



ComDeValCo framework: designing software components and systems using MDD, executable models, and TDD

Bazil PÂRV

Babeş-Bolyai University

Cluj-Napoca

email: bparv@cs.ubbcluj.ro

Simona-Claudia

MOTOGNA

Babeş-Bolyai University

Cluj-Napoca

email: motogna@cs.ubbcluj.ro

Ioan LAZĂR

Babeş-Bolyai University

Cluj-Napoca

email: ilazar@cs.ubbcluj.ro

Istvan-Gergely CZIBULA

Babeş-Bolyai University

Cluj-Napoca

email: istvanc@cs.ubbcluj.ro

Codruţ-Lucian LAZĂR

Babeş-Bolyai University

Cluj-Napoca

email: clazar@cs.ubbcluj.ro

Abstract. This paper provides an overall description of COMDEVALCO, a framework for component definition, validation and composition. It comprises a modeling language, a component repository and a set of tools aimed to assist developers in all activities above.

1 Introduction

The main benefits of component-based development are [32]: (i) loose coupling among the application components, (ii) third-party component selection, and

Computing Classification System 1998: D.2.11

Mathematics Subject Classification 2010: 68Q60

Key words and phrases: model-driven development, test-driven development, executable models

(iii) increased component reuse. In traditional component-based approaches, the set of components is statically configured, i.e. the benefits outlined above typically extend only to the development portion of the software system life-cycle, not to the run-time portion [2].

Modern component models and frameworks allow components unavailable at the time of application construction to be later integrated into the application, after its deployment [25]. Such frameworks use a dynamic execution environment, providing the following: (a) *dynamic availability of components* – components can be installed, updated, and removed at runtime, and their provided and required interfaces are managed dynamically; (b) *dynamic re-configuration* – the configuration properties of a running component can be changed, and (c) *dynamic composition* – new components can be composed at runtime from other existing components.

Development approach is another key aspect of component-based development. 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. It is convenient to build simple models, without great intellectual effort and considerable investments in time. What is really important regarding resulting models is their accessibility, ease of understanding and analyzing, and a reasonable degree of formality.

COMDEVALCO project started three years ago having the above requirements in mind. Its main goal is to help developers in the component-based development process, i.e. to build, validate and assemble simple or complex components, using a platform-independent modeling language and a set of tools.

The structure of the paper is as follows. After this introductory section, the next two contain background information and a short description of the evolution of COMDEVALCO framework. The sections 4 to 6 describe in some detail the components of the framework, i.e. the modeling language, the component repository and the toolset, following the natural evolution of programming paradigms, from procedural to component-based, with modular and object-oriented as intermediate steps. The last section draws some conclusions and states further work to be done.

2 Background

The construction of software components is simplified by (1) applying a model-driven development (MDD) approach and (2) separating the business logic of a component from the nonfunctional requirements.

2.1 Model-driven development

Model-Driven Architecture (MDA) framework allows system specification independently of a particular platform and for transforming the system specification into one for a particular platform. MDA is considered the OMG approach to Model Driven Engineering (MDE), which is a development solution to applications that have to deal with increased platform complexity and domain concepts, aiming to raise the level of abstraction in program specification and to increase automation in program development. According to MDE, the system development is based on models at different levels of abstraction; later, model transformations partially automate some steps of program development. Besides MDA, the other MDE approach is Domain Specific Modeling.

Unfortunately, development processes based on MDA are not widely used today because most of them are viewed as heavy-weight processes – they cannot deliver (incrementally) partial implementations to be executed as soon as possible.

In this context, an alternative is to execute UML models. For such processes, models must act just like code, and UML 2 and its Action Semantics [22] provide a foundation to construct executable models. A model is executable if it contains 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 described in [14] has an execution semantics for a subset of actions sufficient for computational completeness. It includes two basic elements: an action language, specifying the elements that can be used, and an operational semantics, establishing how the elements can be placed in a model, and how the model can be interpreted. Again, there are some inconveniences: creating reasonable-sized executable UML models is difficult, because the UML primitives from the UML Action Semantics package are too fine-grained.

Another alternative is represented by agile MDA processes [15], which apply the main Agile Alliance principles (e.g. testing first, immediate execution) into a classical MDA process. The requirement of making such process models to act just like code, means that they must be executable.

2.2 Separation of the business logic and non-functional requirements

This principle targets two important aspects of software development. First, the developer will concentrate on the functionality with no concern on data access or presentation issues. Second, such an approach supports reuse on a larger scale. Early commercial component models such as Component Object Model (COM) (Microsoft, 1995), Enterprise Java-Beans 2.1 (Sun, 2003), and CORBA Component Model (OMG, 2002) propose specific application programming interfaces, so they do not offer a clear separation between functional and non-functional requirements. These approaches increase the development costs and decrease the component's potential of reuse.

There are many other component models developed by the academic community which provide solutions for the separation problem but do not provide dynamic execution environment features [3]. Some of these frameworks – such as iPOJO [2], OSGi framework [25], and SCA [23] – have similar features to our COMDeVALCo approach.

3 ComDeValCo evolution

MDA and Agile principles are the driving forces of our proposal, COMDeVALCo – a framework for Software Component Definition, Validation, and Composition [26].

The framework is intended to cover two sub-processes of the component-based development: component development and component-based system development.

Component development starts with its definition, using an object-oriented modeling language, and graphical tools. Modeling language provides the necessary precision and consistency, and the use of graphical tools simplifies developer's work. Once defined, component models are passed to a verification and validation (V & V) step, which checks their correctness and evaluates their performance. When a component passes V & V step, it is stored in a component repository, for later (re)use.

Component-based system development takes the components from repository and uses graphical tools, for: (a) selecting components fulfilling a specific requirement, (b) performing consistency checks regarding component assembly and (c) including a component in the already existing architecture of the target system. When the assembly step is completed, and the target system is completely built, other tools will perform system V & V, as well as

performance evaluation operations on it.

COMDEVALCO framework consists of:

- a *modeling language*, used to describe component models;
- a *component repository*, which stores and retrieves software components and systems, and
- a *toolset*, aimed to help developers to define, check, and validate software components and systems, and to provide maintenance operations for the component repository.

The next three sections describe in more detail the three components of the COMDEVALCO framework. A detailed presentation of the procedural and modular stages of COMDEVALCO project is given in [28, 29], while object-oriented and component-based features are discussed in [30].

4 The modeling language

The software component model is described by a platform-independent modeling language, having the following features:

- all elements are objects, instances of classes defined at logical level, with no relationship to a concrete object-oriented programming language;
- top-level language constructs cover both complete software systems and concrete software components;
- there is a 1:1 relationship between the internal representation of the component model – an aggregated object – and its external representation on a persistent media, using various formats: XML, object serialization, etc.

