial problem is splitted into small sub-problems, then each sub-problem is solved independently (or their solutions are identified), and finally the target system is built by composing the solutions of sub-problems. The evolution of software systems development include the use of several paradigms: procedural, modular, object-based and oriented, and component-based; most authors consider component-based development as the paradigm of the third millenium. The drivers of this evolution were at least the following:

- the increased complexity of the problems to be solved (and consequently of the systems to be built);

Received by the editors: November 10, 2007.

2000 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification – *Formal methods, Model checking, Validation*; D.2.13 [**SOFTWARE ENGINEERING**]: Reusable Software – *Reuse models*; I.6.5 [**SIMULATION AND MODELING**]: Model Development – *Modeling methodologies* .

- the need for performance (w.r.t. time, money, and throughput): producing new state-of-the-art systems in short time, and with less money.

One of the drivers of this evolution was *reuse*. Early forms of software reuse are collectively known as *code reuse*. Nowadays, software reuse covers also *design reuse*. Successful design fragments are collected into catalog form and collectively known as *design patterns*; they represent not complete designs, but partial solutions, ready to be reused into new designs or contexts. Also, class libraries evolved into *frameworks*, which represent complete system architectures. They are an incarnation of *inversion of control* design principle, being a set of cooperating classes that make up a reusable design from a specific application domain.

The paper is organized as follows: after this introductory section, the second one is discussing component-based development process, and the current status of research and industry efforts in the field. Third section presents the proposed solution, **ComDeValCo** framework, detailing its components: modeling language, component repository and the toolset. The last section contains some conclusions and plans further efforts.

## 2. THE PROBLEM

2.1. **The problem: component-based software development.** The process of component-based software development (or CBD for short) has two sub-processes more or less independent: component development process and system development process. Naturally, the requirements on components are derived from system requirements; the absence of a relationship, such as causal, may produce severe difficulties in both sub-processes mentioned above.

The system construction by assembling software components [CL02] has several steps: component specification, component evaluation, component testing, and component integration. The system development sub-process focuses on identifying reusable entities and selecting the components fulfilling the requirements, while in the component development sub-process the emphasis is on component reuse: from the beginning, components are designed as reusable entities. Component's degree of reuse depends on its generality, while the easiness in identification, understanding, and use is affected by the component specification. The sole communication channel with the environment is the component's interface(s). In other words, the client components of a component can only rely on the contracts specified in the interfaces implemented by the component. Thus, it is obvious that component development must be interface-driven. One of major CBD challenges is to design appropriate interfaces.

In our opinion, the main CBD challenge is to provide a general, flexible and extensible model, for both components and software systems. This model should be language-independent, as well as programming-paradigm independent, allowing the reuse at design level.

2.2. **CBD design process models.** The design process of a component-based system [HW00] follows the same steps as in the classical methods: the design of architecture, which depicts the structure of the system (which are its parts) and the design of behavior (how these parts interact in order to fulfill the requirements). The structural description establishes component interconnections, while behavioral description states the ways in which each component uses the services provided by interconnected components in order to fulfill its tasks.

The main idea of CBD is to build the target system from existing components; this has several consequences on the target system's life-cycle. First, the system development sub-process [HW00] is separated from the component development sub-process. Second, a new sub-process arises: component identification and evaluation. Third, the activities in both sub-processes differ from the traditional methods: the focus is on component identification and verification (for system development), and on component reuse (in the case of component development).

The paper [CL02] describes a software systems development model which can be used in component-based development. The classical waterfall life-cycle model was upgraded such that it contains component-centric activities: requirements analysis and design - specific to the waterfall model - are combined with component identification and selection - specific to the component-based development. The design stage includes architectural design and activities related to component identification, selection, and adaptation.

Another viewpoint, given in [WR02], considers the following steps in building a software system from components: a) connecting the components such that they match; b) understanding the interconnections between components, and c) examining the behavior of the whole target system with respect to the requirements.

