Defining Object-Oriented Execution Semantics Using Graph Transformations

Harmen Kastenberg[⋆], Anneke Kleppe[⋆⋆], and Arend Rensink

University of Twente
Department of Computer Science
Enschede, The Netherlands
{h.kastenberg,rensink,kleppeag}@cs.utwente.nl

Abstract. In this paper we describe an application of the theory of graph transformations to the practise of language design. In particular, we have defined the static and dynamic semantics of a small but realistic object-oriented language (called TAAL) by mapping the language constructs to graphs (the static semantics) and modelling their effect by graph transformation rules (the dynamic semantics). This gives rise to execution models for all TAAL-programs, which can be used as the basis for formal verification.

This work constitutes a first step towards a method for defining all aspects of software languages, besides their concrete syntax, in a consistent and rigorous manner. Such a method facilitates the integration of formal correctness in the software development trajectory.

1 Introduction

A widely recognized proposal for combating the maintenance and evolution problems faced in software engineering is the model driven approach, brought to the world's attention by the OMG's Model Driven Architecture (MDA) framework [17]. In this approach, models and model transformations are central concepts. The models are specified in diverse (modeling and programming) software languages (SLs), and the model transformations define relations between these languages.

Model transformations are intended to be correctness preserving: they should not introduce errors or essential changes. This, however, can be guaranteed only if the meaning of the SLs involved is defined with sufficient precision. Unfortunately, this is often lacking: many SLs have a well-defined syntax but only an informal semantics, e.g. described by text or, in the case of a programming language, by a compiler.

The longer-term goal of our research is to define a way in which all aspects of SLs, besides their concrete syntax, can be defined in a consistent and rigorous

[⋆] The author is employed in the GROOVE project funded by the Dutch NWO (project number 612.000.314).

[⋆⋆] The author is employed in the GRASLAND project funded by the Dutch NWO (project number 612.063.408).

manner. As a common formal foundation we use graphs and graph transformations, which we believe to be powerful enough to capture all relevant SL aspects. Furthermore, current research in the field of graph transformations [23] offers us a large knowledge base of theories ready to apply to our topic. Ultimately, we plan to develop a meta-language for designing SLs. This meta-language will enable us to provide semantic definitions of the source and target SLs involved in a given model transformation on a compatible basis; this in turn will enable us to precisely formulate and check the requirement of correctness preservation. We believe these abilities to be essential in realizing the full potential of MDA.

This paper describes the first phase of our research: the formal definition of both the static and the dynamic semantics of a small but realistic object-oriented language, called TAAL, using graph transformations. We have defined our own language because in this way we can avoid dealing with more complex constructs like exception handling and multi-threading. Still TAAL includes common object-oriented features like inheritance. While formal, we do not leave this exercise on a theoretical level only: we have developed a parser/analyzer and used an existing graph transformation tool so as to actually simulate programs. In fact, all graphs shown in this paper are directly taken from the implementation. We are confident that we can extend the approach described here to be applicable to a large category of SLs, including modeling languages and imperative programming languages.

This paper is structured as follows. Sect. 2 gives an overview of our approach and introduces graph transformations and TAAL. In Sect. 3 we discuss how we represent and generate the flow of control of a TAAL-program. Sect. 4 then discusses our main contribution, namely our way of specifying object-oriented dynamic semantics through an operational definition. We conclude in Sect. 5 with a brief description of the tooling used and some remarks on related and further work. All steps described in this paper are explained by using a simple example.

2 Approach

In this work we model object-oriented programs as graphs, and specify their semantics using graph transformations. The approach we have taken is to define a small language that nonetheless contains the most relevant concepts from object-oriented programming languages. This language is called TAAL. We define a series of transformations that will turn any TAAL-program into a simulation of its execution.

The transformations are depicted in Fig. 1. The first transformation, from textual program to Flat Abstract Syntax Graph, actually consists of three transformations. Due to space limitations, we do not discuss the details of these transformations, two of which are similar to the first steps in a compiler [3]. The interested reader is referred to [13]. The more interesting transformations, i.e. the flow graph construction and the simulation, involve the application of

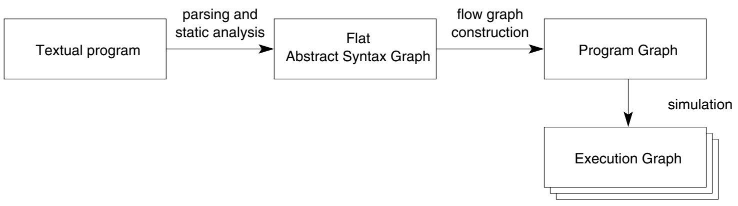

Fig. 1. Overview of the transformation from program to simulation

graph transformations and will therefore be discussed in more detail. The graph transformations are carried out in the Groove Tool Set [20].