The initial evolution of the modeling language includes the following steps and elements:

- **initial object model**, developed in the preliminary phase of the project (feasibility study);
- **Procedural Action Language (PAL)**, with a concrete syntax and graphical notations for statements and program units;
- **execution infrastructure**, including the concepts of module and execution environment; the module has both classical (including several data types, functions, and procedures) and modern (containing several data types, including components – unit of deployment) semantics;
- **type system**, containing primitive types, then vectors and structures.

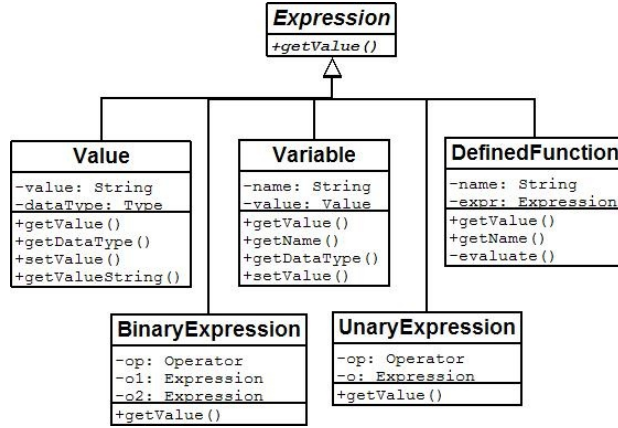


Figure 1: Expression class hierarchy

4.1 Initial object model

The initial object model was structured on three layers, from simple to complex: (1) low-level (syntactical) constructs, (2) execution control constructs (statements) and (3) program units.

The *lowest layer* (as Figure 1 shows, see for more details [27]) contains basic constructs of the modeling language, classes `Type`, `Declaration`, `Value`, `Variable` and `Expression`. The concrete subclasses of the abstract class `Expression` are: `Value`, `Variable`, `BinaryExpression`, `UnaryExpression`, and `DefinedFunction` are also subclasses of `Expression`.

The *middle layer* contains objects which model the execution control, all of them inheriting from a base class `Statement`. This class hierarchy uses *Composite* and *Interpreter* design patterns, as Figure 2 shows. `Statement` has a single abstract operation, `execute()`, which produces a state change in many situations.

The subclasses of `SimpleStatement` are the following: `CallStatement`, `AssignmentStatement`, `InputStatement`, `OutputStatement`, `LoopStatement` and `BranchStatement`. They cover all control structures of imperative programming and have appropriate implementations for their `execute()` methods.

Program units considered in the initial model are shown in Figure 3: `Program`, `Procedure` and `Function`. They belong to the *upper layer* of the modeling language.

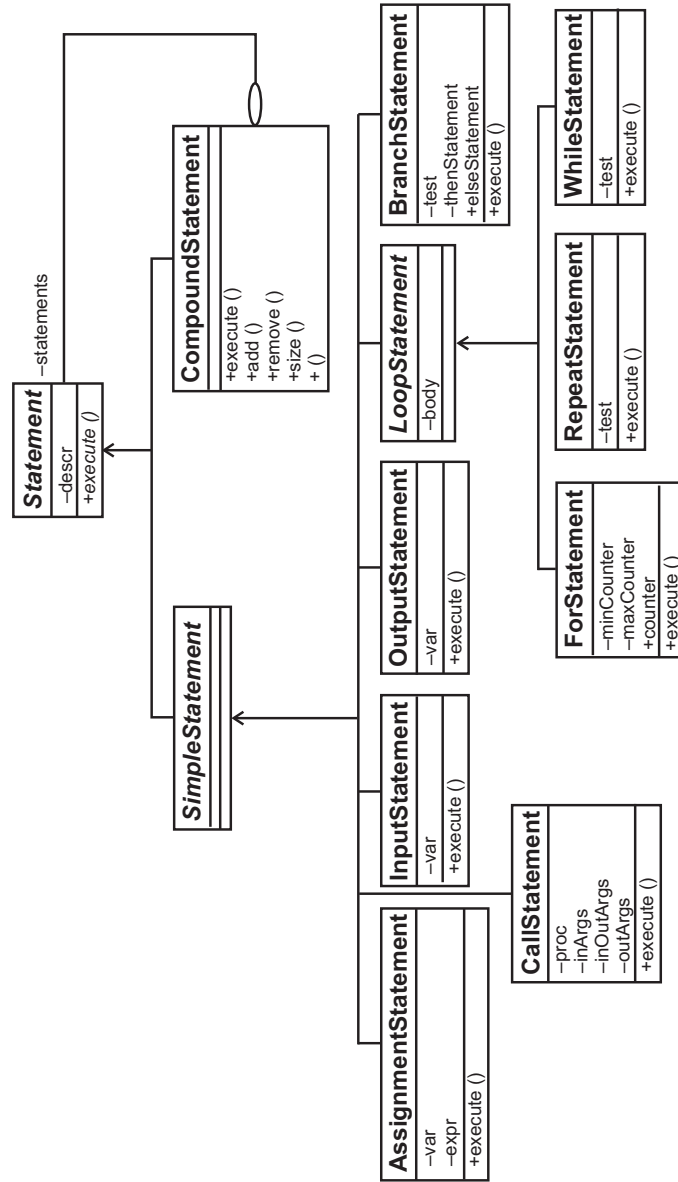


Figure 2: Statement class hierarchy

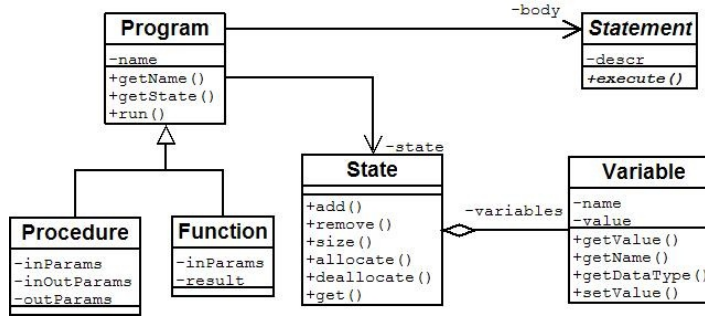


Figure 3: Program units

Program objects are executable components, having a name, a state and a body of statements; their state is made up of all **Variable** objects local to the component, and the body is a **Statement** object. The only operation is `run()`, implemented by the call `body.execute()`.

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. A **Function** object has just a list of *in* parameters and returns a **Value** object.

4.2 PAL, procedural action language

Having the above initial object model in mind, an action language PAL (Procedural Action Language) was developed. PAL, described in more detail in [4], has a concrete (textual) syntax and graphical notations corresponding to statement and component objects. The concrete syntax allows the developer to express quickly the body of an operation, while graphical notations help in understanding the control flow. A subset of PAL constructs is shown in Figure 4.