2.3. **Component and system models.** There are many ways to deal with component-based software development. The simplest one is to add to the contract (interface) of the component all requirements w.r.t. its use (the meaningful interconnections to other components). Unfortunately, this supplement to the component specification is a time and effort-consuming activity. For every newly-created component, one must identify all compatible components, and after that the contracts of these components must be updated in order to include this new component.

An alternative solution, given in [WR02], is to think the behavior of the target system as a separate activity, which is performed without accessing the target system under construction. Unfortunately, this activity is very complex, but there are models which can help. These models do not reproduce the behavior of the whole system; they will cover only particular aspects of the system which are of interest at a specific time. This approach neglects insignificant details, thus reducing the complexity of the resulting model, total effort and time for building the model.

Building and testing a real target system is more difficult and takes a greater volume of resources than the corresponding model evaluation. This is because the models do not address complexity of the system and the subtleties of its environment. Other potential benefit of using models is their controllability, i.e. their evaluation before the target system is designed. In this case, the models are analyzed, simulated, and evaluated using software tools. The system developer builds a model which is taken by the evaluation tool. The comparison of the model behavior with respect to the system requirements is made by using either the modeling language or some other specialized languages.

In order to be evaluated, the models need to be precise, complete, and consistent. Generally speaking, if the degree of model (i.e. modeling language) formality is low, the model is a good candidate for inconsistencies, because some modeling constructs do not have a unique interpretation. When the model is used to assist the process of designing interactions between different components / parts of a system, or to assess the correctness of the system, preciseness, completeness and consistency are a must.

The success of using models (formal or not) is influenced in part by the availability and the degree of acceptance of modeling tools and techniques developed by the software development community. Those who build models need to perceive the usefulness of the models [HW99], need to find a tradeoff between model complexity and its ease of use. It is convenient to build simple models, without great investments in time and intellectual effort. More important, the resulting models need to be accessible, easy to understand and analyze, and to have a reasonable degree of formality.

It is recognized that modeling is not used today in the software development process at its full strength. Usually, models are only simple design notes, thrown away after the coding is completed. However, model-based approach (or model-driven approach in software development, MDA) gains more adepts. In MDA, the model of the system is the center of software development process. At least four modeling notations are currently used: finite state machines ([EG03], [LL00], [WR00]), statecharts ([GM95]), Petri nets ([AW], [PJ02]) and role-activity diagrams ([HW00], [HHW01], [HWC04], [WR02]).

An important direction concerning the use of models in the CBD process is represented by the Object Management Group consortium (OMG) efforts, known collectively as executable UML. For example, [OMG05a] describes the semantics of some simple UML constructions, intended to be used in model validation and simulation. Model-level testing and debugging is covered in more detail by [OMG05b], while [OMG05c] contains specifications belonging to different application domains. Also, [OMG], which refers to current OMG technology adoption processes, contains more references regarding model validation and simulation.

Business process modeling comes from another perspective, but has the same final goal as our problem. For example, [Liu04] uses abstract logic trees to represent UML activity diagrams, allowing the study of their properties using graph algorithms. Also, [Hol04] and [Gru07] compare structured programming primitives with UML activity diagrams, studying graphical ways of representing structured processes.

Another industry initiative related to model specification, validation, and simulation is AGEDIS - Automated Generation and Execution of Test Suites for DIstributed Component-based Software [AGEDIS], a project ended in 2004.

## 3. THE SOLUTION: ComDeValCo

The proposed solution is **ComDeValCo** - a conceptual framework for Software **Com**ponents **De**finition, **Val**idation, and **Co**mposition. Its constituents are meant to cover both sub-processes discussed in 2.1: component development and component-based system development. This paper should be seen as a presentation of a solution for these two interconnected sub-processes and as a plan for the further developments of the framework.

