STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

ON SIMPLIFYING THE CONSTRUCTION OF EXECUTABLE UML STRUCTURED ACTIVITIES

C.-L. LAZĂR AND I. LAZĂR

Abstract. UML, with its Action Semantics package, allows the user to create object-oriented executable models. Creating such models, however, is a very difficult task, because the UML primitives are too fine-grained and because UML has many variation points. This article proposes a computationally complete subset of the Action Semantics and raises the level at which the user works, from actions to statements and expressions. New graphical notations are also proposed, so that the resulting structured activity diagram is more intuitive and clear.

1. Introduction

The Action Semantics package from UML [9] gives the user the possibility to create executable models [11]. Before, the behavior of an operation, for instance, had to be specified using an opaque expression, which means platform dependent code. The package supports many features, so that it may be used in different domains, not just in tasks similar to programming. The actions are also very flexible: the structured control nodes, for instance, are more general than the corresponding statements found in the most used programming languages [5].

The UML activities support both a structured action model and a flow action model, each one being more suited for a specific modeling task than another. They are equivalent only for small examples, and, in general, the functionality written in one action model can be converted in the other form, though not easy. The two action models are not independent of each other, as the structured action model mainly addresses control, and still needs flow to pass data between actions.

Received by the editors: October 20, 2008.

1991 *Mathematics Subject Classification.* 68N15, 68N30.

1998 *CR Categories and Descriptors.* D.2.2 [**SOFTWARE ENGINEERING**]: Design Tools and Techniques – *Computer-aided software engineering, Flow charts, Object-oriented design methods*.

Key words and phrases. UML, Action Semantics, Structured Activities, Action Language.

The structured model fits better with a textual notation style, which is usually designed for well nested control, using variables to pass data between actions, instead of data flow. The textual notation is what most programmers are used to, so the structured action model would be more suited for programmers using UML.

1.1. **The Problem and Motivation.** It is very hard to use the Action Semantics package directly while trying to reproduce the functionality from a simple piece of code written in a programming language. This happens because the Action Semantics package supports too many features, which makes it hard to be learned, it has many variation points, which makes it hard to be used properly [13], and combining the actions inside an activity feels like working in an assembler language. Many implicit things from a programming language code have to be explicitly formulated in the UML model, which creates a need for a better tool support in this area.

Another problem with the Action Semantics is that no notations are given for many elements. In general, the graphical notation from Action is used, with different stereotypes. The graphical notations can be improved a lot, and this article proposes a new set of graphical notations. Also, textual notations may be used, but this is not covered here.

1.2. **The Solution.** In this article we choose a well defined subset of the UML action semantics, in order to represent the structured activities, as this is still in the process of standardization [7]. The subset must be computationally complete, and have a precise behavior (as opposed to the semantic variation points from UML, which are many).

New graphical notations are introduced, which help create a clear and simplified view of the structured activity. The expressions, for instance, will be presented as an aggregate to the user, not as distinct UML objects, even though, behind the scenes, the expressions are represented using the UML model.

This article is meant to expand the Procedural Action Language model, the UML profile and the graphical notations proposed in [10].

2. Action Semantics (subset)

A Procedural Action Language (PAL) model was presented in previous articles [12, 6, 10]. A UML profile was also defined, so that the PAL model can be exchanged among UML compliant tools. This PAL model is used as a target of what are the desired capabilities of the chosen subset of Action Semantics, with certain deviations.

The new model moves away from the *procedural* aspect, to *object – oriented*. The structured Activity will no longer be done for a standalone operation or program, but as the behavior of an Operation that is an owned operation of an UML Class.

2.1. **SequenceNode and StructuredActivityNode.** We choose to represent the main block of an activity and all the other blocks with SequenceNodes, and we follow as much as possible the structured programming model. If the *push* model (as described below) is used to represent the statements, then each group of actions corresponding to one statement from a programming language will be grouped inside a StructuredActivityNode. This is done in order to maintain a manageable model, because the number of actions will grow very fast.