4.3 Modular constructs

The modeling language was improved in other two directions, by (1) including module and execution environment, and (2) extending its type system with structured types.

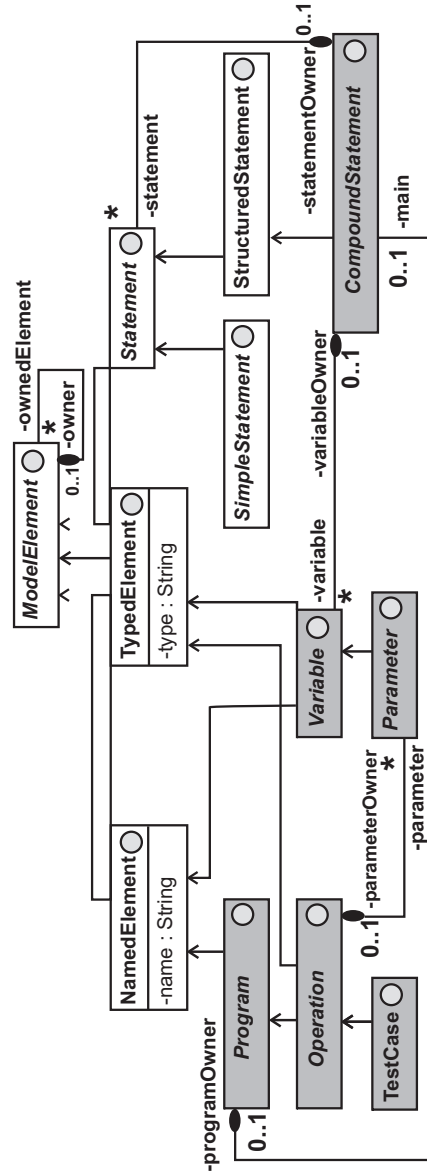


Figure 4: PAL – extract from metamodel

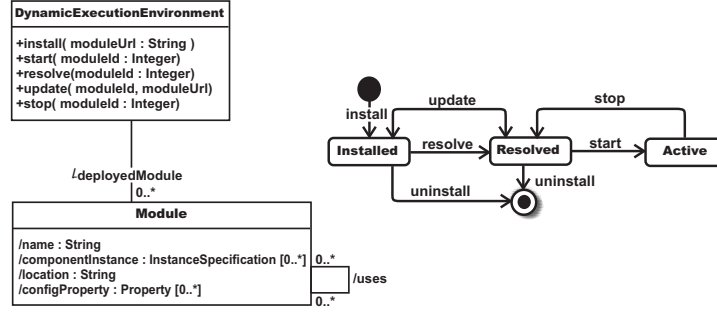


Figure 5: The execution environment and the module – excerpt from the metamodel

4.3.1 Module and dynamic execution environment

The module is considered in the general case, as a deployment unit. It can include either (a) several data types, functions, and procedures – as in the traditional modular programming model, or (b) several data types, including components – as in the case of component-based programming. The elements included in a module may use other elements from other modules; in other words, there are dependency relations between modules, which must be specified during module definition phase.

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*, shown in Figure 5.

The dynamic execution environment loads modules and starts their execution provided that all dependencies are solved. The state diagram (Figure 5, right) depicts the states of a module and state transitions.

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. Some of the existing execution environments – like OSGi [25] – have these features, assembling Java applications from dynamic modules.

Following this pattern, we can satisfy both (static) modular programming requirements and those of assembling applications from dynamic modules. Our proposal is described in greater detail in [5].

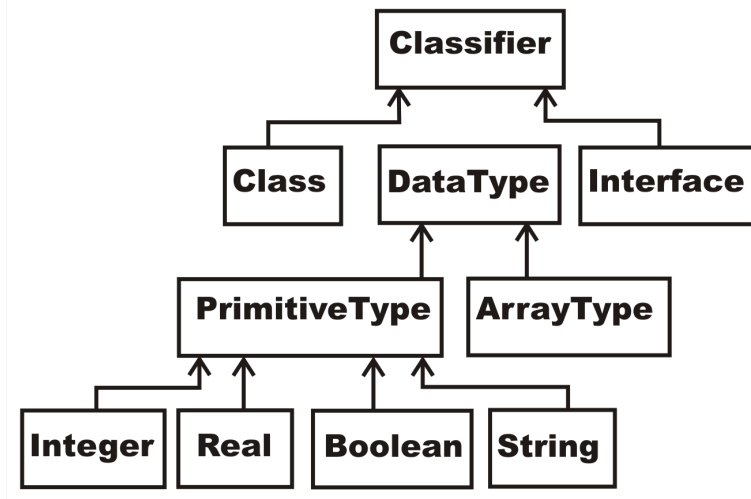


Figure 6: Metaclasses for data types

4.3.2 Type system extensions

The initial modeling language used only primitive data types. Currently, its type system includes a new **ArrayType** class – see Figure 6. Also, PAL grammar was changed to allow the definition of tables (vectors) and structured types, like lists. These achievements are described in detail in [17], which proposes an extensible data type hierarchy.

Papers [5, 6] discuss in great detail UML stereotypes aimed to define new concepts included in the modeling language. As an example, Figure 7 shows UML stereotypes for modules and components.

4.4 Object-oriented constructs

The infrastructure aimed to support procedural and modular programming was extended in order to allow the implementation of object-oriented, component-based and service-oriented concepts. The module, considered as unit of deployment, contains user-defined types, including classes and interfaces.

4.4.1 Classes and interfaces

Classes and interfaces are defined according to the UML standard and fUML specification. The latter, Foundational UML, published by OMG in 2008 [21],

