CREAT. MATH. INFORM.
21 (2012), No. 1, 95 - 102

Online version at `http://creative-mathematics.ubm.ro/`
Print Edition: ISSN 1584 - 286X Online Edition: ISSN 1843 - 441X

An approach to MDA - ComDeValCo framework

S. MOTOGNA, I. LAZĂR and B. PÂRV

ABSTRACT. Software design based on Model Driven Architecture can be essentially improved including agile principles as immediate execution and test first development. This paper shows how a concrete tool, the ComDeValCo framework, has been constructed and enhanced to support such an approach. The paper discusses in detail how the constructs for component models and for the dynamic execution environment have been introduced.

1. BACKGROUND

Component-based approaches are used nowadays in developing large and complex software applications, and the process is becoming more complicated and difficult to predict. In consequence, there are several approaches related to the simplification of the construction of components. One solution is to separate the business logic of a component from the non-functional requirements related to the container in which the component execution will be managed. In such a context, developers concentrate on implementing the business logic of the component, then they configure declaratively the deployment of components.

Another important aspect of component models and frameworks refers to the development approach. Approaches in which modeling is at the core of the development activities also simplify the component construction process [1]. One of the main component based development's 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. Well-known such approaches are based on UML and MDA.

MDA framework [9] provides an approach for specifying systems independently of a particular platform and for transforming the system specification into one for a particular platform. The most important benefits are higher abstraction level in program specification and increase of automation in program development. The availability of such tools and the easiness of their use has contributed to the success of MDA. But development processes based on MDA are considered heavy-weight processes since they cannot deliver (incrementally) partial implementations to be executed as soon as possible.

In this context, executing UML models became a necessity for development processes based on extensive modeling. For such processes, models must act just like code, and UML 2 and its Action Semantics [11] provide a foundation to construct executable models. In order to make a model executable, it must contain a complete and precise behavior description. Unfortunately, creating such a model is a tedious task or an impossible one because of many UML semantic variation points. Executable UML [5] means an execution

Received: 31.10.2011. In revised form: 30.06.2011. Accepted: 30.11.2011
2010 *Mathematics Subject Classification.* 68N30, 68P99.
Key words and phrases. *Repository, executable models.*

semantics for a subset of actions sufficient for computational completeness. Two basic elements are required for such subsets: an action language and an operational semantics. The action language specifies the elements that can be used while the operational semantics establishes how the elements can be placed in a model, and how the model can be interpreted. Again, creating reasonable sized executable UML models is difficult, because the UML primitives from the UML Action Semantics package are too low level.

The Executable Foundational UML (fUML [12]) is a computationally complete and compact subset of UML, designed to simplify the creation of executable UML models. The semantics of UML operations can be specified as programs written in fUML. The fUML standard provides a simplified subset of UML Action Semantics package (abstract syntax) for creating executable UML models. It also simplifies the context to which the actions need to apply. For instance, the structure of the model will consist of packages, classes, properties, operations and associations, while the interfaces and association classes are not included. However, the description uses low-level UML primitive making the process of creating reasonable sized executable UML models difficult.

Applying the Agile development [4] principles (test first, immediate execution) may represent a solution for the current status, as described in the next sections.

Our solution, ComDeValCo [14] - a conceptual framework for Software Component Definition, Validation, and Composition had started by proposing a way of defining executable models that could be turned in software components. After such components are defined, they are ready to be used at a design level. Such definitions should be stored in some way in order to be ready for reuse. The component repository will represent the persistent part of the framework, containing the models of all fully validated components.

The rest of the paper is organized as follows: Section 2 presents the ComDeValCo framework and its fundamental constituents. The next two sections concetrate on the evolvement of our framework: defining the object model, as the initial configuration and then extending this model to include features related to modules and execution environment. Section 5 shows how an MDA approach centered on this model can be developed, and then the last section draws some conclusions and discusses related work.

2. ComDeValCo framework

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 stores and retrieves 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.

