Int. J. of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844
Vol. III (2008), No. 2, pp. 183-195

COMDEVALCO framework - the modeling language for procedural paradigm

Bazil Pârv, Ioan Lazăr, Simona Motogna

Abstract: This work is part of a series referring to COMDEVALCO - a framework for Software Component Definition, Validation, and Composition. Its constituents are: a modeling language, a component repository and a set of tools. This is the first paper describing the object-oriented modeling language, which contains fine-grained constructions, aimed to give a precise description of software components. The current status of the language reflects the procedural paradigm constructs.
Keywords: Software/Program Verification, Validation, Modeling methodologies, Computer-aided software engineering, Flow charts, Object-oriented design methods, Testing tools.

1 Introduction

Two main forces drive the software development today: complexity and change. Software development community looks for methods and practices to deal with these challenges.

Complexity in software development [6] is the same as in mathematics, dealing with problem-solving. The usual way of solving it is its reduction by reorganization. Brooks [2] makes a distinction between essential and accidental complexity. Essential complexity belongs to the problem to be solved and cannot be reduced or eliminated. Accidental complexity belongs to its solution, being created when fine-grained, general-purpose abstractions are used to directly implement coarse-grained, problem-specific concepts. It can be reduced or even eliminated by increasing the level of abstraction, i.e. by using more coarse-grained, problem-specific concepts (classes).

The other challenge in software development is *change management*: how to build software systems able to change. Software needs to change in response to changes in its operational environment and in its requirements. Sources of change are both in the problem domain (invalidating existing requirements, adding new ones) and in the solution domain, due to technological evolution.

Both challenges can be addressed in a disciplined manner using *models*, which increase the level of abstraction and allow for the automation of the software development process. Model-driven development (MDD) processes propose the creation of extensive models before the source code is written. An example of MDD approach is the Object Management Group's Model Driven Architecture [12] standard. Agile versions of MDD approaches have been defined in order to better deal with change management. Agile MDD processes [1] create *agile models* (models "just barely good enough") that drive the overall development efforts, instead of creating extensive models before writing source code. Another agile approach, Agile MDA [8], emphasizes on complete *executable models* [9].

Our work follows the idea of executable models, proposing an object-based modeling language that fits the procedural paradigm and allows construction and execution of models. Software components considered in our approach to the procedural paradigm are *Program* (the only executable), *Procedure*, and *Function*, and the software development process is component-based and model-driven. The modeling language constructs allow both precise *definition* of these components (called *program units*), and their *verification and validation* (V & V), by *simulating* their execution. Once these components pass the V & V process, they can be stored in a component repository, ready for later use in the development process.

The structure of this paper is as follows. After this introductory section, the next one is discussing current status, problems, and ideal properties of a modeling language. The third section presents the proposed modeling language, starting with low-level constructs, followed by statements, and finally

program units. The current status of our work is described in the fourth section, while the last one contains some conclusions and plans further efforts.

2 Modeling process: current status, problems, and desired features

2.1 Modeling languages

It is generally recognized that the use of models raises the level of abstraction and favors the automation of the software development process. Unfortunately, as Greenfield and Short stated in [6], the largest impediment to achieve these tasks was "the promulgation of imprecise general-purpose modeling languages as both de facto and de jure standards", namely UML.

Martin Fowler [5] identifies three different UML goals: *informal design* (sketching), *model-driven development* (blueprinting), and *direct execution* (model interpretation), noticing a similar and independent opinion made by Steve Mellor [8]. The conclusion is that UML succeeded in the first goal and failed in the others; the reasons for this failure, as they are discussed in [6], are: (1) lack of precision, (2) poor support for component-based development, and (3) weak extensibility mechanisms.

UML 2 and its Action Semantics [15] provide the foundation to construct executable models, but the standardization efforts for defining a subset of actions sufficient for computational completeness are still in progress [14]. In order to make UML a computational-complete specification language, there are some tools [3, 17, 18, 11] which have defined *non-standard* subsets of actions. Other issues related to UML 2 refer to the graphical notations and textual notations. The current version of UML does not define graphical notations for easy manipulation of UML elements. Moreover, there are UML elements (e.g. UML structured activity nodes) without a proposed graphical notation, and textual notations for behavioral elements are still in the process of standardization [14].