During flow graph construction we apply a set of graph transformation rules to transform a plain graph representing the abstract syntax of the textual program, called the Flat Abstract Syntax Graph, into a graph that includes control flow information. The result of this transformation is called the Program Graph. The execution of the TAAL-program is simulated by Execution Graphs which are the result of applying another set of graph transformation rules. These rules define the dynamic (or execution) semantics of our object-oriented language TAAL.

Note that the Program Graph and the corresponding Execution Graphs are at a different level of modelling. This is reflected in Fig. 1 by the use of a vertical arrow instead of a horizontal one. The Program Graph is a single graph representing the static TAAL-program including control flow information, whereas during simulation the dynamics of the program execution are represented by a series of Execution Graphs, each of which represents the system state at a certain point in time.

2.1 The Formalism

After the parsing and static analysis phase, the textual program in Fig. 1 is represented as a plain graph (the Flat Abstract Syntax Graph), and the subsequent transformations are driven by sets of graph transformation rules. Such rules are themselves given as graphs. This will be shown later in this section.

In this paper we use edge-labelled graphs, defined over a global set *Lab* of labels, as follows.

Definition 1. *A* graph $G = \langle V, E \rangle$ *consists of:*
- *a set* V *of* vertices *(or* nodes*), and*
- *a set* $E \subseteq V \times Lab \times V$ *of* edges.

The following is a definition of a graph transformation rule.

Definition 2. *A* graph transformation rule $p = \langle L, R, \mathcal{N} \rangle$ *consists of:*

- *a graph* L *being the* left hand side *(LHS) of the rule;*
- *a graph* R *being the* right hand side *(RHS) of the rule;*
- *a set of graphs* $\mathcal{N}$ *being the* negative application conditions *(NACs).*

The application of a graph transformation rule transforms a graph G, the *source graph*, into a graph H, the *target graph*, by looking for an occurrence of L in G and then replacing that occurrence of L with R, resulting in H. The role of the NACs [10] is that they can still prevent application of the rule when an occurrence of the LHS has been found, namely if there is an occurrence of some $N \in \mathcal{N}$ in G that *extends* the candidate occurrence of L. A precise technical description of the search for occurrences and the transformation process is given in [22]; for a more theoretical exposition see [23].

It is important to realize that the application of a graph transformation rule to a given graph G is non-deterministic in that there may be more than occurrence of L in G; but for any particular occurrence, the application is deterministic.

Individual graph transformation rules are collected into *graph production systems* (GPSs), which as a whole are used to model transformation or computation processes. The application of a GPS comes down to the unscheduled, non-deterministic application of successive individual rules until a graph is obtained that cannot be transformed any further. Note that this introduces another level of non-determinism, namely in the choice of rule to be applied. A GPSs is *confluent* if it is such that the order of application actually does not make a difference to the end result.

In this paper we use two GPSs, namely to model the flow graph construction and simulation steps of Fig. 1. As we will see, the first of these is confluent whereas the second is deterministic due to the fact that at any stage during the transformation system, at most one rule is applicable, which then has precisely one occurrence.

When visualizing graphs and graph transformation rules, we use a shorthand notation for labelled edges pointing from one node to itself, so called *self-edges*. In this shorthand notation we put the label inside the node.

Example 1. Fig. 2 (i) depicts a graph transformation rule by showing its LHS and its RHS. The LHS consists of three nodes and five labelled edges; the RHS consists of three nodes and six labelled edges. Note that self-edges are also counted. The result of applying this transformation rule is the creation of a flowNext-edge and the redirection of the flowIn-edge. (To be precise, the flowIn-edge in the LHS graph will be removed and a new flowIn-edge will be created.)

In this paper we use a shorthand notation for graph transformation rules by displaying them as single graphs. The different roles a graph element can have in the transformation process are distinguished by different coloured shapes:

– thin solid nodes and edges, called *readers*: they are required to be in the source graph in order for the rule to apply, and are unaffected by rule application, i.e. they are still present in the target graph.
– thin dashed nodes and edges (blue in a coloured print-out), called *erasers*: they are required to be in the source graph in order for the rule to apply, and are deleted by rule application.
– fat solid nodes and edges (green), called *creators*: they are not required to be in the source graph, and are created by rule application.

– fat dashed nodes and edges (red), called *embargoes* (or negative application conditions): they are forbidden to occur in a graph in order for the rule to apply.

Fig. 2 (ii) shows the graph transformation rule from Fig. 2 (i) using the described shorthand notation. Note that this rule does not include negative application conditions.