2.1. Modeling Language.

The modeling language was designed in order to provide both graphical and textual notations for easy manipulation of language elements. It has to be simple and easy to handle for both the developers and users of the model. The basic features are:

- all elements are objects, with no relationship to a concrete programming language;
- covers both complete software systems *Program* and concrete software components (*Procedure, Function, Module, Class, Interface, Connector, Component*);
- it allows automatic code generation towards concrete programming languages.

2.2. **Component Repository.** The Component repository represents the persistent part of the framework, containing the models of all full validated components. Its development include separate steps for designing the data model, establishing indexing and searching criteria, and choosing the representation format.

The main functionalities consist in storing and retrieving software components and systems. The storage uses OASIS RIM [8] as a component representation format. The components inside the repository are classified according to one or multiple keywords, for a faster search.

2.3. **ComDeValCo toolset.** The toolset is intended to automate many tasks and to assist developers in performing component definition, validation and verification, maintenance of component repository, and component assembly. The tools initially considered were the following:

- 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.

First version of DEFCOMP was an Eclipse plug-in, covering model construction, execution, and testing, thus having VALCOMP functionality also. Program units can be expressed in both graphical or textual ways. The two different editing perspectives of DEFCOMP are synchronized, acting on the same model.

VALCOMP was designed with the agile test-driven development process in mind, allowing developers to build, execute, and test applications in an incremental way, in short development cycles.

DEFSYS and VALSYS were initially considered as tools for developing, verifying and validating software systems by assembling components taken from component repositories. Later on, by adopting a test-driven development method, these two sub-processes (component definition and system definition) were considered as a whole, and DEFCOMP and VALCOMP tools address all needed functionality. This way, the functionality of ComDeValCo workbench covers both component/software system development/ verification and validation activities.

3. Initial Object Model

The initial object model considered for the ComDeValCo framework was structured on three layers, in a top-down perspective, as represented in Figure 1: (1) program units, (2) execution control constructs (statements) and (3) low-level (syntactical) constructs.

Program units considered so far were *Program*, *Procedure* and *Function*. They belong to the upper layer of the modeling language. *Program* is the only executable, as sugested by the operation *run* from its specification. *Procedure* and *Function* represent concrete software components. A procedure declaration states its name, formal parameters, local state, and body. *Procedure* class inherits naturally from *Program* class; additionally, separate lists for in, in-out and out parameters are needed for a complete implementation of CallStatement.execute() method. We have taken into consideration user-defined functions with no side-effects, the only goal of their execution being the return of a value; in other words,