The chosen structure of the Activity is like this: the Activity has one InitialNode that marks the beginning of the execution, one SequenceNode as the main node of the activity (the body of the operation), and an ActivityFinalNode that marks the finalization of the execution [3, 5]. One ControlFlow edge will go from the InitialNode to the main SequenceNode, and one ControlFlow edge will go from the SequenceNode to the ActivityFinalNode. The SequenceNode is a structured node, so it may contain other actions. Also, it will execute the actions in order, without a need for explicit ControlFlow edges.

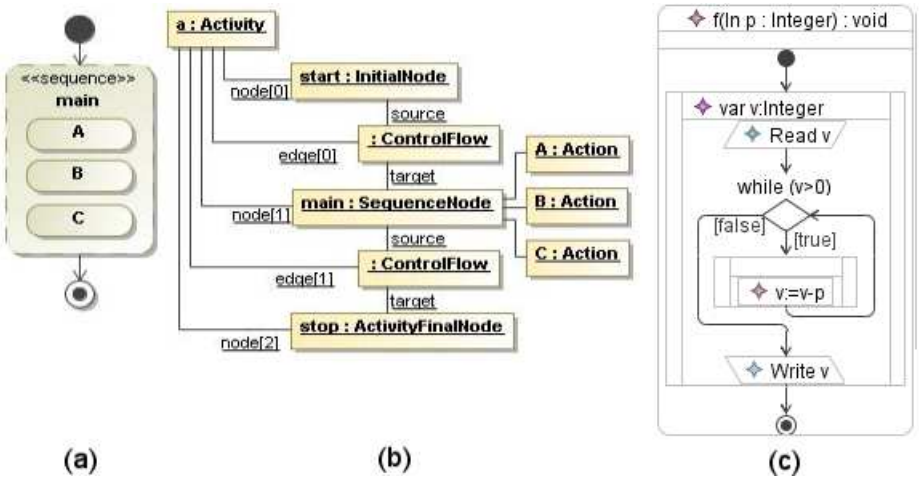

FIGURE 1. General Structure of an Activity

Figure 1 shows the general structure of an activity. Part (a) shows a possible representation inside an UML tool, part (b) shows the model structure of an activity, and part (c) presents the proposed representation of a simple

sample activity. The tools should automatically arrange the elements in the diagram, using a top-to-bottom layout for the statements, optionally showing the implicit control flow with arrows.

We propose to represent the blocks of statements with rectangles, with a double edge on the left and right sides, as shown in Figures 1, 2 and 5.

2.2. **Variables.** The SequenceNode has a set of Variables, that may be used for computations inside the node. The Action Semantics package provides convenient actions to access the values of the variables: AddVariableValueAction (to set a value to the variable), ReadVariableValueAction (to read the value from a variable), and others. The proposed representation for the used variables is shown in Figures 1, 2 and 5. Each block will present its set of variables at the top, in a distinct compartment.

2.3. **Parameters.** The Operation owns a set of Parameters, that describe the inputs and outputs of the operation. We choose that the Activity that represents the behavior of an Operation will always have a similar set of owned Parameters as the Operation. One operation may be invoked from the behavior of another operation by using CallOperationAction [2]. The tools may automate keeping the Parameters of the Operation in sync with the Parameters of the Activity.

There are no actions in UML to access the values of the parameters. Instead, the standards ask for ActivityParameterNodes [1, 4] to be used to provide the parameter input values when the activity starts and to output values to the parameters when the activity ends. One ActivityParameterNode will be created for each *in* and *inout* parameter, only with outgoing edges, and one ActivityParameterNode will be created for each *inout*, *out* and *return* parameter, only with incoming edges.

The input parameter nodes will receive control when the activity starts, at the same time as the InitialNode, and they will provide their parameter values to the outgoing edges. Because the parameter data object may flow over only one outgoing edge (the least resistant one), the usual approach would be to use an intermediate ForkNode [3] to copy the value to all the InputPins of the actions that require it.