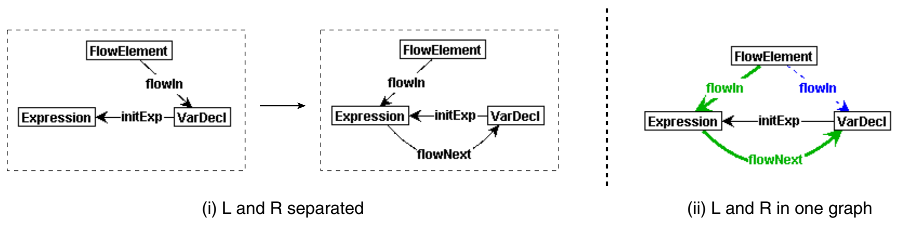

Fig. 2. Example of a graph transformation rule

When using graphs for representing *states* and graph transformations as *state transitions*, applying graph transformations is analogue to creating a *transition system* in which the transitions are labelled with the names of the transformation rules. In Sect. 4 we will show a labelled transition system generated this way representing the simulation of an example program.

2.2 The Mini Language TAAL

In this section we discuss the mini language TAAL, which incorporates the basic aspects of many commonly used object-oriented programming languages. For instance, the notions of class, attribute, operation, inheritance, instantiation and overriding are all present. The meta-model shown in Fig. 3 gives an impression of the abstract syntax of the language. The driving intuition behind the semantics is that a TAAL-program has essentially the same meaning as a corresponding Java-program. An important difference with Java is that the start of the program is represented by a single start expression. Listing 1 contains a TAAL-program that will be used as example throughout this paper. More details on the definition of TAAL can be found in [13]. We emphasize that the developed sets of transformation rules enable simulation of any TAAL-program, not just this example.

Some elements from Fig. 3 that will be referred to later in this paper are Program, ObjectType, OperImpl, and VarDecl. The class Program represents the whole program. The Flat Abstract Syntax Graph of any TAAL-program has exactly one instance of this class. Within a TAAL-program we can declare multiple data-structures, each being an instance of ObjectType. Such a data-structure may contain operations (being instances of OperImpl) and fields (being instances of VarDecl). The classes Statement and Expression are abstract and indicate that the language facilitates different types of both.

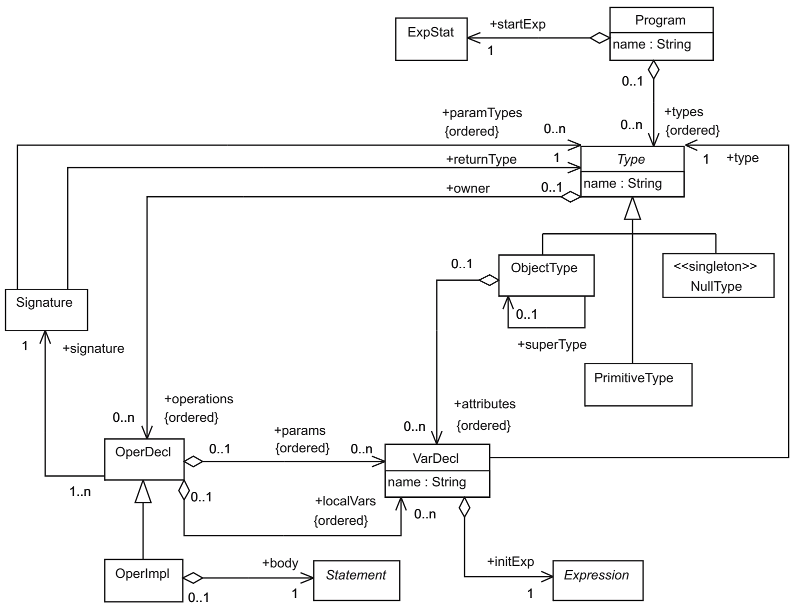

Fig. 3. The types in the abstract syntax graph meta-model

```
program amoebaworld
{ new Amoeba().clone() }
  class Amoeba
  child: Amoeba;
  clone() : Amoeba {
    child := new Amoeba();
    return child;
  }
  endclass
endprogram
```

Listing 1. An example TAAL-program

The result of the parsing and static analysis of a TAAL program (see Fig. 1) is a Flat Abstract Syntax Graph. The Flat Abstract Syntax Graph of the example from Listing 1 is shown in Fig. 4. This graph is an instance of the meta-model from Fig. 3. Some cross-referencing edges have been grayed-out in order to make the graph more readable.