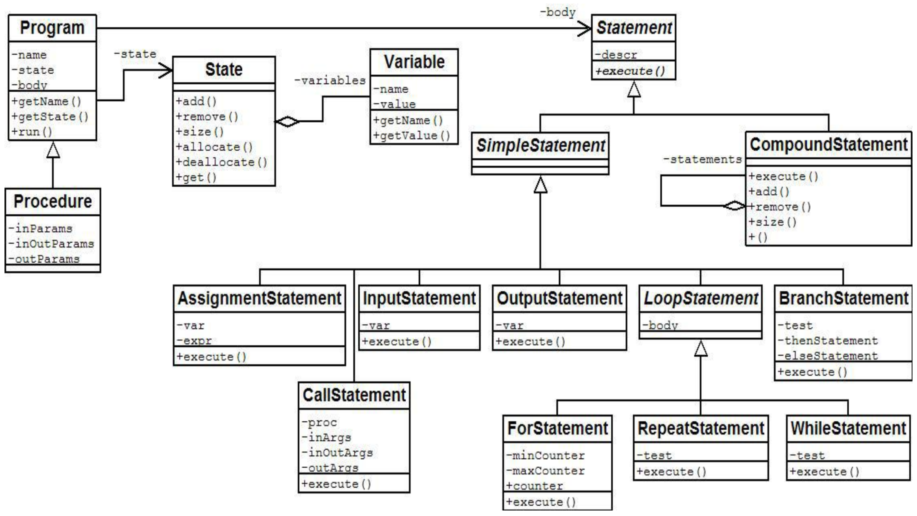

FIGURE 1. Initial object model

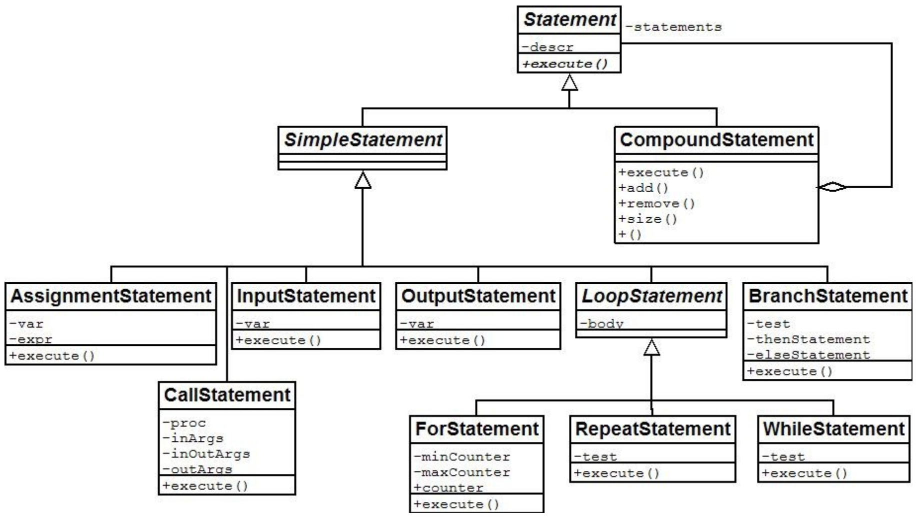

FIGURE 2. *Statement* class hierarchy

the state of the caller is not changed by the function call. Consequently, the *Function* class has just a list of in parameters and returns a Value object.

The middle layer contains objects which model the execution control, all of them inheriting from a base class *Statement*. They correspond to traditional statements from an imperative programming language, as shown in Figure 2.

The lowest layer contains basic constructs of the modeling language, classes *Type*, *Declaration*, *Value*, *Variable* and *Expression*. The data type concept is abstractized by *Type* objects, while the association between a name (identifier) and a specific *Type* object is made by *Declaration* objects. A value (literal) of a specific *Type* is encapsulated in a *Value* object,

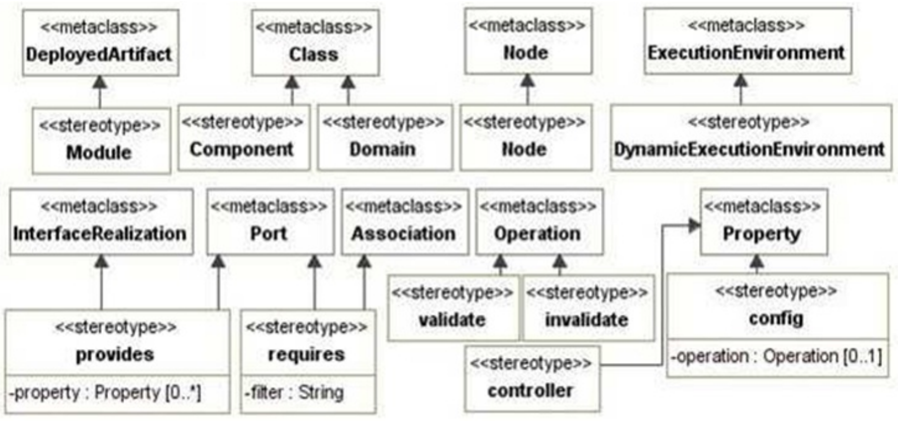

FIGURE 3. UML profile for iCOMPONENT

which are used in several places: *Variable* objects are used as user-friendly names for values stored at concrete memory addresses; a function returns a *Value*; the evaluation of an *Expression* produces a *Value*.