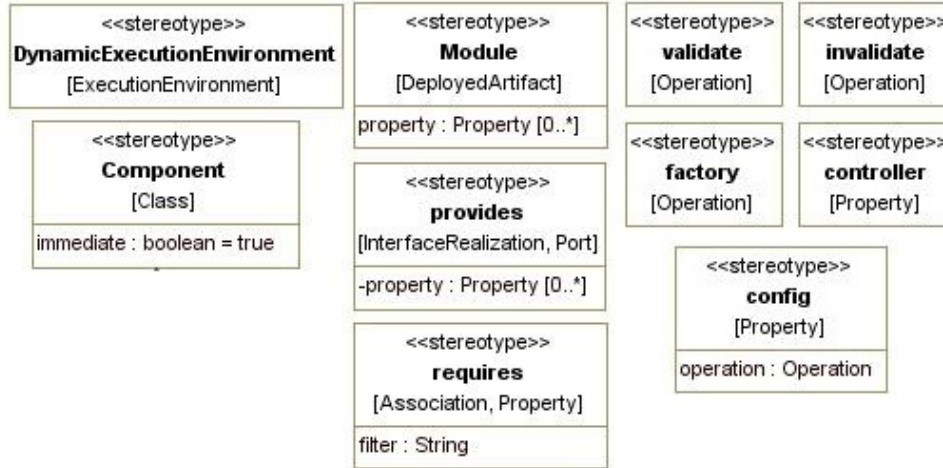


Figure 7: UML stereotypes for defining modules and components

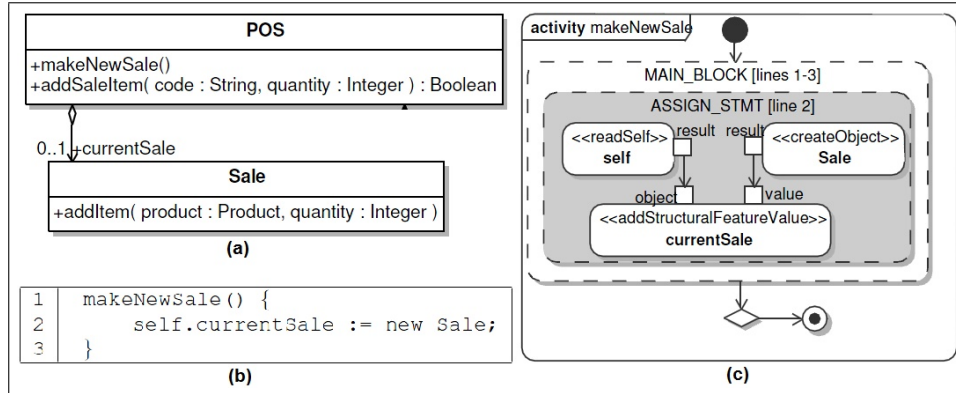


Figure 8: (a) Domain model; (b) Concrete syntax and (c) Graphical description for an operation definition

establishes a UML subset and a semantics for model execution.

COMDEVALCO workbench, described in more detail in the next section, allows the developer to define classes and interfaces according to UML standard. Figure 8 (a) shows a Point of Sale (POS) domain model fragment. Figure 8 (b) and (c) shows a simple example of how operations are defined. The textual syntax for constructing UML models is very important for their rapid

development. As a matter of fact, OMG issued a request for proposing such a textual syntax; in this respect, our proposal can be considered as a response from academic community.

4.4.2 PAL improvements

Another important contribution is the synchronization between the textual representation – Figure 8 (b) – and the graphical one – Figure 8 (c), compliant to the fUML standard. COMDEVALCO workbench allows the user to define operations using either textual or graphical perspectives, and to switch between the two views at any moment.

The previous version of action language PAL, implementing the modular paradigm concepts, used primitive types and vectors. Its type system now includes *Class* and *Interface* types; also, PAL grammar was updated to allow the use of these new constructs.

These results are described in full detail in [9].

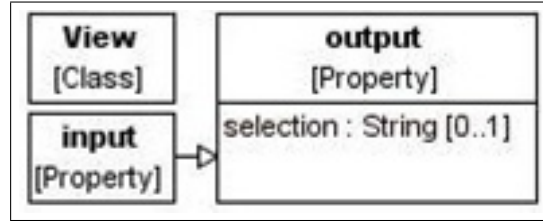


Figure 9: Stereotypes for user operations

4.5 Component-based constructs

The first contribution is a platform-independent model for components, *iComponent*, used to model component-based and service-oriented systems. Later on, these platform-independent models will be automatically transformed into platform-dependent models like OSGi, Sun EJB3, JBoss Seam, Grails, and so on by using appropriate mappings. The first sub-section contains more details about this topic.

In order to validate the proposed ideas, some particular applications were considered, involving component construction for OSGi and Web systems. During these efforts, we proposed a new prototyping approach aimed to speed up the model construction and established the mappings between the platform-independent model described in the first sub-section and the target platform-

specific ones. The last sub-section describe these results in more detail and provide the full references.

4.5.1 *iComponent* meta-model

The platform-independent meta-model for component definition, *iComponent*, is shown in Figure 10. The components are defined as simple classes (*Component*) which implement (*provides*) and use (*requires*) some interfaces. Components are assembled into modules (*Module*) and are deployed into domains (*Domain*) which establish a process configuration needed for the system to work (*Node* or *DynamicExecutionEnvironment*).

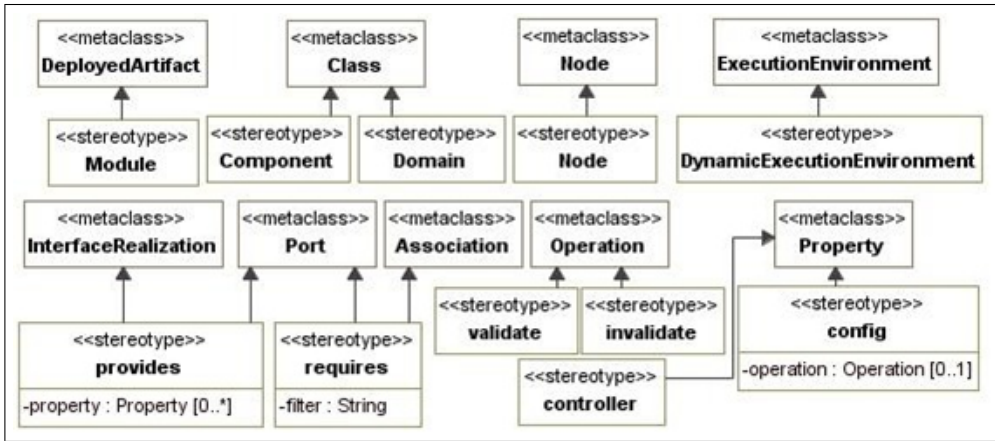


Figure 10: *iComponent* – UML stereotypes for component definition

The dynamic execution environment manages the component life-cycle, as Figure 11 shows. The *validate*, *invalidate*, *config* and *controller* stereotypes can be used to intercept and generate events related to the component state.

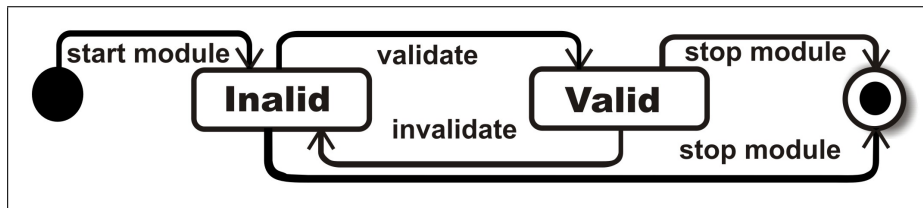


Figure 11: Component life-cycle in a dynamic execution environment

Component binding is performed automatically by the dynamic execution

environment, which injects the appropriate dependencies between components. Component selection takes into account the interfaces they implement and some other features which can be associated to the implemented interfaces (using the properties of *provides* and *requires* stereotypes).