3 Flow Graph Construction

Flow graph construction is the analysis of the flow of control and the construction of flow graph elements which will later on enable the program's simulation. The result of this analysis is the Program Graph (cf. Fig. 1), which consists of the

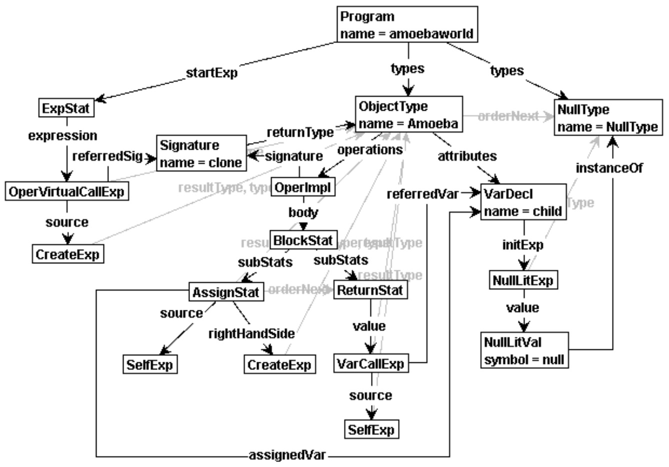

Fig. 4. Flat Abstract Syntax Graph for the example

Flat Abstract Syntax Graph enriched with a number of flow graphs. In this section we will describe the structure of flow graphs and the way we construct them.

Flow Graph Structure. Traditionally (e.g. [9]), flow graphs are directed graphs consisting of four types of nodes (also called flow elements), namely one *start node*, one *end node*, and a number of *procedure* and *predicate nodes* in between, which are connected by *successor*-edges. In our approach we enrich flow graphs with a new node-type, namely the *context node*, and distinct between three types of successor-edges, namely flowNext, flowTrue, and flowFalse. Fig. 5 shows the meta-model of such Flow Graphs.

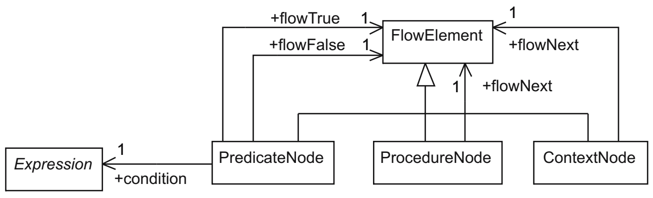

Fig. 5. Flow Graph meta-model

Procedure nodes represent statements or expressions after which it is deterministic which statement to execute next. Predicate nodes represent executable statements and expressions that are related to a boolean condition. The actual value of the condition determines which statement will be executed next. The context nodes represent the start and end node of each Flow Graph. Note that as a result every Flow Graph is cyclic.

The edges in a Flow Graph represent the sequential relation between statements. Fig. 5 shows what kind of edges are allowed between different flow elements. The edges have one of the labels *flowNext*, *flowTrue*, or *flowFalse*.

Flow Graphs, in this paper, appear at three different contexts corresponding to the type of context-node (the types are elements from Fig. 3).

– Program context. Program Flow Graphs control the startup of the program being modelled. In TAAL, program startup is modelled by the execution of the initial expression of the program (line 2 in Listing 1). A Program Graph always contains exactly one Flow Graph at Program context.
– ObjectType context. ObjectType Flow Graphs are traversed when an object is instantiated. Object creation will be discussed in more detail in Sect. 4. A Program Graph contains an ObjectType Flow Graph for each ObjectType being specified in the original program.
– OperImpl context. OperImpl Flow Graphs control the execution of the body of operations. A Program Graph contains an OperImpl Flow Graph for each operation that has been implemented in the original program.

Flow Graphs that appear in the Program Graph at any context are not interconnected. The connection between different Flow Graphs is established during simulation. For example, when instantiating a class inside a operation, the Flow Graph of that operation and the Flow Graph of the object to be created are then 'dynamically connected'. This will be discussed in more detail in Sect. 4.

Graph Transformations for Flow Construction. To extract the flow information from the abstract syntax graph, we apply a set of graph transformation rules that traverses the syntax graph in a *top-down* fashion. The general approach is that for every type of statement or expression we specify a graph transformation rule. Each rule contains a node representing the statement type involved. Fig. 2 showed the Flow Graph construction rule for a VarDecl-element. These graph transformation rules together form a *confluent* graph production system.

After finishing the phase of Flow Graph construction the part of the Program Graph which models the Flow Graphs (i.e. projected on the flow-edges) is an instance of the meta-model shown in Fig. 5. The Program Graph which is constructed from the Flat Abstract Syntax Graph from Fig. 4 is shown in Fig. 6. Elements in Fig. 6 that are not part of any Flow Graph are grayed-out.

4 Simulation

This section presents the next step from Fig. 1, namely defining the operational semantics of TAAL, in terms of graph transformation. The graphs being transformed are so-called *Execution Graphs*, which represent snapshots of the program state. The transformation rules themselves simulate individual program constructs. The resulting GPS, when applied to a flow graph of the kind