The sub-process of component development starts with its definition, using an object-oriented modeling language, and graphical tools. The modeling language provides the necessary precision and consistency, and the use of graphical tools simplifies developer's work, which doesn't need to know the notations of modeling language. Once defined, component models are passed to a V & V (verification and validation) process, which is indended to check their correctness and to evaluate their performances. When a component passes V & V step, it is stored in a component repository, for later (re)use.

The sub-process of component-based system development takes the components already stored in repository and uses graphical tools, intended to: select components fulfilling a specific requirement, perform consistency checks regarding component assembly and include a component in the already existing architecture of the target system. When the assembly process is completed, and the target system is built, other tools will perform V & V, as well as performance evaluation operations on it.

Constituents of the conceptual framework are: the modeling language, the component repository and the toolset. Any model of a software component is described by means of a modeling language, programming language-independent, in which all modeling elements are objects. The component repository represents the persistent part of the framework and its goal is to store and retrieve valid component models. The toolset is aimed to help developers to define, check, and validate software components and systems, as well as to provide maintenance operations for the component repository.

The rest of this section gives a short description of the above constituents, illustrating their current status and intended further developments. More detalied descriptions will be given in separate papers.

3.1. **Modeling language.** *The software component model* is described by an object-oriented modeling language, all modeling elements being objects. The modeling language is independent from any object-oriented programming language and has the following features:

- all language elements (constructs) are objects, instances of classes defined at logical level, with no relationship to a concrete object-oriented programming language;
- language constructs cover both categories of software component discussed - the target software system - `Program` (the only executable) and proper software components (not executable by themselves, but ready to be assembled into a software system) - `Procedure, Function, Module, Class, Interface, Connector, Component;`
- there is a 1:1 relationship between the internal representation of the component model - seen as aggregated object - and its external representation on a persistent media, using various formats: XML, object serialization, etc.

A software component is fully defined, i.e. its model contains both component specification and component implementation. For example, `Program` components have three main constituents: the name, the state and the body. `Procedure` components, which are a specialization of `Program,`have as specific constituents their in, out, and in-out parameters (seen as lists). Component state contains all declared variables (names and values), while its body is a `CompoundStatement` (modeling construct defined using the *Composite* design pattern). Figure 1 is a UML class diagram showing some of the modeling elements already in place and their relationships. These elements constitute a UML metamodel, and can be used to build new UML profiles, in order to use existing CASE tools to build component models.

Statement subclasses are `SimpleStatement` and `CompoundStatement,`with `SimpleStatement` subclasses covering all control statements in an imperative

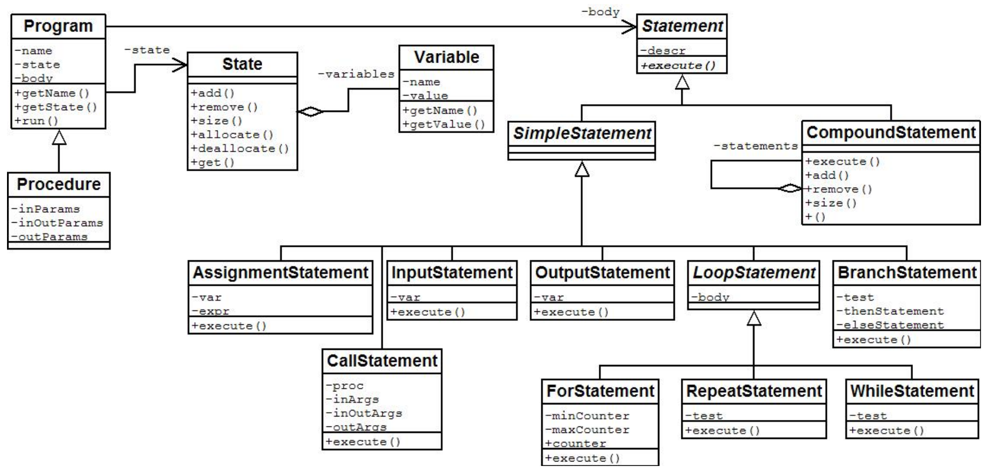

FIGURE 1. Class diagram (procedural paradigm, proof of concept)

programming language: `AssignmentStatement, CallStatement, InputStatement, OutputStatement, LoopStatement,`and `BranchStatement.`

The current version of the modeling language contains constructs belonging to the procedural paradigm, and is described in more detail in [Parv08].