The output parameter nodes will copy the values that reached them to the parameters when the activity ends, at the same time when the ActivityFinalNode is executed. The values reaching the parameter nodes will overwrite each other, so, at the end, only the last value that reaches the parameter node will be set to the parameter. Because an action cannot start executing unless all incoming edges provide a token, the usual approach is not to set the edges from the activity actions to go directly in the parameter node, but to merge them before they reach the node, using a MergeNode [3].

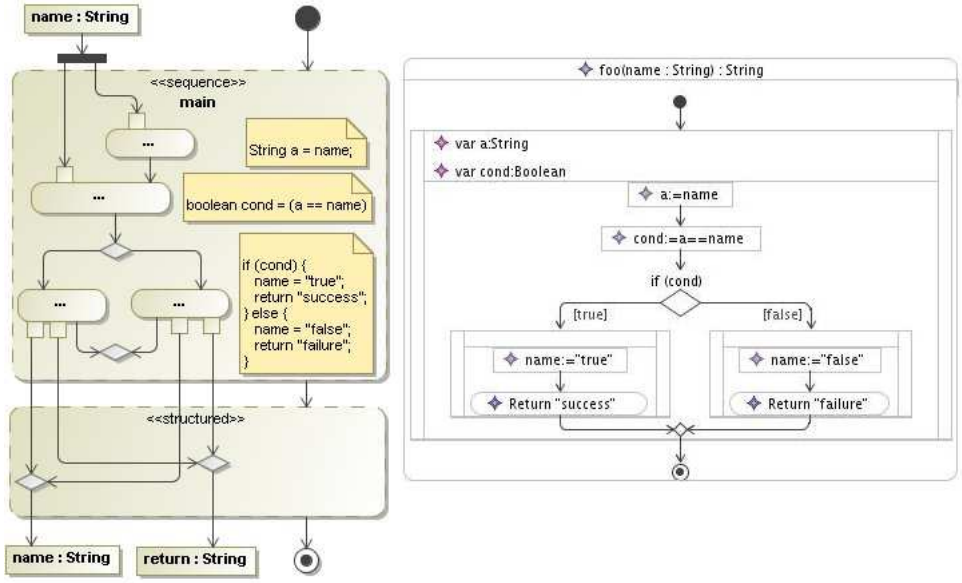

FIGURE 2. Activity with *inout* and *return* Parameters

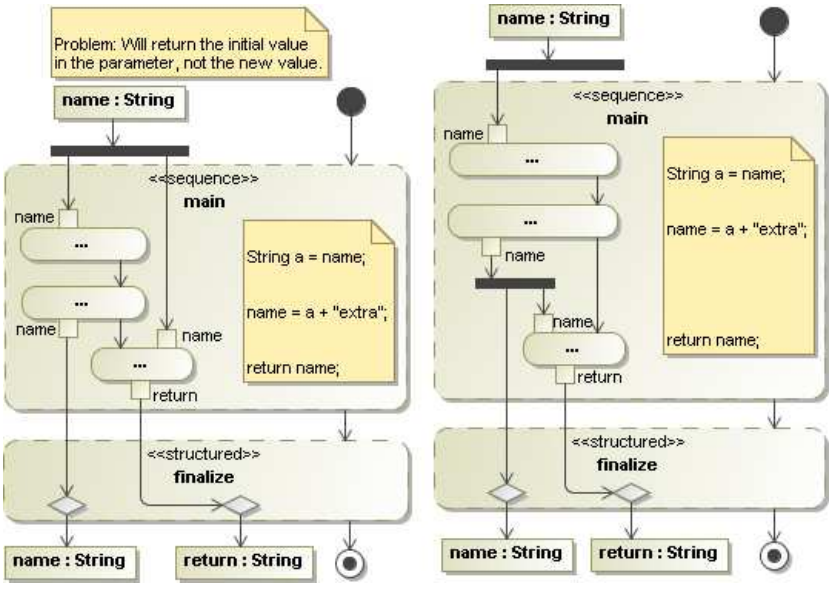

FIGURE 3. Problem (left) and Fix (right) for an Activity with an *inout* Parameter

Figure 2, on the left side, exemplifies the usual approach of working with parameters, for an *inout* and a *return* parameter. The figure uses the UML notations. The "..." actions represent an action or a group of actions that provide the functionality mentioned in the notes placed on the right side.