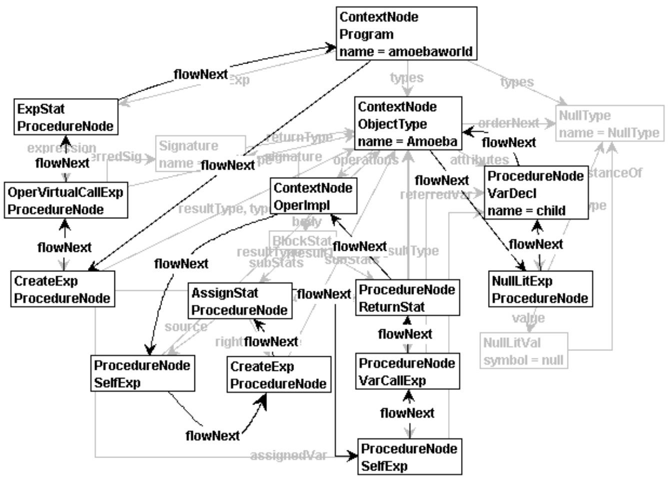

Fig. 6. Program Graph of the example highlighting its Flow Graphs

discussed in the previous section, gives to a transition system, in which the graphs are states and rules applications are transitions. Since program execution is deterministic, so are the transition systems; in other words, at any point in time at most one rule from the GPS is applicable. (In Sect. 5 we briefly discuss the extension of this work to parallel programs, which instead will be non-deterministic, due the independent execution of parallel threads.)

4.1 Execution Graphs

Each Execution Graph combines three kinds of information: a *Program Graph* (see Sect. 3), which provides static context information, a *Value Graph*, which models the data part of the current state, and a *Frame Graph*, which models the process part of the current state. In compiler terms, the Value Graph models the heap and the Frame Graph the stack during program execution.

A Value Graph contains elements representing the objects that will be created and referred to while executing the program. A meta-model for the Value Graph is shown in Fig. 7. The meta-model was inspired by the instance models from [5] and [18]. The new concepts in value graphs are: Value, which stands for any data value, be it a primitive value or an object; and Slot, which is essentially a container for such a value. Slots can either represent program variables (the sub-type VarSlot) or holders of auxiliary, intermediate values (AuxSlot). For the former there is always an associated variable declaration (VarDecl), whereas the latter are bound to Expressions in the Program Graph at which the intermediate values occur.

The Frame Graph meta-model is shown in Fig. 8. It essentially introduces only one new type of node: the Frame. This stands for the execution of the

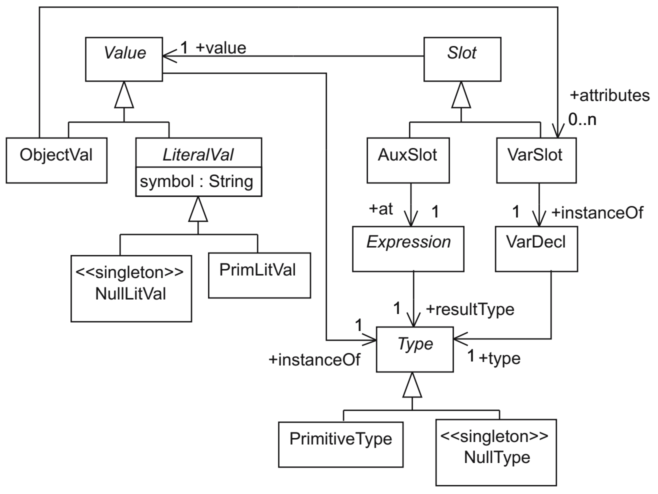

Fig. 7. Value Graph meta-model

program fragment at a ContextNode (see Fig. 5), with a pointer (labelled pc for *program counter*) to the FlowElement in the corresponding Flow Graph at which control currently resides. In fact, for each sub-type of ContextNode there is one Frame sub-type.

An example (partial) Execution Graph can be found in Fig. 9. This represents the state of our example program (Listing 1) before executing the `return`-statement in Line 7. At this moment three frames are active: the ProgramFrame, the OperFrame for the `clone` method, and the ConstrFrame for the creation of the new object.