2.2 Component-based development and the modeling process

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 concerning the 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 [4] 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.

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.

The design process of a component-based system [7] 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.

2.3 Ideal properties of a modeling language

Our discussion here follows the general uses of a modeling language identified by Martin Fowler [5] and Steve Mellor [8]: *informal design*, *model-driven development*, and *direct execution*. In order to fulfill these goals, a modeling language should have: (1) good degree of precision, (2) good support for CBD, (3) good support for Agile MDA processes, and (4) good extensibility mechanisms.

In order to be precise, a modeling language needs to have fine-grained constructs, which allow both the complete definition of computing processes and the simulation of their execution. The language elements should cover low-level constructs referring to data types, expressions, program state and behavior (body of statements).

As we stated above, in order to offer a good support for CBD, a modeling language needs to build general, flexible and extensible models for both components and software systems. The resulting models should be both language-independent and programming-paradigm independent.

In order to offer a good support for agile MDA processes, a modeling language should provide *a metamodel*, together with *graphical* and *textual notations* for easy manipulation of language constructs. The metamodel should satisfy two important properties: *computability* and *completeness*.

The extensibility of a modeling language means the extensibility of its set of constructs, like data types, expressions, statements, and program units.

3 A modeling language proposal: COMDEVALCO solution

This section discusses in more detail our proposal of a modeling language, part of a framework for component definition, validation, and composition.

The proposed solution is COMDEVALCO - a conceptual framework for Software COMponents DEfinition, VALidation, and COmposition. Its constituents are meant to cover both CBD-related sub-processes described in 2.2: component development and component-based system development.

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 the modeling language. Once defined, component models are passed to a V & V (verification and validation) process, which is intended to check their correctness and to evaluate their performances. When a component passes the 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 as a compound object, using the elements of the object-based modeling language. 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.

Starting in a top-down manner, program units considered are *Program* (the only executable) and proper software components specific to the procedural paradigm - *Procedure* and *Function* (see Figure 1). Each of these software components has a *name*, a *state*, and a *body*; the *state* is given by all *Variable*s local to the component, and the *body* is, generally speaking, a *Statement*.

According to the imperative paradigm, the program execution is seen as a set of state changes, i.e. the execution of a statement changes the *Value* of a *Variable*, usually by evaluating an *Expression*.

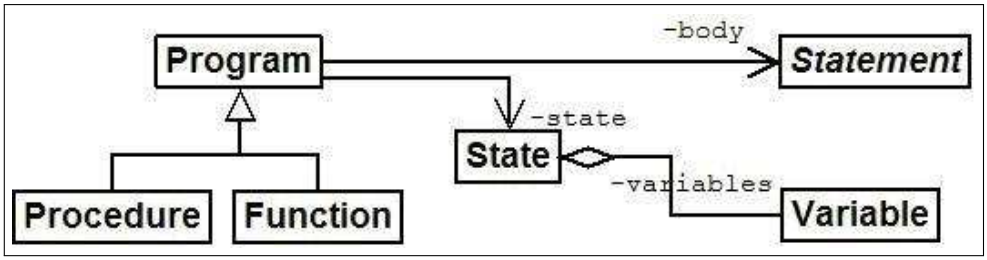

Figure 1: COMDEVALCO modeling language. Main constructs in the procedural paradigm

We describe below, in a bottom-up manner, the language elements as they are considered so far.

3.1 Low-level constructs

Basic language constructs are *Type* and *Declaration*. *Type* class abstracts the concept of data type, while a *Declaration* object is used to associate a specific *Type* object to a name (identifier). This corresponds to explicit variable declaration in imperative programming languages.

The next important concept is *Value*. Each *Value* object encapsulates a value of a specific *Type*. Values are fundamental in our model, because a variable represents an alternate name for a value stored in the memory, a function returns a value, or, more generally, the process of evaluating an expression returns a value.