Using this approach of accessing the parameter values from the actions inside the activity has some problems:

- the model and diagram get very complicated when the functionality is bigger, or when there are many parameters, or if the parameters are accessed many times. The diagram may be fixed if the tools would not show the edges from the parameter nodes.
- the values that are intermediately set to an *inout* parameter during the execution cannot be read, if this scheme is used, as the subsequent actions using the parameter value will receive the initial parameter value from the input parameter node. This can be fixed by passing the intermediate values to the subsequent actions that use the parameter, but this will lead to complicated structures. This issue is presented in Figure 3.
- the *out* parameters cannot be built incrementally, as the stored values cannot be accessed. This can be solved with schemes similar to the one mentioned for *inout* parameters.

To solve these problems, we propose using an alternative approach, presented in Figure 4. For each parameter, except the *return* parameter, there should be a similar Variable (with the same name and type) at the Activity level, and the actions that want to access the parameters will access the corresponding variable instead.

An initialization StructuredActivityNode is introduced between the InitialNode and the main sequence node, having initialization actions:

- for each of the variables corresponding to the *in* and *inout* parameters there will be an AddVariableValueAction that will set the value received directly from the corresponding input ActivityParameterNode to the variable
- for each of the variables corresponding to the *out* parameters there will be an AddVariableValueAction that will set LiteralNull value to the variable.

The actions from the main sequence node that need to access the parameters will simply connect themselves to AddVariableValueActions, ReadVariableValueActions and ClearVariableValueActions configured with the proper variables.

A finalize StructuredActivityNode is introduced between the main sequence node and the ActivityFinalNode, having actions that will read the