The papers [5, 18] describe in full detail the above presented results.

4.5.2 PAL extension

The modeling language and PAL grammar were extended to cover the use of components and to provide support for Web application modeling. Figure 12 contains an excerpt from the corresponding UML profile. This support was introduced as a necessary step for model validation. As we mentioned earlier, the target of COMDEVALCO project is to allow the modeling of a large variety of component-based and service-oriented systems. For example, the platform-independent model described in the first subsection allows us to model OSGi and OASIS Service Component Architecture systems.

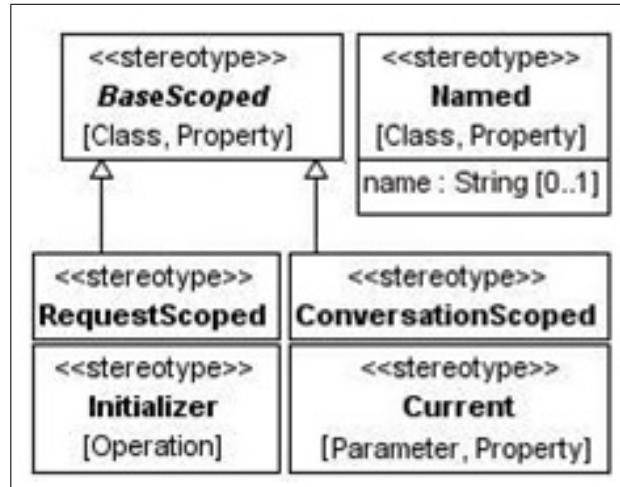


Figure 12: Stereotypes for web applications

The papers [18, 9] describe these results in more detail.

4.6 Analysis of robustness

At the end of 2009, OMG published the second revision of Foundational UML (fUML) specification. Also, at the beginning of 2010, OMG published the first version of Alf (Action language for fUML).

The proposed platform-independent infrastructure of COMDeVALCo is based on the two above-mentioned specifications. The earlier versions of our model were made compliant with these specifications. This way, we are entitled to say that our model and development methods are among the first releases of this kind, based on public OMG standards. Papers [11, 12, 13] describe in more detail these achievements.

5 ComDeValCo workbench

COMDeVALCo *toolset* is intended to automate many tasks and to assist developers in performing component definition and V & V, 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 – software system definition by component assembly;
- VALSYS – software system V & V;
- SIMCOMP, SIMSYS – component and software system simulation;
- GENEXE – automatic generation of (platform-specific) executable components and software systems.

5.1 First developments: DEFCOMP, VALCOMP, ComDeValCo workbench

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 (see Figure 13) are synchronized, acting on the same model, which uses PAL meta-model.

VALCOMP tool 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. The proposed Agile MDA process builds programs in four-step increments:

1. *Add a test.* For each new functionality to be added, create first a test case, expressed in PAL, which also includes assertion-based constructs. Test cases comply to UML Testing Profile [19, 20].

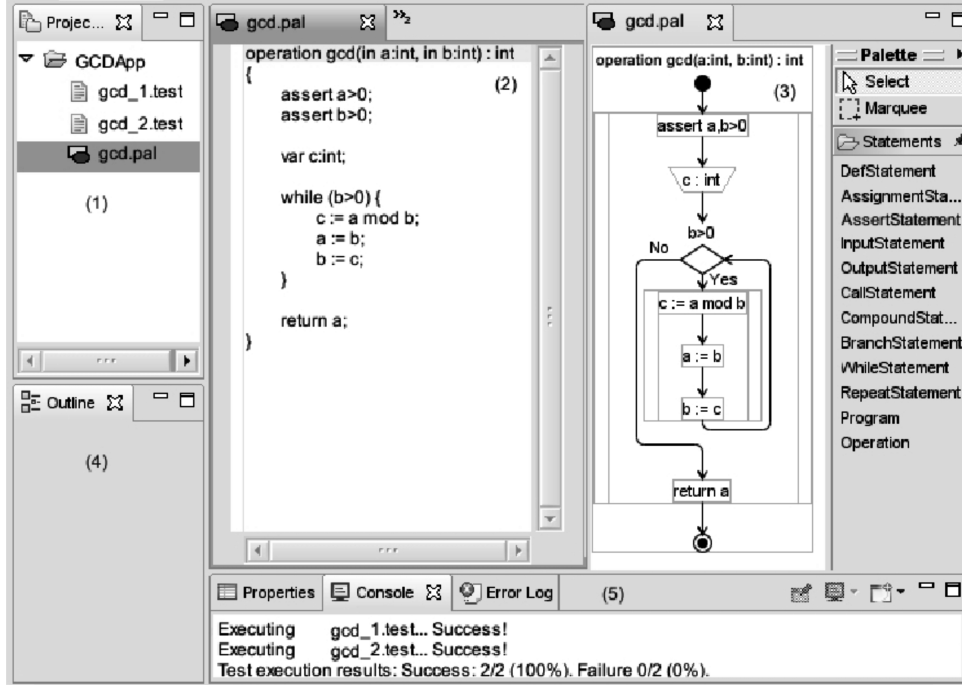


Figure 13: COMDEVALCO workbench: modeling perspective

2. *Execute all tests.* At first execution, 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 PAL.
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).

DEFCOMP has a debugging perspective also (see Figure 14), and the developer can adnotate the model with breakpoints. Besides assertions, used for testing and functional code, PAL includes pre- and post-conditions for procedures/functions and invariants for cycles.