Having these facts in mind, we designed the *Expression* class hierarchy shown in Figure 2. The root of the hierarchy, *Expression*, is abstract and has a single abstract operation, *getValue*(), overriden by subclasses and returning a *Value* result. The concrete specializations of *Expression* are: *Value*, *Variable*, *BinaryExpression*, *UnaryExpression*, and *DefinedFunction*.

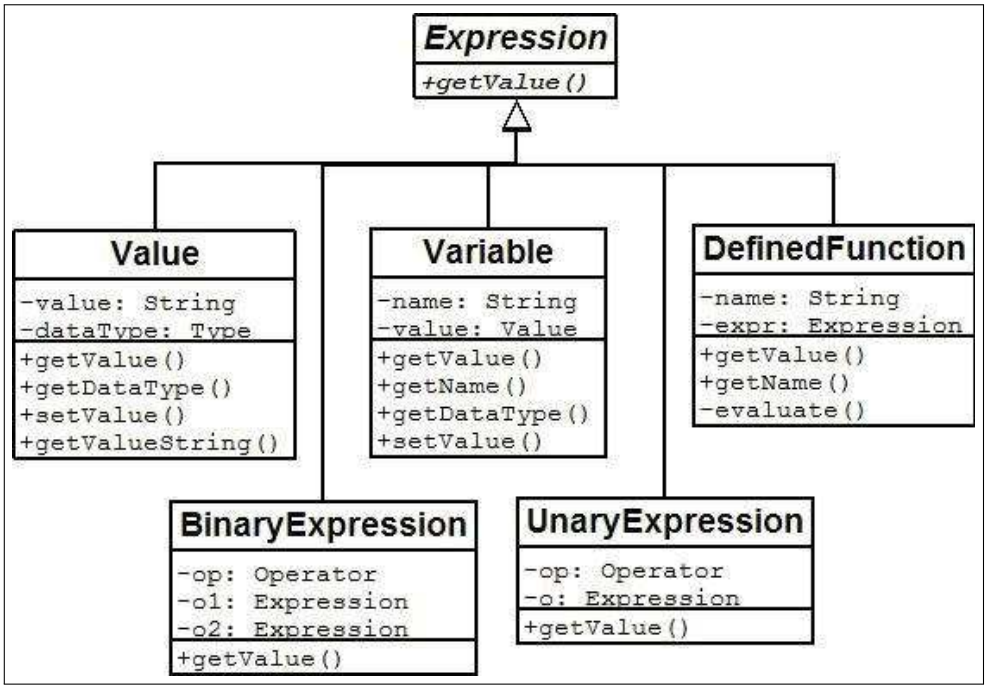

Figure 2: *Expression* class hierachy

Value is the simplest *Expression* descendant, its instances corresponding to typed literals (constants). Its proper operations are: *getDataType*(), which returns the data type of the value stored in the object,

getValueString(), returning the value as a string, and *setValue*().

Variable is probably one of the most important subclasses of *Expression*, having multiple uses in our model: (1) a *Variable* is a simple *Expression*; (2) all *Statement* objects deal with *Variable* instances, and (3) the state of a program unit is seen as a collection of *Variable* objects. According to the general definition, each *Variable* object has a name (identifier) and a value (a *Value* object). Its own operations are: *getName*(), *getDataType*(), and *setValue*().

Specific expression classes considered so far are *BinaryExpression* and *UnaryExpression*, having an operator and two, respectively one expression operands. The extensible *Operator* class implements *evaluate*() operations for all operand types, called by *getValue*() code in *BinaryExpression* and *UnaryExpression*.

DefinedFunction object corresponds to a one-argument function call. Its instance variables are the name of the function and its actual parameter, an expression.

3.2 Statements

Statement class hierarchy employs *Composite* design pattern, with subclasses *SimpleStatement* and *CompoundStatement*. A *CompoundStatement* object contains a list of *Statement* objects; both concrete simple and compound statement objects are treated uniformly.

As Figure 3 shows, *Statement* class is abstract and represents the root of all simple and compound statement classes. Its single abstract operation is *execute*(), which usually produces a state change. *Statement*'s concrete subclasses will implement this operation accordingly.