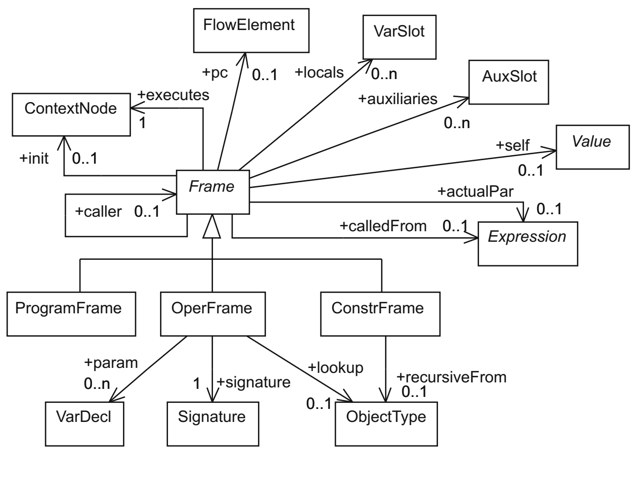

Fig. 8. Frame Graph meta-model

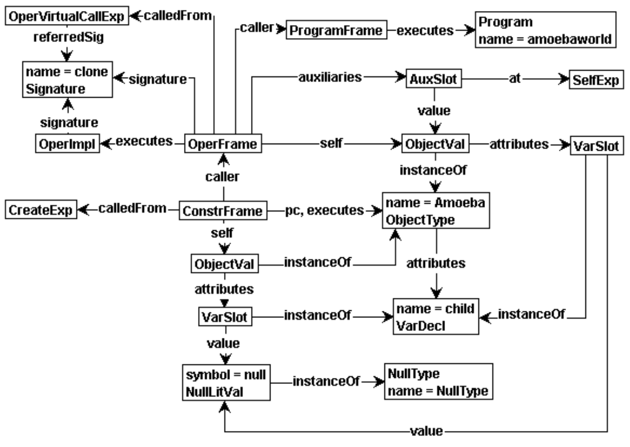

Fig. 9. Fragment of an Execution Graph (at Line 7 of Listing 1)

4.2 Operational Semantics

We now discuss the graph transformation rules that define the dynamic semantics of TAAL. The rules essentially define the effect of the individual statements and expressions of the program in terms of the Value Graph and Frame Graph. For instance, object creation, and assignment to attributes are reflected in the Value Graph, whereas method invocation is reflected mainly in the Frame Graph.

This means that, when we apply the resulting transformation system to the start graph of a given program (being the Program Graph resulting from the Flow Graph construction described in the previous section), each rule application corresponds to the execution of a small step in the program. As an example, Fig. 10 shows the resulting transition system for the example program of Listing 1 in the form of another graph, where the edge labels are rule names. In Sect. 5 we describe the tools used to generate this view.

The complete set of simulation rules for TAAL is too large to include in this paper. They can be found in [13]. Fig. 11 shows a few example rules. The complete set of rules can be divided into three categories: flow element execution rules, object creation rules, and method lookup rules. We believe these three categories to be invariant with respect to the chosen language.

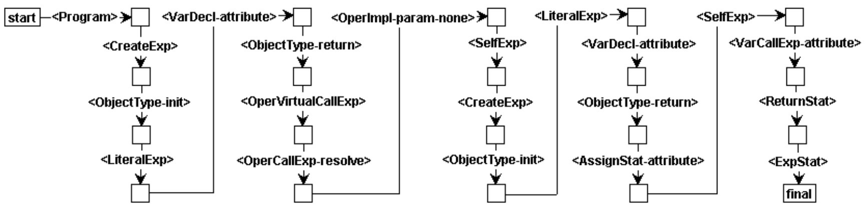

Fig. 10. Transition system of the simulation of Listing 1

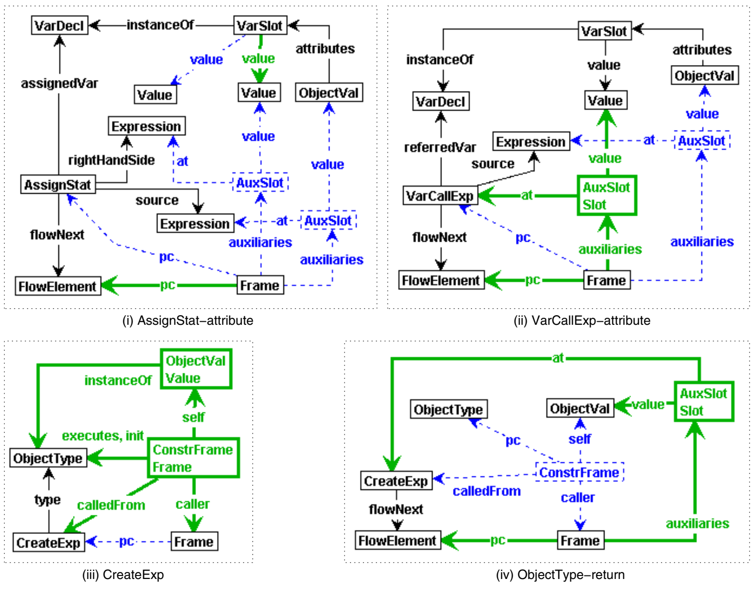

Fig. 11. Example simulation rules

Flow Element Execution. This category consists of a small number of rules (usually one, in some cases two) per kind of FlowElement. These rules describe the essential function of that particular FlowElement. They are always triggered by the fact that the pc-edge from a Frame node (in the Frame Graph) arrives at an instance of the relevant flow element type (in the Program Graph), and they also always adapt the pc-edge to point to a next statement in the Flow Graph. We illustrate this on two examples.

VarCallExp. A VarCallExp is an expression that retrieves the value of a variable. We distinguish two cases: the variable may be an instance variable or attribute (signalled by the fact that the VarCallExp-node has a source) or a local variable or parameter. The first of these is illustrated in Fig. 11 (ii). In either case the referredVar (in the Program Graph) identifies a unique VarSlot (in the Value Graph); execution of the VarCallExp-rule then consists of creating a fresh AuxSlot for the expression and assigning the current value of the referredVar to it. Furthermore, the pc-pointer is moved forward.