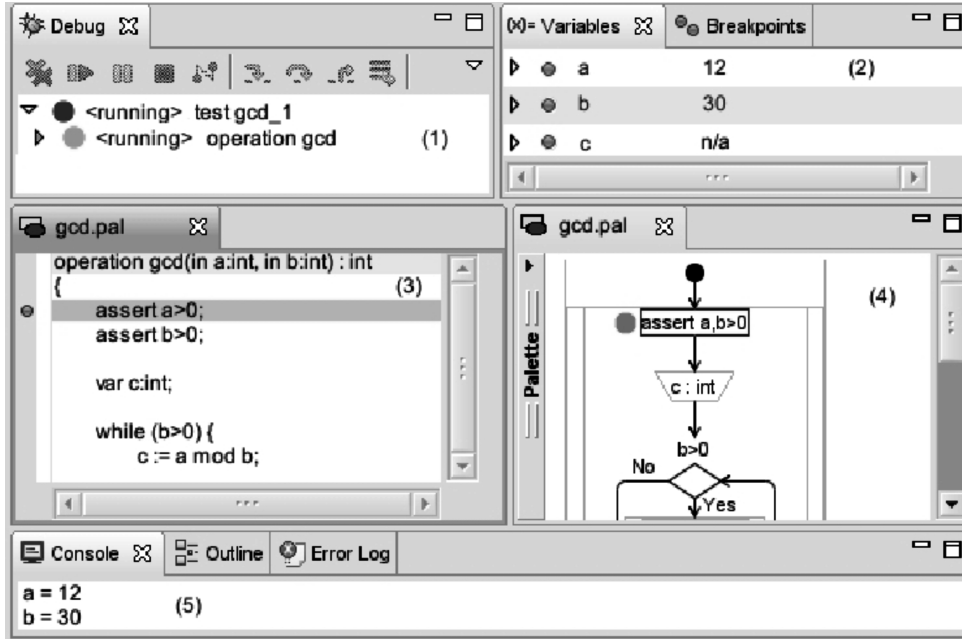


Figure 14: COMDEVALCO workbench: debugging/simulation perspective

Later on, DEFCOMP and VALCOMP, were included in the so-called COMDEVALCO *workbench*.

5.2 Modular paradigm improvements: DEFCOMP, VALCOMP, DEFSYS and VALSYS

The functionality of DEFCOMP and VALCOMP, parts of the COMDEVALCO *workbench*, was extended to cover the two new concepts included in the modeling language, module and execution environment. The results are described in the papers [1, 5, 7].

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

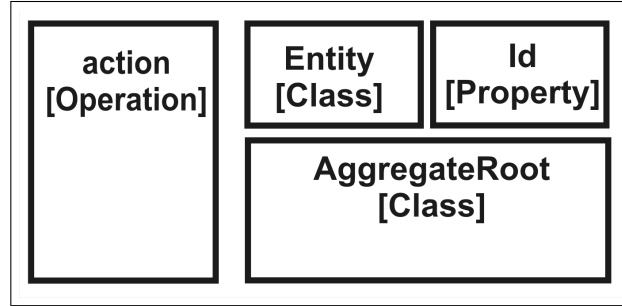


Figure 15: Stereotypes for domain and actions

```
wizard CreateEntity {
  guard : self.isKindOf(Class)
  title : "Create entity " + self.name
  do {
    var id : new Property; id.name := 'id';
    id.class := self;
    -- other attributes and operations added
  }
}
```

Figure 16: Entity creation

validation activities. These results are described in more detail in [7].

5.3 Object- and component-based improvements: DEFCOMP and VALCOMP

As parts of the COMDEVALCO workbench, DEFCOMP and VALCOMP tools were extended to allow the use of new updates of the modeling language referring to object-oriented and component-based concepts. Also, these updates include support for architecture (Model-View-Controller architectural pattern) and domain modeling. Figures 9 and 15 contain excerpts from the corresponding UML profiles.

Other added functionality refers to the creation of new model elements by applying recommended design and architecture best practices. These were implemented as M2M transformations, as the Figure 16 shows.

Papers [18, 8, 10] describe in full detail these achievements.

Additionally, DEFComp and VALComp tools were extended to support new constructions included in the modeling language and illustrated in Figure 10. Besides these improvements, a new development method was proposed, involving the creation of the following models: service (interface) models, structural models (component composition), implementation models (for simple components), verification models (for simple and compound components), and deployment models (assembling a component-based system).

These improvements are discussed in full detail in the papers [6, 18].

5.4 SIMCOMP and SIMSYS

During system development, need to be considered at least the following key issues related to requirements: expressing requirements as clearly as possible, appropriate mapping of requirements to system components and checking that all requirements are implemented. Our proposed methodology takes into account these issues, as described below.

In order to improve the clarity of requirements, a method that automatically translates textual description of use cases into executable models was proposed. A component (more precisely an active object) is associated to each use case, in order to define the behaviour described by the use case's scenarios. The UML activity associated to the component is automatically generated, starting from textual description. Because the generated models are executable, the developer is able to execute the use case at once; these experiments could help the developer to find some defects in the requirements and to improve their clarity.

Another active component is associated to a set of related use cases; its meaning is to describe the integration of individual behaviours of use cases into a subsystem's behaviour. By experimenting the resulting executable model, developers could observe and repair the integration of requirements.

In order to check that all requirements are implemented, an behaviour-driven development approach was considered. The novelty of this approach is given by the fact that it is applied to executable models. Developers may describe requirements in the form of **given-when-then** scenarios: **given** "some context" **when** "something happens" **then** "the system enters in some state". Each scenario is detailed by an activity; when this activity is executed, it will provide an answer related to the specified system state. Papers [12, 13] describe these results in more detail.

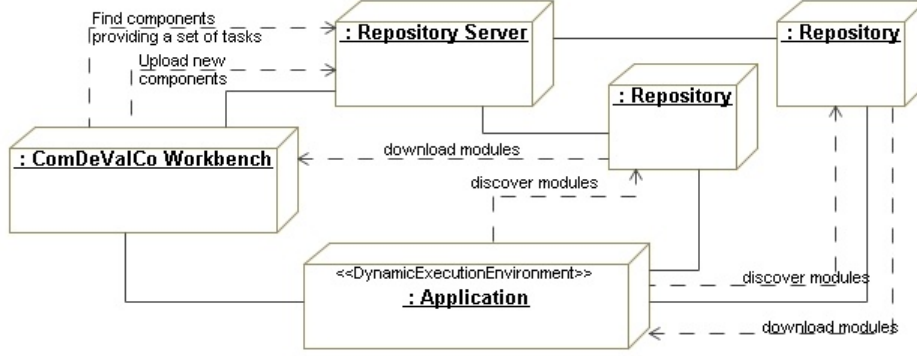


Figure 17: COMDEVALCO framework: interactions

5.5 GENEXE

One of most difficult tasks of software engineering is the complete generation of executable code on a specific platform, without additional (manual) coding activities using platform-dependent languages. Using COMDEVALCO approach, the solution is simple because all models are executable, and the component behaviour is completely described by these models.

In order to produce executable code, a mapping between action languages (PAL or Alf) and the considered concrete platform(s) is needed. Currently, the concrete platform is Java, but COMDEVALCO workbench is able to support other platforms. Papers [11, 13] describe these achievements in greater detail.

6 The component repository

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.

Figure 17 shows the interactions between component repositories, COMDEVALCO workbench and client applications. The details of the proposed solution are presented in [5, 16].