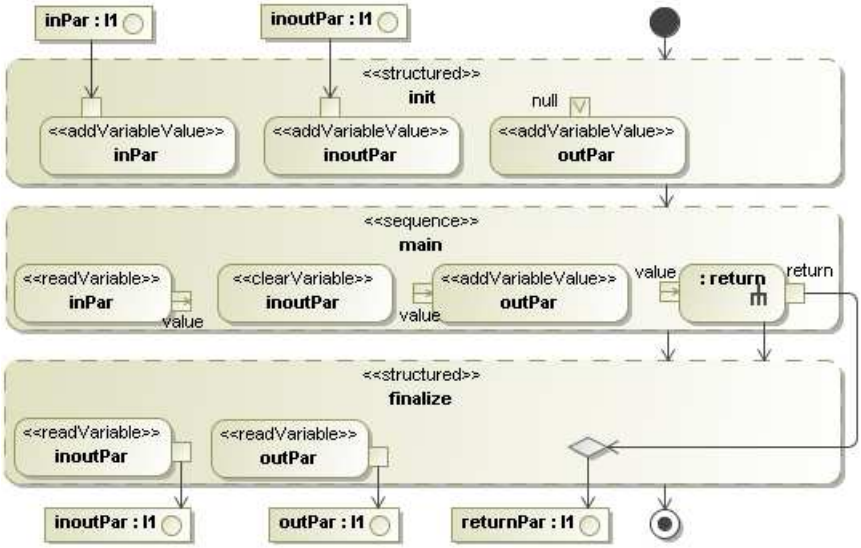

FIGURE 4. Proposed Structure of an Activity

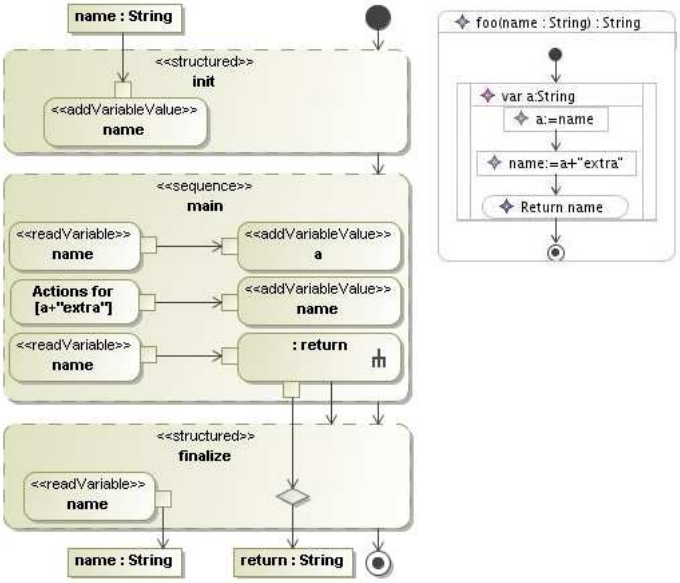

FIGURE 5. Solution and Representation for an Activity with *inout* Parameters

variable values from the *inout* and *out* variables and send them directly to the corresponding output ActivityParameterNodes.

The *return* parameter is handled using the usual approach, but this is described in a subsection below (Return / Output Statement).

In Figure 5 we present the example above with the *inout* parameter problem, solved with this approach (the structured nodes marking the statements are omitted, for brevity). The creation of the variables that correspond to the parameters, along with the *init* and *finalize* nodes containing the variables initialization / output actions, should be automated by the tools. The right side of the figure shows our proposed representation for the activity.

The parameters are presented graphically as part of the activity signature. A distinct compartment containing all the parameters may also be present at the activity level, similar to the compartment for the block variables.

2.4. **Model for Statements and Expressions.** The actions that form each statement may be composed using either the default *push* style model (data tokens will be pushed using ObjectFlow edges from OutputPins to InputPins), or, by using the *pull* style model (data tokens will be pulled by ActionInputPins from Actions with exactly one OutputPin) [5]. The expressions needed in conditions, for instance, are constructed in the same manner. The difference between a statement and an expression is that an expression provides an output value, which is used by a statement (for instance, the *test* node of the LoopNode is an expression that provides a boolean value).

In the *push* style model, the actions are all contained in the same node. The control will arrive at the actions with no input edges, and the data will be pushed through the actions, to the root action. This is not a very intuitive flow for the developers used to structured programming, but UML provides graphical notations and the UML tools have graphical support for it.