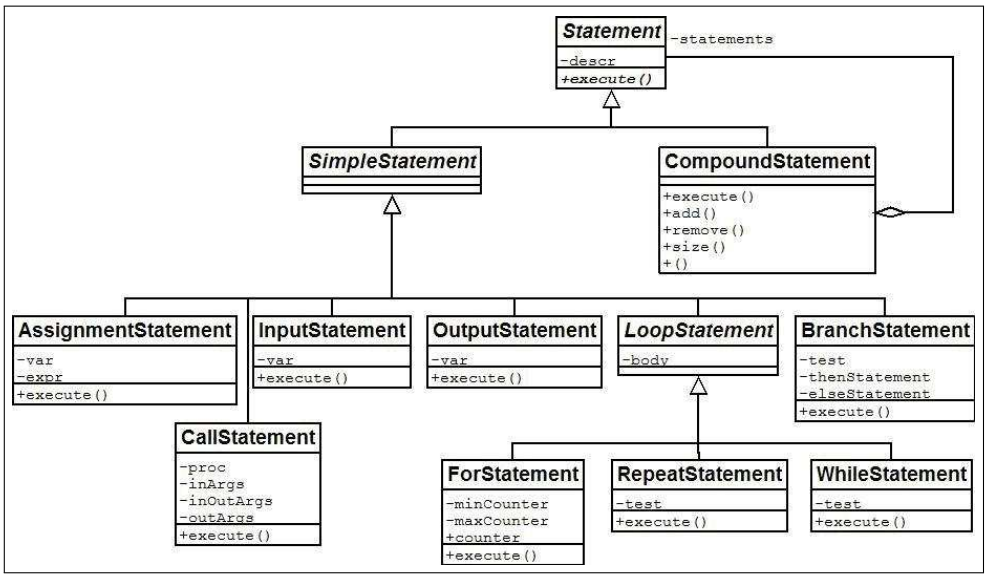

Figure 3: *Statement* class hierachy

SimpleStatement subclasses cover all important control statements in any imperative programming language: *AssignmentStatement*, *CallStatement*, *InputStatement*, *OutputStatement*, *LoopStatement*, and *BranchStatement*.

AssignmentStatement object takes an *Expression* and a *Variable*; after its execution, the variable takes the value of that expression. *InputStatement* takes only a *Variable*, being considered as a special kind of assignment: its execution means reading a value from standard input, evaluating and assigning it to the considered variable. The execution of an *OutputStatement* extracts the value of its *Variable* to standard output.

CallStatement object corresponds to a procedure call. Its instance variables are the procedure being

called (the callee object) and actual (call) parameters, whose values belong to the caller object. According to the definition of a *Procedure* object (see next subsection), three different parameter lists are needed, corresponding to in-, in-out, and out parameters. The execution of this statement has five steps: (1) extracting the values of in- and in-out parameters from the caller state; (2) building the callee state; (3) running the callee object; (4) extracting the values of in-out and out parameters from the callee state and (5) updating the state of the caller object with these values.

All loop statements execute repeatedly their *body*, a *Statement* object. Three different loop statements were considered in our design, considered as subclasses of the abstract *LoopStatement*: *ForStatement*, *RepeatStatement*, and *WhileStatement*.

In the case of a *ForStatement* object, the number of iterations is known a priori, and its execution uses a counter *Variable*, with values ranging from lower to upper bounds. *WhileStatement* and *RepeatStatement* objects use a test *Expression* to continue the iterative process. Their execution differs by the test strategy, i.e. evaluate test then execute body (*while*), respectively execute body then evaluate test (*repeat*).

BranchStatement objects correspond to *if-then-else* constructs. The condition to be checked is an *Expression* object, and both branches are *Statement* objects. Its execution evaluates test expression and then, based on its value, executes the corresponding statement.

3.3 Program units

As we already stated, the program units considered so far are *Program*, *Procedure*, and *Function* (see Figure 4), specific to the procedural paradigm.

Program is the only executable software component, having a *name*, a *state*, and a *body*; the *state* is made up by all *Variable*s local to the component, and the *body* is a *Statement* object. The only operation of a *Program* object is its *run*() method, implemented by a call to the *execute*() method of its *body*.