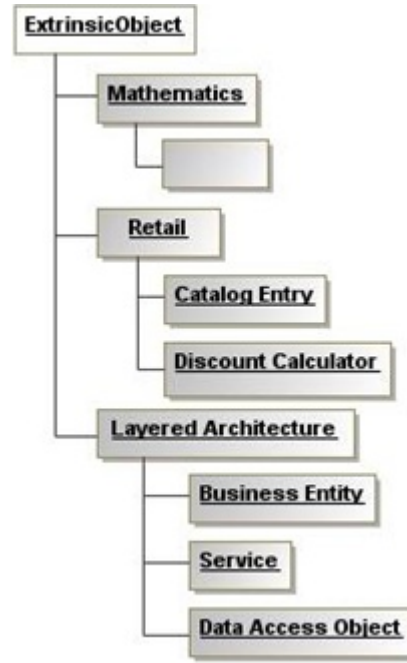


Figure 18: Classification scheme

6.1 Component classification criteria

The starting point in the work of providing a correct taxonomy for components was the establishment of classification criteria. Several concrete approaches were considered; Figure 18 shows such a classification scheme.

Software components can be classified upon different criteria, including information domain (e.g. **Retail**) and functionality (e.g. **Service**). These criteria may be used in searching for components stored in the repositories. The paper [16] discusses these matters in great detail.

6.2 Component representation in the repository

In order to describe the representation of components in the repository, an object model (shown in Figure 19) was defined. This model includes all component types covered by the modeling language and allows for adding new ones. Its classes and their relationships are described in detail in [5, 16].

RegistryObject is the root of all objects managed by the component repos-

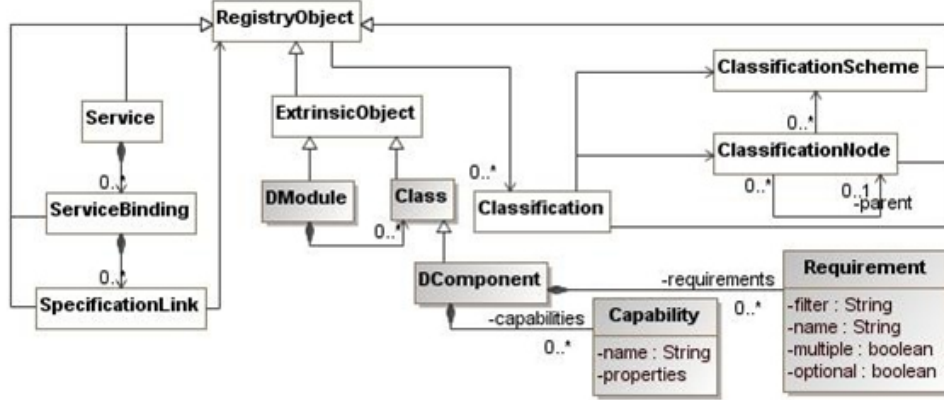


Figure 19: Representation of objects in repositories

itory, from which all components inherit thru `ExtrinsicObject` class. The two concepts, *classification* and *classification scheme* have distinct classes in this hierarchy, thus providing a greater degree of flexibility.

Component representation format complies to the OASIS RIM (Registry Information Model) standard [24]. To achieve this, the class `ExtrinsicObject` was extended by adding subclasses specific to component-based applications: `DModule` – module, `DComponent` – component, `Capability` – functionality provided by a component, and `Requirement` – the dependencies of a component. The paper [16] discusses these achievements in great detail.

6.3 Adding components to component repository

When a component passes all V & V checks, it can be stored in the repository, for later (re)use. Several types of components were considered, implementing different functionality: console-type and graphical user interfaces, CRUD operations, MVC architectural pattern, behavioral design patterns and so on. These components may be reused either to build more complex components or to assemble (build) software applications.

Another benefit of this stage was the validation of the proposed development methodology. Additionally, the components in the repository may serve as usage examples for the proposed conceptual infrastructure. Papers [12, 13] describe in more detail these aspects.

Originally, all components in the repository were platform-independent.

By using GENEXE, the developer can produce platform-dependent components, that need also to be stored in the repository. Thus, requirements related to component repository were changed, allowing it to manage platform-dependent components.

Another real-world issue refers to the fact that a software system may contain components of both worlds: platform-independent and platform-dependent. More precisely, the execution of a platform-independent component may trigger the execution of a platform-dependent component. According to our knowledge so far, only CASE tools for designing hardware components (e.g. IBM Rhapsody) cover this situation.

7 Conclusions and further work

In our opinion, the main contributions of the COMDEVALCO framework are: a concrete syntax for fUML, *iComponent* – a platform-independent component model for dynamic execution environments, and an agile MDA approach for building executable models.

Compared to other concrete approaches, like iPOJO and SCA, our proposal is platform-independent. By using *iComponent* profile, COMDEVALCO models can be constructed using any UML tool and can be executed in any executable UML tool.

As we mentioned above, the original idea of using platform-independent executable models was of a great importance. Subsequent developments originated especially by OMG and related to fUML and Alf prove that our research fits into the mainstream of current ideas and standards. By making COMDEVALCO infrastructure compliant to very recent OMG specifications related to fUML and Alf, our solution is among the first ones able to build platform-independent components based on executable models.

The intended use of COMDEVALCO framework is twofold. The first target is componentbased development, since COMDEVALCO conforms to UML and MDA standards, providing a complete framework for executable service-oriented component models.

The second target is of an academic nature. COMDEVALCO can be used in many Software Engineering courses as an example of applying model-driven principles in the software development process. At a beginner level, students get used earlier with model-based development, while at an advanced level, the framework may be used for model-driven V & V tasks.

Future developments of COMDEVALCO framework include: improving model

V & V capabilities, model transformation and SOAML [31] compliance. More precisely, model V & V will cover the investigation of multi-modal test execution techniques in the context of fUML by using UML composite structures and test data concepts.

The COMDEVAlCo workbench will also include other model transformation capabilities, allowing the generation of full executable code from executable models. The download server is planned to be live in the Spring of 2011.