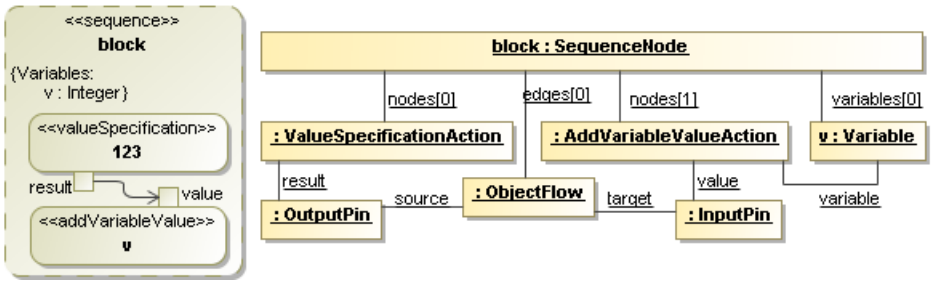

FIGURE 6. Assignment Statement (v:=123) With *push* Style Model

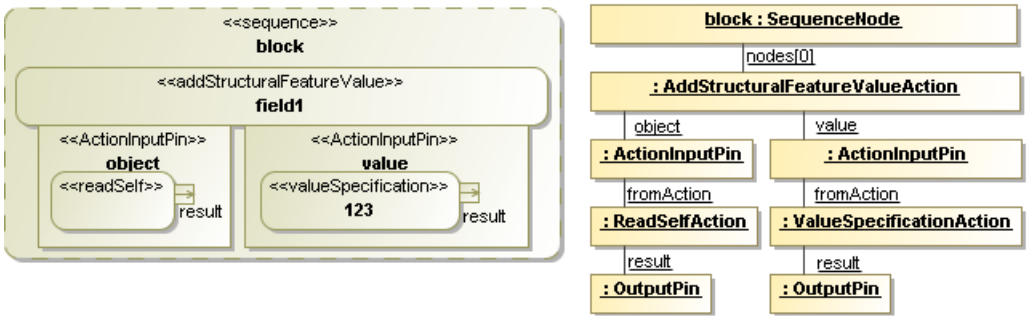

FIGURE 7. Assignment Statement (self.field1:=123) With *pull* Style Model
(Graphical representation on the left is not UML compliant!)

In the *pull* style model, the root action contains the action input pins, which, in turn, contain the *from* actions, and so on. The control will arrive at the root action, which will begin its execution by trying to get the data tokens from the ActionInputPins, which, in turn, will pull the data tokens from the contained actions, by executing them. The control will arrive in this way at the leaf actions. After the actions are executed and the data tokens are placed in the OutputPins, these data tokens are used as the values for the ActionInputPins. The problem with the *pull* style is that UML provides no graphical notations for the ActionInputPins, as these are meant to be used in textual representations. And this means that most of the UML tools do not have graphical support for the ActionInputPins.

UML provides a special kind of ActionInputPin, called ValuePin [4], that is a shorthand for an ActionInputPin providing the value from a ValueSpecificationAction. The ValuePin provides the value directly from a ValueSpecification. The UML tools might have graphical support for the ValuePin, though UML doesn't propose a graphical notation. However, using ValuePin only, in conjunction with InputPins, is insufficient for more complex statements.

The *pull* style model is chosen, as it fits better to our purpose, and it produces fewer objects, grouped in a well nested structure. However, in order to be able to exchange the models between UML tools, a conversion tool between the two styles is needed, so that a *pull* style model may be viewed and edited inside a UML compliant tool, as a *push* style model.

2.5. **Assignment / Input Statement.**

- The AssignmentStatement from the PAL model is represented with an action structure that has the root an AddVariableValueAction (if the statement assigns a value to a Variable), or an AddStructuralFeatureValueAction (if the statement assigns a value to a Property of a Classifier). The isReplaceAll boolean property of the action will be set to true. The proposed representation for this statement is shown in Figures 1, 2 and 5 (a simple rectangle containing the textual representation).
- The InputStatement from PAL is represented with the same actions, with the difference that the input value is obtained from a CallBehaviorAction using a FunctionBehavior called *read*, with one return parameter. The proposed representation for this statement is shown in Figure 1.

2.6. **Return / Output Statement.**

- The *return* parameter is handled using the usual approach (in a non-structured fashion), because the *return* parameter is set only once in an execution path, and after it is set, the execution of the activity has to end. A *return* action sends its result value to the MergeNode found in the $finalize$ StructuredActivityNode, which forwards it to the *return* ActivityParameterNode. Also, the *return* action gives the control to the $finalize$ node, so that the values from the variables are copied to the corresponding parameter nodes and forcing the execution of the activity to end. A CallBehaviorAction is used as the root action of the *return* statement, which means a special *return* FunctionBehavior needs to exist. This behavior should have one *in* parameter (the value to be returned) and one *return* parameter (the same value, that is returned). The action needs one output pin, in order to forward the value to be returned to the *return* ActivityParameterNode. The proposed representation for this statement is shown in Figures 2 and 5.
- The OutputStatement from PAL is represented in a similar fashion, by using a CallBehaviorAction as the root action of the statement. The used FunctionBehavior is called *write* and it has only an *in* parameter. The proposed representation for this statement is shown in Figure 1.