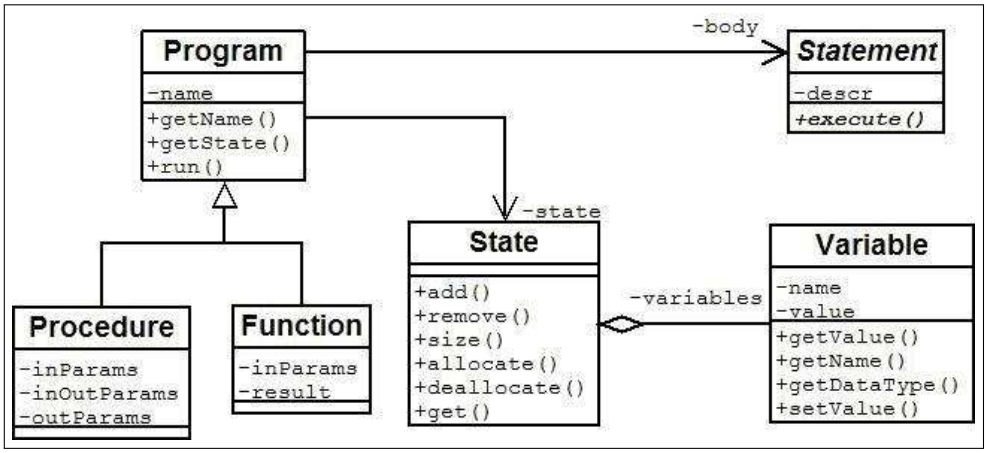

Figure 4: Program units class hierachy

Proper software components are *Procedure* and *Function*. As in the imperative programming world, a *Procedure* declaration must define its name, formal parameters, local state and procedure body. Consequently, it is easy to consider *Procedure* as a *Program* subclass; the proper instance variables of the *Procedure* class are the lists of in-, in-out, and out parameters, needed for a complete implementation of *execute*() method from *CallStatement* class, discussed in 3.2.

The concept of a user-defined function in imperative programming languages considers it without side-effects, i.e. the only result of its execution is the value it produces, without affecting the caller's state. Having this in mind, we designed the *Function* class accordingly, i.e. it has only input (*in*) formal parameters, and produces as result a *Value* object.

4 Experimental results

From a methodological viewpoint, our main concern was to model all theoretical aspects in concrete objects - constructs of the modeling language. The idea was to apply an iterative and incremental process: start with simple objects, perform checks after each modeling step, in order to be sure that things work.

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

The initial step of our work was to prove that things are working well. So we conducted first a proof-of-concept study, and then we started the development of graphical tools.

4.1 Modeling language: proof-of-concept stage

The modeling language described in Section 3 was implemented in Java. The classes containing the current implementation are included in three packages: `syntax` (`Expression` classes, `Declaration`, `DeclarationList`, `Operator`, `State`, and `Type`), `statements` (all classes in Figure 3), and `programUnits` (`Program`, `Procedure`, and `Function`). The implementations were tested by building some simple components, like solving of polynomial equations of first and second degree, or computing the integer square root (`isqrt()`) of a positive integer.

As we discuss below, in order to test a proper component `P`, two program units need to be designed: the component `P` itself and the corresponding test driver (a `Program` component). In each situation, the building process has two main steps: (1) build the state of the component and (2) build its body. As the state is a set of `Variable` objects, its creation is a sequence of `allocate()` messages. Additionally, in the case of `Procedure` and `Function` components, the parameter lists need to be also defined, in the same way. Next, the body of a component is a `CompoundStatement`, so the building process needs to create all the `Statement` objects which describe the computing process, and to include them into the body, preserving the sequence of computing steps.