4. COMPONENT MODEL - iCOMPONENT

The platform independent component model for dynamic execution environments, called iCOMPONENT [3] aims to simplify the component development by allowing developers to concentrate only on implementing the business logic of the component and then to configure declaratively the component deployment. It is intended to be a framework which supports dynamic availability, and reconfiguration of components, in the style of OSGi [13] and iPOJO [2] principles.

A dynamic execution environment is added to the existing ComDeValCo framework. The component execution is built on top of the infrastructure built for executable UML structured activities by adding component manipulation capabilities to the existing Action Language [6].

In order to ease the process of implementing modular concepts, an adaptable infrastructure was created, based on a meta-model defining the concepts of module and execution environment. The dynamic execution environment loads modules and starts their execution provided that all dependencies are solved.

Traditional (static) execution environments load all modules of an application before starting its execution. The proposed model supports this scenario, but adds dynamic module load/unload facilities. Following this pattern, we can satisfy both (static) modular programming requirements and those of assembling applications from dynamic modules.

The UML profile of iCOMPONENT, as depicted in Figure 3, highlights its constituents:
• *Dynamic Execution Environment* - extends the UML 2.0 ExecutionEnvironment metaclass and represents an execution environment that provides capabilities for dynamic availability, reconfiguration, and composition of components;

• *Module* - which extends the UML Artifact metaclass and represents the unit of deployment. The set of model elements that are manifested in the artifact is indicated by the manifestation property of the *Artifact*;
• *Component* - extends the UML metaclass Class (from StructuredClasses) and represents a component type. By extending the metaclass Class, *Component* may have methods and attributes, and also may participate in associations and generalizations.
• *Nodes* and *Dynamic Execution Environments*. *Node* stereotype extends UML Node metaclass. A node may deploy several modules, and therefore possible several components instantiated by these modules. The *DynamicExecutionEnvironment* stereotype extends *Node*, in which we may use: (a) the properties associated to a service that is published by a component, and (b) dynamic binding using filters for selecting the services required by a component, in a similar way to the iPOJO approach [2].
• *Domain*. A *Domain* represents a complete configuration for system deployment, and consists of nodes and connectors between nodes. It may have several nodes, each containing several components. Here a node is seen as a process on a computer. The binding of the components in a specific domain is regardless of the nodes in which the components are deployed.

5. Agile MDA Development

The iCOMPONENT model is adapted such that it will allow assembling and deploying components and services altogether, and may serve as a basis for an agile MDA approach for software development. Any UML case tool can be used to construct the models. Working with executable models will enforce conformance to fUML specification. The approach consists of applying the following steps in the specified order:

- *Model Description*: this process will be accomplished in an incremental way, such that the model is described on the following layers: Services, Structure, Deployment;
- *Test-first development for simple components*: for each simple or monolithic component (implementing a unique specific functionality), programs are built in four-step increments:
  (1) Add a test. For each new functionality to be added, create first a test case, expressed in Action Language, which also includes assertion-based constructs. Test cases comply to UML Testing Profile [10].
  (2) Execute all tests. At first exection, the test added at previous step fails. The execution engine (virtual machine) of DEFCOMP is used for test execution also, similar to other automatic tools. The major difference is that DEFCOMP executes platform-independent models, PIMs, from which platform-dependent models or even complete implementations can be generated, including automatic generation of test cases.
  (3) Add production code expressed in Action Language.
  (4) Execute again all tests and go back to step (3) if at least one of the tests fails. When all tests succeed, start another development cycle (increment), going back to step (1).

[7] presents this approach in detail, including an example.

6. Conclusions and related work

The paper is focusing on how the ComDeValCo framework has been extended from an initial procedural model to a component model, that also offers a dynamic execution environment.

This component model can then be succesfully used as an Agile MDA development approach for component system. Several academic and commercial solutions targeting component models and service orientation are under development. Among the MDA approaches that are similar with the iComponent approach, we mention iPOJO and OSGi framework.

iPOJO (injected Plain Old Java Objects) [2] is a service-oriented component framework supporting the service-oriented component model concepts and dynamic availability of components, following the POJO approach (Plain Old Java Objects). All service-oriented aspects, such as service publication, the required service discovery and selection are managed by an associated component container. The operations in iPOJO are similar to iComponent operations, but our approach is platform-independent, while iPOJO is restricted to Java.Also, the current version of iPOJO does not provide a clear separation of the business logic and non-functional requirements for all operations discussed above.