AssignStat. The effect of an AssignStat is to make a variable (modelled by a VarSlot) point to a pre-computed value - the rightHandSide of the assignment. Just as for the VarCallExp-rule, we have to distinguish the cases of instance and local variables; the former is illustrated in Fig. 11 (i). In either case, the assignedVar (possibly together with the AuxSlot at the source-referenced Expression) uniquely identifies a VarSlot-instance; this receives the value of the AuxSlot at the rightHandSide. The AuxSlot-instances involved are subsequently discarded.

Object Creation. This consists of allocating and initializing nodes for a new object and its instance variables. In most object-oriented languages, allocation and initialization are done in two different passes, of which the first assigns a default initial value to all fields. In TAAL, we have taken a more simplistic approach: all attributes have an initializing expression. This means we can construct locations for the variables and simultaneously assign initial values to those locations, provided we take care that this process starts at the top of the inheritance hierarchy. This results in the following steps:

Allocation: The actual object creation occurs when control reaches a CreateExp-instance. A ConstrFrame and an ObjectVal are created straight away. The ObjectVal is referenced through self from the ConstrFrame. Moreover, the fresh ConstrFrame has an init-pointer to the ObjectType, to indicate the fact that we are initializing an instance of this type. This is shown as rule CreateExp in Fig. 11 (iii).

Initialization: A ConstrFrame-instance with an init-edge to an ObjectType is treated in either of two ways, depending on whether the ObjectType has a super type. If it has a super type, then a new ConstrFrame is created recursively for that, but with the same self. If it has no super type, then execution is started, by replacing the init-edge with a pc-edge pointing to the first FlowElement reachable from the ObjectType. The subsequent simulation rules will compute initial values and assign them to newly instantiated AuxSlot-instances for the ObjectVal.

Termination: A ConstrFrame terminates when the pc-edge has arrived (back) at the ObjectType. The frame is discarded, and a pc-edge is (re)created at the caller frame. Just as for initialization, there is a case distinction, depending on whether the current frame was called recursively from a sub-type or directly from a CreateExp. The latter case is depicted in Fig. 11 (iv): the ConstrFrame is discarded and the underlying object, pointed to by self, is returned to the caller, where it is assigned to an AuxSlot-instance (also created freshly) storing the value at the CreateExp-node.

Method Lookup. This is a phase that occurs each time after a method is called (through an OperCallExp). The call itself creates a new OperFrame (in the Frame Graph). However, the call (in the Program Graph) only references the *signature* of the method to be executed; a matching method implementation (OperImpl) must be looked up in the server object's self-type. When it is found, the arguments (in the Value Graph) are transferred to that OperImpl's formal parameters. Finally, the new OperFrame is started by creating a pc-edge for it, after which the flow element execution rules take over.

5 Conclusion

The work described in this paper shows a complete example of how programming languages can be defined using graphs and graph transformation rules. The language definition of TAAL includes all necessary parts of a language definition: (abstract) syntax and semantics, which have been defined using a single formalism. Although other work has been presented that uses graphs and graph transformation rules (e.g. [6]) for (parts of) language definitions, none of these reaches the same level of completeness. For instance, the semantics specification in [27] merely includes the static semantics, while our work encompasses execution semantics as well as static semantics. Independently, Hausmann and Engels [11, 8] have developed a similar approach to the definition of language semantics. Both their and our work is based on earlier work by the pUML group [15, 5].

The use of graph transformation rules to specify the semantic rules offers a number of advantages. First, the visual representation of the graph transformation rules provides an intuitive understanding of the semantics. Second, the graph transformation rules offer the possibility to include in one mathematical structure, the graph, information on both the run-time system and the program that is being executed. Traditional approaches to operational semantics (e.g. [1, 26, 19, 4, 12, 2, 7]) often need to revert to inclusion of run-time concepts in the syntax definition, e.g. inclusion of the concept of location to indicate a value that may possibly change over time. This seems to be an artificial manner of integrating parts of the language definition, i.e. of the abstract syntax and the semantic domain, that become much more natural in a graph representation. Finally, in graph transformation rules, context information can be included more naturally and uniformly than for example when using SOS-rules [26].

The example language that we have chosen comprises some of the fundamental aspects of object-oriented programming languages, like inheritance, including dynamic method look-up, and object creation. The structure of our solution makes us confident that the approach can be extended to real-life software languages in the object-oriented paradigm:

– All the transformation steps (parsing, static analysis, flow generation and simulation) are structured according to the concepts in the abstract syntax. This lends a modularity to the definitions that is independent of the language being defined.
– The structure of the Flow and Execution Graphs is generic, in the sense that the elements therein are not specific to TAAL; rather, they capture the essential aspects of imperative, object-oriented languages.