Consider the simplest example of designing a component `EcGr1` which solves polynomial equations of the first degree, and its corresponding test program. The building process of `EcGr1` is defined in the following static method:

```
public static Procedure buildEcGr1() {
  DeclarationList inP = new DeclarationList();     // in params
  DeclarationList outP = new DeclarationList();    // out params
  DeclarationList inOutP = new DeclarationList();  // in-out params
  DeclarationList locale = new DeclarationList();  // local state
  // input parameters
  inP.allocate("a", Value.tDouble);
  inP.allocate("b", Value.tDouble);
  // output parameters
  outP.allocate("cod", Value.tInt);
  outP.allocate("x", Value.tDouble);
  CompoundStatement body =
      new CompoundStatement("Solves the equation a x + b = 0");
  Procedure proc =
      new Procedure("proc EcGr1", locale, body, inP, outP, inOutP);
  // body: BranchStatement(s1, s2)  (a == 0)
  Statement s11 =
      new AssignmentStatement("cod = 2", // infinite solution set
                              proc.getLocalState().get("cod"),
                              new Value(Value.tInt, "2"));
  Statement s12 =
      new AssignmentStatement("cod = 1", // no solution
```

```
                                    proc.getLocalState().get("cod"),
                                    new Value(Value.tInt, "1"));
    Expression e1 = proc.getLocalState().get("b");
    Expression e2 = new Value(Value.tDouble, "0");
    Expression e =
        new BinaryExpression(e1, e2, new Operator(Operator.EQ));
    // s1: BranchStatement(s11, s12)  (b == 0)
    BranchStatement s1 = new BranchStatement("b == 0", e, s11, s12);
    CompoundStatement s2 = new CompoundStatement("unique solution");
    s2.add(new AssignmentStatement("cod = 0",
        proc.getLocalState().get("cod"),  new Value(Value.tInt, "0")));
    e1 = proc.getLocalState().get("b");
    e2 = new Value(Value.tInt, "-1");
    Expression e3 = new BinaryExpression(e1, e2,
                                new Operator(Operator.TIMES)); // -b
    e2 = proc.getLocalState().get("a");
    e = new BinaryExpression(e3, e2, new Operator(Operator.DIV));  // -b/a
    s2.add(new AssignmentStatement("x = -b/a",
                proc.getLocalState().get("x"), e));
    e1 = proc.getLocalState().get("a");
    e2 = new Value(Value.tInt, "0");
    e = new BinaryExpression(e1, e2, new Operator(Operator.EQ));
    body.add(new BranchStatement("a == 0", e, s1, s2));
    return proc;
  }
```

The method builds a `Procedure` object who implements the well-known algorithm for solving first degree polynomial equations (its body being a `BranchStatement` object) and returns it when the building process is done. This object has two *in* parameters (`a` and `b`) and two *out* parameters (`x` and `cod`).

The test driver `Program` object is built as follows:

```
  public static Program buildProgEcGr1() {
    DeclarationList state = new DeclarationList();
    CompoundStatement body =
            new CompoundStatement("Test driver for EcGr1");
    Program prog = new Program("DemoEcGr1", state, body);
    state.allocate("ca", Value.tDouble);  // coefficient a
    state.allocate("cb", Value.tDouble);  // coefficient b
    state.allocate("rez", Value.tInt);    // return code
    state.allocate("sol", Value.tDouble); // the solution
    // resets state
    prog.setState(state);
    // start program
    body.add(new OutputStatement("*** Program " +
              prog.getName() + " started ***", null));
    // read coeffs
    body.add(new InputStatement("Read coeff ca", prog.getState().get("ca")));
    body.add(new InputStatement("Read coeff cb", prog.getState().get("cb")));
    // calls EcGr1
    DeclarationList pIn = new DeclarationList();
    pIn.allocate(prog.getState().get("ca"));
    pIn.allocate(prog.getState().get("cb"));
    DeclarationList pOut = new DeclarationList();
    pOut.allocate(prog.getState().get("rez"));
```

```
    pOut.allocate(prog.getState().get("sol"));
    DeclarationList pInOut = new DeclarationList();
    Procedure ecGr1 = buildEcGr1(); // create the EcGr1 procedure object
    SimpleStatement s = new CallStatement("call EcGr1", ecGr1, pIn, pInOut, pOut);
    body.add(s);
    // print results
    Statement s11 = new OutputStatement("unique solution", null);
    Statement s12 = new OutputStatement("print solution",
                                    prog.getState().get("sol"));
    CompoundStatement s1 = new CompoundStatement("Print unique solution");
    s1.add(s11);
    s1.add(s12);
    Expression e1 = prog.getState().get("rez");
    Expression e2 = new Value(Value.tInt, "1");
    Expression e = new BinaryExpression(e1, e2, new Operator(Operator.EQ));
    s11 = new OutputStatement("empty solution set", null);
                                          // cod = 1 (a=0, b<>0)
    s12 = new OutputStatement("infinite solution set", null);
                                          // cod = 2 (a=0, b=0)
    BranchStatement s2 = new BranchStatement("rez == 1", e, s11, s12);
    e2 = new Value(Value.tInt, "0");
    e = new BinaryExpression(e1, e2, new Operator(Operator.EQ));
    BranchStatement st = new BranchStatement("rez == 0", e, s1, s2);
    body.add(st);
    body.add(new OutputStatement("*** Program " + prog.getName() +
                                 " terminated ***", null));
    return prog;
  }
```