3.2. **Component repository.** *Component repository* represents the persistent part of the framework, containing the models of all full validated components. Its development include the design of its data model, establishing indexing and searching criteria, as well as the format of representation. The ways of describing, indexing and searching considered will exploit XML-based protocols used to describe and discover Web services (WSDL and UDDI).

3.3. **The toolset.** *The toolset* is intended to automate many tasks and to assist developers in performing component definition and V & V tasks, maintenance of component repository, and component assembly. The tools are:

- DEFCOMP - component definition;
- VALCOMP - component V & V;
- REPCOMP - component repository management;
- DEFSYS, VALSYS - software system definition by component assembly, respectively V & V;
- SIMCOMP, SIMSYS - component and software system simulation;
- GENEXE - automatic generation of executable software systems.

From another perspective, the toolset will include some existing CASE tools, covering in part or in whole some of the functions above.

3.4. **Features of the proposed solution.** The proposed solution brings original elements in at least the following directions:

- the object model is precise and fine-grained, because all objects are rigorously defined, and the component behavior is described at statement level. The UML metamodel has no correspondent for modeling constructs more fine-grained than `Program, Procedure` and `Function;`
- the models are executable, verifiable, and evaluable because each component can be executed; moreover, one can use tools for checking validity and evaluating complexity;
- the models are independent of a specific (object-oriented) programming language and programming paradigm;
- modeling language is flexible and extensible; the dimensions of extensibility are: statement set, component definition, data type definition, and the component family;
- the statement set is extensible, by simply considering new (possible) primitive statements; as Figure 1 suggests, inheritance and composition are the main code reuse mechanisms used to define new statements;
- the component definition is extensible (we started with the simplest implementation of the component, using simple data types and expressions; next steps will include component specification, which needs more elaborate data types and expressions);
- data type definitions are also extensible (we started with simple data types; next steps will add structured types to the model, then object types - `Class`and `Interface`);
- the component family is extensible (we started with procedural paradigm components - `Program , Procedure` and `Function;`next steps will add: modular components - `Module -` object-oriented ones - `Class` and `Interface,`and, finally - `Component);`
- modeling language allows automatic code generation for components in a concrete programming language, according to Model Driven Architecture (MDA) specifications. One can define mappings from the modeling elements to specific constructs in a concrete programming language in a declarative way.

## 4. CONCLUSIONS AND FURTHER WORK

From methodological viewpoint, the main issue is to completely model all theoretical aspects in concrete objects - elements of modeling language. The modeling process is a gradual one, in order to keep its complexity under control. The main principle to be followed is to perform small steps; a step means here either implementing a new concept (transforming the concept into an object), or extending either a model element, a tool, or component repository (by adding new features). One starts with simple objects and check after each step that things work.

Each modeling step include both theoretical/analytical activities - the abstract model of the concept - and practical/applicative ones - coding, testing and integrating it in the framework.

The intended use of the conceptual framework covers research, education, and industry applications. The competitive advantages are as follows:

- full compliance to the principles and methods of component-based software development, by covering both sub-processes - component and system development;
- high level of abstraction, assured by the independence of a specific programming language;
- ease of use: the model complexity is hidden behind a set of diagrams (model views), easy to define, understand, and manipulate;
- focus on reuse: the framework favors the definition and use of reusable software components by its constituent: component repository.

Other developments will include: maintenance of component repository by including new components, publishing the access interface to the component repository as a Web service, modeling at higher levels of abstraction, like workflows, business processes, application domain frameworks.

## 5. ACKNOWLEDGEMENTS

This work was supported by the grant ID_546, sponsored by NURC - Romanian National University Research Council (CNCSIS).