Acknowledgements

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

References

- [1] I. G. Czibula, C. L. Lazăr, B. Pârv, S. Motogna, I. Lazăr, COMDEVAlCo Tools for procedural paradigm, *Int. J. of Computers, Communication and Control* **3**, suppl. issue (2008) 243–247. [⇒65](#)
- [2] C. Escofier, R. S. Hall, Dynamically adaptable applications with iPOJO service components, *Proc. 6th Conference on Software Composition*, M. Lumpe & W. Vanderperre (Eds.), *Lecture Notes in Comput. Sci.* **4829** (2007) 113–128. [⇒49](#), [51](#)
- [3] K. K. Lau, Z. Wang, A taxonomy of software component models, *Proc. 31st EUROMICRO Conference of Software Engineering and Advanced Applications*, Porto, Portugal, 2005, pp. 88–95. [⇒51](#)
- [4] I. Lazăr, B. Pârv, S. Motogna, I.G. Czibula, C. L. Lazăr, An agile MDA approach for executable UML structured activities, *Studia Univ. Babeş-Bolyai, Inform.* **52**, 2 (2007) 101–114. [⇒55](#)
- [5] I. Lazăr, B. Pârv, S. Motogna, I. G. Czibula, C. L. Lazăr, An agile MDA approach for the development of service-oriented component-based applications, *Proc. CANS08 Complexity and Intelligence of the Artificial and Natural Complex Systems*, Tg. Mureş, Romania, 2008, pp. 37–46. [⇒57](#), [58](#), [62](#), [65](#), [68](#), [69](#)
- [6] I. Lazăr, B. Pârv, S. Motogna, I. G. Czibula, C. L. Lazăr, iComponent: a platform-independent component model for dynamic execution environ-

- ments, *Proc. 10th SYNASC Symposium of Symbolic and Numeric Algorithms for Scientific Computing*, Timișoara, Romania, 2008, pp. 257–264. ⇒ 58, 67
- [7] I. Lazăr, C. L. Lazăr, On simplifying the construction of executable UML structured activities, *Studia Univ. Babeș-Bolyai, Inform.* **53**, 2 (2008) 147–160. ⇒ 65, 66
- [8] I. Lazăr, S. Motogna, B. Pârv, Rapid prototyping of conversational web flows, *Proc. 2nd KEPT International Conference Knowledge Engineering: Principles and Techniques*, Cluj-Napoca, Romania, 2009, pp. 223–230. ⇒ 66
- [9] I. Lazăr, I. G. Czibula, S. Motogna, B. Pârv, C. L. Lazăr, Rapid prototyping of service-oriented applications on OSGi platform, *Proc. 4th BCI Balkan Conference in Informatics*, Thessaloniki, Greece, 2009, pp. 217–222. ⇒ 60, 62
- [10] I. Lazăr, B. Pârv, S. Motogna, I. G. Czibula, C. L. Lazăr, Using a fUML action language to construct UML models, *Proc. 11th SYNASC Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timișoara, Romania, 2009, pp. 93–101. ⇒ 66
- [11] C. L. Lazăr, I. Lazăr, B. Pârv, S. Motogna, I. G. Czibula, Code generation from an fUML action Language, *Int. J. of Computers, Communication and Control* **5**, 5 (2010) 775–782. ⇒ 63, 68
- [12] I. Lazăr, S. Motogna, B. Pârv, Behaviour-driven development of foundational UML components, *Electronic Notes on Theoret. Comput. Sci.* **264**, 1 (2010) 91–105. ⇒ 63, 67, 70
- [13] I. Lazăr, S. Motogna, B. Pârv, Realizing use cases for full code generation in the context of fUML, *Proc. MDA & MTDD International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development*, Athens, Greece, 2010, pp. 80–89. ⇒ 63, 67, 68, 70
- [14] S. J. Mellor, M. J. Balcer, *Executable UML: a Foundation for Model-driven Architecture*, Addison-Wesley, MA, 2002. ⇒ 50
- [15] S. J. Mellor, *Agile MDA*, Technical Report, 2005. http://www.omg.org/mda/mda_files/AgileMDA.pdf ⇒ 50

-
- [16] S. Motogna, I. Lazăr, B. Pârv, I. G. Czibula, C. L. Lazăr, Component classification criteria for a platform-independent component repository, *Creative J. Math & Inf.* **17**, 3 (2008) 481–486. [⇒ 68, 69, 70](#)
 - [17] S. Motogna, B. Pârv, I. Lazăr, I. G. Czibula, C. L. Lazăr, Extension of an OCL-based executable UML components action language, *Studia Univ. Babeş-Bolyai, Inform.* **53**, 2 (2008) 15–26. [⇒ 58](#)
 - [18] S. Motogna, I. Lazăr, B. Pârv, I. G. Czibula, An agile MDA approach for service-oriented components, *Electronic Notes on Theoret. Comput. Sci.* **253**, 1 (2009) 95–110. [⇒ 62, 66, 67](#)
 - [19] Object Management Group, *UML 2.0 testing profile specification*, 2005. [⇒ 63](#)
 - [20] Object Management Group, *Model-level testing and debugging*, 2007. [⇒ 63](#)
 - [21] Object Management Group, *Semantics of a Foundational Subset for Executable UML models (FUML)*, 2008. [⇒ 58](#)
 - [22] Object Management Group, *UML superstructure specification*, Rev. 2.3, May 2010. [⇒ 50](#)
 - [23] OASIS, *SCA service component architecture. Assembly model specification*, Version 1.1., 2007. [⇒ 51](#)
 - [24] OASIS, *RIM registry information model*, 2007. [⇒ 70](#)
 - [25] OSGi Alliance, *OSGi service platform core specification*, Release 4, Version 4.1., 2007. [⇒ 49, 51, 57](#)
 - [26] B. Pârv, S. Motogna, I. Lazăr, I. G. Czibula, C. L. Lazăr, COMDEVALCO – A framework for software component definition, validation, and composition, *Studia Univ. Babeş-Bolyai, Inform.* **52**, 2 (2007) 59–68. [⇒ 51](#)
 - [27] B. Pârv, I. Lazăr, S. Motogna, COMDEVALCO framework – The modeling language for procedural paradigm, *Int. J. of Computers, Communication and Control* **3**, 2 (2008) 183–195. [⇒ 53](#)
 - [28] B. Pârv, I. Lazăr, S. Motogna, I. G. Czibula, C. L. Lazăr, COMDEVALCO framework: procedural and modular issues, *Proc. 2nd KEPT International Conference Knowledge Engineering: Principles and Techniques*, Cluj-Napoca, Romania, 2009, pp. 213–222. [⇒ 52](#)

-
- [29] B. Pârv, I. Lazăr, S. Motogna, I. G. Czibula, C. L. Lazăr, COMDEVAlCo Framework – Component modeling and validation issues, *Proc. 5th IC-TAMI International Conference on Theory and Applications of Mathematics and Informatics*, Alba-Iulia, Romania, 2009, pp. 83–101. ⇒ 52
 - [30] B. Pârv, I. Lazăr, S. Motogna, I. G. Czibula, C. L. Lazăr, COMDEVAlCo framework: Working with objects and components, *Int. J. of Computers, Communication and Control*, **6** (2011) (accepted). ⇒ 52
 - [31] Service oriented architecture modeling language (SOAML), <http://www.omg.org/spec/SoaML/> OMG, 2009. ⇒ 72
 - [32] C. Szyperski, D. Gruntz, S. Murer, *Component Software. Beyond Object-Oriented Programming*, 2nd edition, Addison-Wesley, 2002. ⇒ 48

Received: January 27, 2011 • Revised: February 23, 2011