This time, the body of the constructed `Program` object is a sequence of statements which: (1) read the coefficients (using a sequence of `InputStatement` objects), (2) call the `EcGr1` component (using a `CallStatement` object), and (3) print the result (using `BranchStatement` objects in which the condition tests the value of the *out* parameter `rez` while the branches are `OutputStatement` objects).

The above code also contains statements which create and build the state of the `Program` object, and prepare the call process, i.e. create and populate the actual *in*, *out*, and *in-out* parameter lists and then create the callee `Procedure` object `EcGr1` (by invoking `buildEcGr1()` method).

The `main()` method of the demo class first creates the driver `Program` object and then calls its `run()` method:

```
  public static void main(String[] args) {
    TextIO.putln("Demo Program Units");
    Program pEcGr1 = buildProgEcGr1();
    pEcGr1.run();
  }
```

The program object runs its body by executing sequentially the statements in it: (1) asks the user to enter the values of coefficients, (2) calls the `EcGr1` procedure and (3) prints an explanatory message and the unique solution (if this is the case).

Above-described component definition approach is tedious and error-prone. For example, in order to build a `BranchStatement` object, the process is bottom-up: (1) create the `Statement` objects corresponding to the branches and (2) create the `BranchStatement` object containing them.

In a real-world situation, the building process is assisted by graphical tools, as we discuss below. These tools will perform at least the following: (1) graphical or textual building of components, (2)

saving and restoring component definitions to/from a component repository, (3) component testing and debugging.

4.2 The toolset

The COMDEVALCO toolset proposal includes graphical tools for component definition, validation, and composition. This subsection describes current status of these tools.

As part of the COMDEVALCO framework, a procedural action language (PAL) was defined and it is described in [10]. PAL contains all statements included in Figure 3, has a concrete textual syntax for UML structured activities, and graphical notations for some UML structured activity actions. The main idea for simplifying the construction of UML structured activities is to use the pull data flow for expression trees. Also, we propose new graphical notations for conditionals and loops, following the classical flowchart style.

In order to allow the exchange of executable models with other tools, a UML profile is also defined, specifying the mapping between PAL and UML constructs.

A component definition and validation tool is under development, using both graphical and textual PAL notations for building *Program*, *Procedure*, and *Function* program units. This tool is used within an Agile MDA process which includes test-first component design steps: (1) add a test (in the form of a *Program* component calling a non-existing *Procedure* or *Function*), (2) run the tests (in order to report the missing components), (3) add the production code (i.e. design the missing components), and (4) goto step (2). The process ends during the step (2), when all the tests pass. In the steps (1) and (3), developers are allowed to use either the graphical notation or the concrete syntax of PAL; the tool maintains automatically the consistency of the two views. The debugging and testing techniques employed in step (2) are defined according to Model-level Testing and Debugging Specification [14, 13].

A detailed description of this tool will be given in a separate paper.

4.3 Original elements 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* and *Procedure*;
- the models are executable and verifiable because each component can be executed; moreover, one can use tools for validation and evaluation of complexity;
- the models are independent of any specific object-oriented language;
- the modeling language is flexible and extensible in the following dimensions: the statement set, the component (program units) family, the component definition, the data type definition, and the set of components;
- the modeling language allows the use of graphical tools in all the phases: building, validating, and using software component models;
- the 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.