## 6. REFERENCES

[AGEDIS] *AGEDIS, Automated Generation and Execution of Test Suites for DIstributed Component-based Software*, http://www.agedis.de/index.shtml/.

[AW] W.van der Aalst, *PetriNets, tutorial* http://is.tm.tue.nl/staff/wvdaalst/petri_nets.htm.

[CL02] Crnkovic, I., Larsson, M., *Building Reliable Component-Based Software Sistems*, Prentice Hall International, Artech House Publishers, ISBN 1-58053-327-2, Available July 2002. http://www.idt.mdh.se/cbse-book/

[EG03] Eleftherakis, G., *Formal Verification of X-machine Models: Towards Formal Development of Computer-Based Sistems*, PhD, 2003.

[GM95] Glinz, M., *An Integrated Formal Model of Scenarios Based on Statecharts.* In Schfer, W. and Botella, P. (eds.): Software Engineering - ESEC'95. Berlin: Springer, 254-271.

[Gru07] Gruhn, V., Laue, R., *What business process modelers can learn from programmers*, Science of Computer Programming, 16 (2007), No. 1, 4-13.

[Hol04] Holl A., Valentin G., *Structured Business Process Modeling (SBPM)*, Information Systems Research in Scandinavia (IRIS 27), 2004.

[HHW01] Henderson, P., Howard Y., Walters, R.J., *A tool for evaluation of the Software Development Process*, Journal of Systems and Software, Vol 59, No 3, pp 355-362 (2001).

[HW99] Henderson, P., Walters, R.J., *System Design Validation Using Formal Models*, 10th IEEE International Workshop in Rapid System Prototyping, June 99, Clearwater, USA.

[HW00] Henderson, P., Walters, R.J., *Behavioural Analysis of Component-Based Sistems*, Declarative Sistems and Software Engineering Research Group, Department of Electronics and Computer Science, University of Southampton, Southampton, UK, 06 June 2000.

[HWC04] Henderson, P., Walters, R.J., Crouch, S., *Implementing Hierarchical Features in a Graphically Based Formal Modelling Language*, 28th Annual International Computer Software and Applications Conference (COMPSAC 2004), Hong Kong, 2004.

[Liu04] Ying Liu et al., *Business Process Modeling in Abstract Logic Tree*, IBM Research Report RC23444 (C0411-006) November 19, 2004.

[LL00] Jie Liu, Edward A. L., *Component-Based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics*, Proc. of the 2000 IEEE International Symposium on Computer-Aided Control System Design Anchorage, Alaska, USA, September 25-27, 2000, pp 95-100.

[OMG] OMG, *Current OMG Technology Adoption Processes Under Way.* Pending Requests for Proposals, http://www.omg.org/public_schedule/.

[OMG05a] OMG, *Semantics of a Foundational Subset for Executable UML Models RFP*, http://www.omg.org/cgi-bin/doc?ad/2005-4-2/.

[OMG05b] OMG, *Model-level Testing and Debugging*, http://www.omg.org/cgi-bin/doc?ptc/2007-05-14/.

[OMG05c] OMG, *Catalog of OMG Domain Specifications*, http://www.omg.org/cgi-bin/doc?ad/2005-4-2/.

[Parv08] Pârv, B., Lazăr, I., Motogna, S., *ComDeValCo framework - the modeling language for procedural paradigm*, to be published in International Journal of Computers, Communications, and Control (IJCCC), vol. III, 2008.

[PJ02] Padberg, J., *Petri Net Modules*, Journal on Integrated Design and Process Technology vol. 6(4), pp. 121-137, 2002.

[WR02] Walters R. J., *A Graphically based language for constructing, executing and analysing models of software sistems*, PhD, 2002.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 1, M. Kogălniceanu, Cluj-Napoca 400084, Romania

*E-mail address*: `bparv,motogna,ilazar,czibula@cs.ubbcluj.ro`