Work that is closely related to ours is by Corradini et al. [6]. They use graph transformations to formalize the semantics of a realistic programming language: they address a fairly large fragment of Java. Technically, the difference is that they interpret method invocation *unfolding* — meaning that the *program graph* changes dynamically. This obviates the need for the frame graph, at the price of having program-dependent rules (namely, one per method implementation).

Another difference is that they provide no tool support, and in that sense theirs is a more theoretical exercise. Another, less directly related source of research is on defining dynamic semantics of (UML-type) design models, where also the idea of using graph transformations has been proposed, e.g. in [8, 16, 25]. Furthermore, in Engels et al. [8] ideas are presented on how to use collaboration diagrams, interpreted as graph transformation rules, for defining SL semantics.

A final aspect of the work reported here is that we have not only developed the TAAL language definition but supporting tools as well. This means that we can actually compile and simulate any TAAL-program and store the resulting transition system so, for instance, all the ingredients for verification are there. Both tool sets as well as the full sets of transformation rules defined the flow generation and simulation phases are available for downloading [14, 21].

An area of further research will be to lift the approach outlined here to a more general level, thus creating a meta language to define software languages, including their semantics. A first step has already been reported in [24], in which rules are specified for building a control flow graph for any imperative object-oriented language. This will give rise to a method for defining the semantics of SLs, which fill the gap currently present in MDA, as pointed out in the introduction. We also intend to investigate whether this approach is applicable for non OO-languages as well. Currently we are working on implementing model checking techniques for verifying object-oriented programs where states are represented as graphs and execution steps as graph transformations.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. Inductive proof outlines for monitors in java. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2003.
3. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
4. K. Bruce, J. Crabtree, and G. Kanapathy. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 603–626. Springer, 1994.
5. T. Clark, A. Evans, S. Kent, S. Brodsky, and S. Cook. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modelling approach, September 2000. Version 1.0 available from `www.puml.org`.
6. A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro. Translating Java code to graph transformation systems. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proceedings of the 2nd International Conference on Graph Transformations (ICGT'04)*, volume 3256 of *Lecture Notes in Computer Science*, pages 383–398. Springer, 2004.
7. F. S. de Boer and C. Pierik. How to cook a complete hoare logic for your pet OO language. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (FMCO'04)*, volume 3188 of *Lecture Notes in Computer Science*, pages 111–133. Springer, 2004.
8. G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of the Third International Conference on the Unified Modelling Language (UML2000)*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.

9. N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach.* International Thomsen Publishing Inc., 2nd edition, 1997.
10. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.
11. J. H. Hausmann. *Dynamic Meta Modeling, A Semantics Description technique for Visual Modeling Languages.* PhD thesis, University of Paderborn, 2006.
12. im B. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.
13. H. Kastenberg, A. Kleppe, and A. Rensink. Engineering object-oriented semantics using graph transformations. CTIT Technical Report 06-12, University of Twente, 2006. Available at `http://www.cs.utwente.nl/~kastenbe/papers/taal.pdf`.
14. A. Kleppe. Taal eclipse plugin, 2006. Available from `http://www.klasse.nl/english/research/taal-install.html`.
15. A. Kleppe and J. Warmer. Unification of static and dynamic semantics of UML, a study in redefining the semantics of the UML using the pUML OO meta modelling approach. Technical report, Klasse Objecten, July 2001. Available at `http://www.klasse.nl/papers/unification-report.pdf`.
16. S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An integrated semantics for UML class, object and state diagrams based on graph transformation. In M. J. Butler, L. Petre, and K. Sere, editors, *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, volume 2335 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2002.
17. OMG. MDA guide version 1.0.1, June 2003. Available from `www.omg.org`.
18. OMG. UML 2.0 OCL specification, October 2003. Available from `www.omg.org`.
19. B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.
20. A. Rensink. The GROOVE Simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2004.
21. A. Rensink. The Groove Tool Set, 2005. Available from `http://groove.sf.net`.
22. A. Rensink. The joys of graph transformation. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 9, 2005.
23. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific, 1997.
24. R. M. Smelik. Specification and construction of control flow semantics. Master's thesis, University of Twente, January 2006.
25. D. Varró. A formal semantics of UML statecharts by model transition systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 1st International Conference on Graph Transformation (ICGT'02)*, volume 2505 of *Lecture Notes in Computer Science*, pages 378–392. Springer, 2002.
26. G. Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, 1993.
27. K.-B. Zhang, M. A. Orgun, and K. Zhang. Visual language semantics specification in the vispro system. In J. S. Jin, P. Eades, D. D. Feng, and H. Yan, editors, *VIP*, volume 22 of *CRPIT*. Australian Computer Society, 2002.