5 Conclusions and further work

This paper describes the current status of the modeling language, part of the COMDEVALCO framework. As we discussed above, this version implements a minimal set of elements, corresponding to the procedural programming paradigm.

The approach considered was aimed to control the complexity of the problem and of the development process. We started with the simplest programming paradigm, using simple data types and expressions and a small but complete set of statement objects. The development process consisted of small steps, meaning either the implementation of a new concept (transforming the concept into an object), or the extension of a model element. As the experiments were successful, we believe that our approach is feasible.

Future developments of the modeling language will include: extending *Type*, *Expression*, and *Operator* classes in order to define and manage structured and object types, extending the program units with constructs specific to modular, object-oriented, and component-based paradigms. These steps are considered within the planned evolution of the COMDEVALCO framework, which include steps for defining the structure of component repository and developing the tools aimed to operate in the component definition, validation, evaluation, simulation, and composition.

Acknowledgements

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

Bibliography

[1] S.W. Ambler, *Agile Model Driven Development (AMDD): The Key to Scaling Agile Software Development*, http://www.agilemodeling.com/essays/amdd.htm.

[2] F.P. Brooks, No silver bullet: essence and accidents in software engineering, *IEEE Computer*, April 1987, pp. 10-19.

[3] K. Carter, *The Action Specification Language Reference Manual*, 2002. http://www.kc.com/

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

[5] M. Fowler, *UmlMode*, May 2003, http://martinfowler.com/bliki/UmlMode.html

[6] J. Grenfield, K. Short, *Software factories: assembling applications with patterns, models, frameworks, and tools*, Wiley, 2004.

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

[8] S.J. Mellor, *Agile MDA*, 2005. http://www.omg.org/mda/mda_files/AgileMDA.pdf

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

[10] I. Lazăr, B. Pârv, S. Motogna, I.G. Czibula, C.L. Lazăr, An Agile MDA Approach for Executable UML Activities, *Studia UBB, Informatica*, LII, No. 2, 2007, pp. 101-114.

[11] P.A. Muller et al, On executable meta-languages applied to model transformations, *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, 2005.

[12] Object Management Group, *MDA Guide Version 1.0.1*, 2003. http://www.omg.org/docs/omg/03-06-01.pdf

[13] Object Management Group. *UML 2.0 Testing Profile Specification*, 2005, http://www.omg.org/cgi-bin/apps/doc?formal/05-07-07.pdf.

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

[15] Object Management Group, *UML 2.1.1 Superstructure Specification*, 2007, http://www.omg.org/cgi-bin/doc?ptc/07-02-03/

[16] B. Pârv, S. Motogna, I. Lazăr, I.G. Czibula, C.L. Lazăr, COMDEVALCO - a framework for software component definition, validation, and composition, *Studia UBB, Informatica*, LII, No. 2, 2007, pp. 59-68.

[17] ProjTech AL: Project Technology, Inc, *Object Action Language*, 2002.

[18] Telelogic AB, *UML 2.0 Action Semantics and Telelogic TAU/Architect and TAU/Developer Action Language*, Version 1.0, 2004.

Bazil Pârv, Ioan Lazăr, and Simona Motogna
Babeş-Bolyai University
Faculty of Mathematics and Computer Science
Department of Computer Science
1, M. Kogalniceanu, Cluj-Napoca 400084, Romania
E-mail: {bparv,ilazar,motogna}@cs.ubbcluj.ro

Received: November 20, 2007

Bazil Pârv is Professor at the Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca. 7 books and university courses, and more than 75 papers. His research topics cover: programming paradigms, component-based sofware development, mathematical modeling in experimental sciences, and computer algebra.

Dr. Ioan Lazăr is Lecturer at the Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca. He published 7 books and university courses, and more than 15 papers. His current research topics include: object-oriented analysis and design, modeling languages, and programming methodologies.

Simona Motogna is Associate Professor at the Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca. She obtained her Ph.D. in 2001, with the thesis *Formal specification of object-oriented languages*. Her topics of interest are: compilers, semantics, formal specification related to object oriented languages and component based programming.