2.7. **Branch Statement.** The BranchStatement from PAL is represented with a ConditionalNode, with one Clause object if only *then* branch is present, or with two Clause objects if *else* branch is also present. The clauses will be

properly ordered by using their successor / predecessor properties. The ConditionalNode will contain all the *test* and *body* executable nodes, and the clauses will properly reference them as *test* or *body* nodes. The decider pin for a *test* clause will always be the output pin of its *test* node. The *else* clause will always have a *true* clause test, meaning that the test node will consist of one ValueSpecificationAction for the *true* LiteralBoolean.

The *body* node is a block of statements and is represented with a single SequenceNode, which will contain the actions for the statements.

The ConditionalNode is not *assured*, meaning that it is possible that no test will succeed (this is needed when the *else* clause is missing). And it is *determinate*, meaning that at most one test will succeed (this is needed when *else* clause is present, so that, if the test of *then* clause passes, the body of *else* clause will not be executed, as the test of *else* clause will always succeed).

An example for the proposed graphical representation is given in Figure 2, on the right side. If *else* branch is missing, there will be a control edge shown, with no statements, going to the merge node at the bottom.

2.8. **While / Do While / For Statement.** WhileStatement and ForStatement from PAL are represented with a tested first LoopNode. DoWhileStatement (a variant of RepeatStatement) is represented with a tested last LoopNode.

The LoopNode is a StructuredActivityNode, so it may have variables, which may be used as iterators for the ForStatement, as opposed to using the built-in system of loop input/output pins, which is hard to use. The LoopNode will contain all the actions for the *setupPart*, *test* and *bodyPart*, which will simply reference the used actions. The iterator may be initialized in the *setupPart* actions. The *test* actions will have to output a boolean value. The decider pin for the *test* will always be the output pin of its *test* actions. The *bodyPart* needs to contain both the actual body actions (inside a SequenceNode) and, if needed, the actions that update the iterator variables.

For the While and Do While statements, the iterator parts are omitted, and only the *test* and *bodyPart* (without the actions that update the iterator variables) will be present.

The loop node has a set of *setupPart* nodes, each one being represented in the model by actions corresponding to a single statement. ControlFlow edges will be set between the *setupPart* nodes, so that the statements are executed in order. The *bodyPart* node includes the main block of statements, which is represented with a single SequenceNode. This node is the first node (has no incoming ControlFlow edges) and will contain the actions for the statements. The statements that update the iterator variables are kept in the *bodyPart* node, also. A ControlFlow edge will go form the main block node to the first

iterator update statement, and the rest of the statements are ordered using ControlFlow edges, similar to the *setupPart* nodes.

The graphical representations are similar to those provided in [14], as they help the user understand the flow of the algorithm [15]. A sample for WhileStatement is provided in Figure 1 (c). The tools might support different layouts for the loop nodes, allowing the users to choose the preferred one. The layout used in [14] for ForStatement is chosen, as it does a good job in visually separating the four parts of the statement, while keeping the occupied space to a minimum and providing an intuitive flow.

2.9. **Extra Object related actions.**

- Reading *self* (or *this*) instance will be done using ReadSelfAction. This instance will need to be provided whenever the invoked operation or the accessed property is not static and no other instance is explicitly specified by the user.
- Creating a new object instance will be done using CreateObjectAction. This action will not invoke any operation, or behavior, so the created instance could be uninitialized. To obtain the *constructor* behavior found in programming languages, the tools could also execute, if needed, an operation that has the same name as the Classifier and one return parameter of the same Classifier type. The CallOperationAction becomes the root, obtaining its *target* input value from the CreateObjectAction. The CallOperationAction will provide the initialized object to the action that needs the instance, not the CreateObjectAction.

2.10. **Primitive Functions.** Similar to the other FunctionBehaviors mentioned before, a FunctionBehavior needs to be created for each primitive operation (==, +, -, ...) between Integer, Boolean and String typed operands, to be used in expressions. The *primitivefunctions* are limited, at this point, to having only operands of the data types defined in UML. All these *primitivefunctions* should be packaged in a separate model resource, so that they may be easily reused in different UML tools and different projects.