Another framework which supports dynamic availability and reconfiguration of components is the OSGi framework [13], which offers a service-oriented component model. OSGi components are bound using a service-oriented interaction pattern, and their structure is described declaratively. Again, OSGi does not offer a clear separation between business logic and the non-functional requirements.

As a conclusion, the approach constructed using iComponent has the following benefits:

- Combines executable models, agile MDA, and platform-independent service-oriented component models for dynamic execution environments;
- Offers clear separation between business logic and non-functional aspects;
- Supports rapid development of application - due to the simplified graphical capabilities.

Acknowledgements. This work was supported by the grant ID 546, sponsored by NURC - Romanian National University Research Council (CNCSIS).

References

[1] Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J. and Neema, S., *Developing Applications Using Model-Driven Design Environments*, Computer **39** (2006), 33–40

[2] Escoffier, C. and Hall, R. S., Dynamically Adaptable Applications with iPOJO Service Components, In *6th Conf. on Software Composition (SC07)*, 2007, 113-128

[3] Lazăr, I., Pârv, B., Motogna, S., Czibula, I-G. and Lazăr, C-L., iComponent: A Platform-independent Component Model for Dynamic Execution Environment, In *Proc. of 10th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing SYNASC 2008*, Timisoara, September, 2008. IEEE Conference Publishing Services, 257–264

[4] Mellor, Stephen J., *Agile MDA*, Technical report, Project Technology, Inc., 2005

[5] Mellor, Stephen J. and Balcer, M. J., *Executable UML: A Foundation for Model-Driven Architecture*, Addison Wesley, 2002

[6] Motogna, S., Pârv, B., Lazăr, I., Czibula, I. G. and Lazăr, C. L., *Extension of an OCL-based Executable UML Components Action Language*, Studia UBB, Informatica, **53** (2008), No. 2, 15–26

[7] Motogna, S., Lazăr, I., Pârv, B. and Czibula, I-G., *An Agile MDA Approach for Service-Oriented Components*, Electron. Notes Theor. Comput. Sci. **253** (2009), 95–110

[8] OASIS RIM, *Registry Information Model*, http://docs.oasis-open.org/ regrep/v3.0/ specs/regrep-rim-3.0-os.pdf

[9] Object Management Group, *MDA Guide*, Version 1.0.1. (2003), URL: http://www.omg.org/ cgi-bin/doc?omg/ 03-06-01.pdf

[10] Schieferdecker, I., Ru Dai, Z., Grabowski, J. and Rennoch, A., *The UML 2.0 Testing Profile Specification*, Proc. TestCom 2003, LNCS 2644, 79–94

[11] Object Management Group, *UML Superstructure Specification*, Rev. 2.1.2, October 2007. http://www.omg.org/ spec/UML/2.1.2/ Superstructure/PDF/

[12] Object Management Group, *Semantics of a Foundational Subset for Executable UML Models,*, Rev. 1.0, Beta 1, 2008, http://www.omg.org/ spec/FUML/

[13] OSGi Alliance, *OSGi Service Platform Core Specification*, Release 4, Version 4.1., 2007, http:// www.osgi.org/

[14] Pârv, B., Motogna, S., Lazăr, I., Czibula, I. G. and Lazăr, C. L., *ComDe- ValCo - a Framework for Software Component Definition, Validation, and Composition*, Stud. Univ. Babeş-Bolyai Inform. **52** (2007), No. 2, 59-68

DEPARTMENT OF COMPUTER SCIENCE
"BABES-BOLYAI" UNIVERSITY
KOGALNICEANU 1, 40004 CLUJ-NAPOCA, ROMANIA
E-mail address: {motogna,ilazar,bparv}@cs.ubbcluj.ro