3. Conclusions and Future Work

Using SequenceNode (sequence), ConditionalNode (decision) and LoopNode (loop) from UML's CompleteStructuredActivities package, the chosen subset of actions is computationally complete.

The new level at which the user creates the executable models is raised from actions to statements and expressions, increasing user efficiency. The tools should take care of a lot of redundant steps while creating the model,

as well as properly arranging the diagram, allowing the user to focus on the actual algorithm.

The Action Semantics subset was chosen in such a way that the resulting models are as simple and clear as possible, while preserving the abstract syntax and the execution semantics of the UML elements. This has great benefits, as the resulting models are small and well structured, which makes it easier for an user to analyze them, if needed. It is also not that hard to create conformant models for small operations using existing UML tools.

There is no UML profile defined for the Action Semantics subset chosen in this article, which means that the executable models can be built without having to apply stereotypes. Instead, the article provides exact operational semantics for the selected elements, so that there is an exact interpretation of the model.

Formal OCL [8] constraints need to be defined, so that the UML models can be statically analyzed for conformance with the proposed action language, before being executed. In order to be conformant with this action language, the models must not contain other UML elements, except those proposed here, and they must also comply with the extra operational semantics defined in this article.

This article provides an exhaustive description for the core of an action language using UML Action Semantics. There are many elements remaining to be considered in the future: preconditions, postconditions, ...; re-analyze the support for arrays; switch statement; in-line if statement (with output value): $a > b \ ? \ a \ : \ b$; other non-structured statements: break, continue; exception handling; threads; synchronized blocks; operations for associations; events.

FunctionBehaviors for common utility operations may also be defined in the future, and packaged together with the primitive functions. Also, more data types could be defined, as the existing ones are far from being enough.

ACKNOWLEDGMENTS

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

References

[1] Conrad Bock. UML 2 activity and action models. *Journal of Object Technology*, 2(4):43–53, 2003.

[2] Conrad Bock. UML 2 activity and action models, part 2: Actions. *Journal of Object Technology*, 2(5):41–56, 2003.

[3] Conrad Bock. UML 2 activity and action models, part 3: Control nodes. *Journal of Object Technology*, 2(6):7–23, 2004.

[4] Conrad Bock. UML 2 activity and action models, part 4: Object nodes. *Journal of Object Technology*, 3(1):27–41, 2004.
[5] Conrad Bock. UML 2 activity and action models, part 6: Structured activities. *Journal of Object Technology*, 4(4):43–66, 2005.
[6] I.-G. Czibula, C.-L. Lazăr, I. Lazăr, S. Motogna, and B. Pârv. Comdevalco development tools for procedural paradigm. *Studia Univ. Babeş-Bolyai*, III, 2008.
[7] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models RFP*. http://www.omg.org/docs/ad/05-04-02.pdf, 2005.
[8] Object Management Group. *Object Constraint Language Specification, version 2.0*. http://www.omg.org/docs/formal/06-05-01.pdf, 2006.
[9] Object Management Group. *UML 2.1.2 Superstructure Specification*. http://www.omg.org/docs/formal/07-11-02.pdf, 2007.
[10] I. Lazăr, B. Pârv, S. Motogna, I.G. Czibula, and C.-L. Lazăr. An agile MDA approach for executable UML structured activities. *Studia Univ. Babeş-Bolyai*, LII(2):101–114, 2008.
[11] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley, 2002.
[12] Bazil Pârv. Comdevalco - a framework for software component definition, validation, and composition. *Studia Univ. Babeş-Bolyai*, LII(2):59–68, 2007.
[13] Tim Schattkowsky and Alexander Förster. On the pitfalls of UML 2 activity modeling. *International Workshop on Modeling in Software Engineering*, 2007.
[14] Tia Watts. *A Structured Flow Chart Editor*. http://watts.cs.sonoma.edu/SFC/.
[15] Tia Watts. The SFC editor a graphical tool for algorithm development. *Journal of Computing Sciences in Colleges*, 4(1):73–85, 2004.

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

E-mail address: ilazar@cs.ubbcluj.ro