Annales Univ. Sci. Budapest., Sect. Comp. 41 (2013) 57–71

ON SEMANTIC DESCRIPTIONS OF SOFTWARE SYSTEMS

László Kozma, **György Orbán**

(Budapest, Hungary)

Dedicated to Professors Zoltán Daróczy and Imre Kátai on the occasion of their 75th birthday

Communicated by Zoltán Horváth

(Received May 30, 2013; accepted September 28, 2013)

Abstract. Formal methods are essential for giving precise descriptions of software systems. In our paper we analysed some approaches to conventional semantics and to action semantics. As a result we suggest to use action semantics for describing semantic properties of software systems including programming and MDE languages.

1. Introduction

Formal methods are essential for giving precise descriptions of software systems including those systems that implement them. For instance, to formally verify a valid implementation of a programming language amounts to giving formal semantics for the languages. There are three groups of users with different point of view who want to use semantic descriptions of programming

Key words and phrases: Conventional semantics of programming languages, Action semantics, Semantic functions, Algebraic specification of abstract data types.
1998 CR Categories and Descriptors: D.3.1 [Formal Definitions and Theory]: Semantics.
https://doi.org/10.71352/ac.41.057

languages. The denotational semantics is suggested for the language designers; for purposes of compiler builders the operational semantics is a good choice; and for the programmers the axiomatic semantics is suggested. The denotational semantics defines the meanings of the programs by mathematical objects that represent the effect of executing the program. Using denotational approach we would like to know what a program does without any details how the given program is executed. Oppositely the operational approach concentrates on how a program is executed. Using axiomatic semantics of programs we can prove that the given program is correct with respect to its specifications. We can distinguish partial correctness (correct every time when it terminates) and total correctness (terminates and correct every times) of programs [7, 13], etc. Action semantics is useful not only for describing semantics of programming languages including documenting design decisions and setting standards for implementation but for expressing semantic properties of modelling languages as well. Although the action notation looks informal its meaning is formalized in terms of algebraic specifications and transition systems. The action semantics is a combination of denotational, operational and algebraic semantics in spite of this they are quite different from these mentioned conventional semantics. We will summarise the advantages and the drawbacks of different semantic methods. During the analysing process we use a simple kernel programming language WK and its extensions WE. This languages are very close to the language **While** [7] and the language Pelican [14]. The abstract syntax of the language WK is given in BNF form as follows.

n ::= 0 | 1 | n0 | n1.
a ::= n | x | a1 "+" a2 | a1 "*" a2 | a1 "-" a2.
b ::= **true** | **false** | a1 "=" a2 | a1 "≤" a2 | "¬"b | b1 "∧" b2
S ::= x":="a | **skip** | S1 ";" S2 | "**if**" b "**then**" S1 "**else**" S2 | "**while**" b "**do**" S.

Where n ranges over binary numbers **Num**;
x ranges over variables **Var**;
a ranges over arithmetic expressions **Aexp**;
b ranges over Boolean expressions **Bexp**;
S ranges over statements **Stm**.

The **Num**, **Var**, **Aexp**, **Bexp**, **Stm** denote syntactic categories and n, x, a, b, S denote meta-variables those will be used to range over the associated syntactic category. Abstract syntax provides an appropriate interface between concrete syntax and semantics. It is usually obtained simply by ignoring those details of parse tree structure which have no semantic significance. While the syntax of programming languages describe the grammatical structure of programs, then the semantics expresses the meaning of grammatically correct programs using

semantic domains (entities) associated with each syntactic categories respectively. The semantics of a programming language can be captured by a semantic function that maps the abstract syntax of each program to the semantic entity representing its behaviour.

In this section we define the semantics of arithmetic and Boolean expressions and the semantic descriptions of statements are discussed in the next sections.

The semantics of arithmetic expressions can be defined by the following semantic functions: semantic function N: **Num** $\rightarrow$ **Z** determining the number represented by a binary number; semantic function A: **Aexp** $\rightarrow$ **(State** $\rightarrow$ **Z)** that is the value of an expression can be determined by a semantic function A which has two arguments the syntactic construct and the state represented by a function **State = Var** $\rightarrow$ **Z**. The definition of **State** means that to each variable the state will associate its current value.

The values of Boolean expressions are truth values, their meanings can be defined by a function B: **Bexp** $\rightarrow$ **(State** $\rightarrow$ **T)**, where T = {tt, ff} that is T consists of the truth values **tt** (for true) and **ff** (for false).

The detailed descriptions of semantic functions N, A, B can be found in [7]. The rest of this paper analyses the denotational and operational semantics in Section 2. Section 3 discusses some problems of algebraic specifications of abstract data types, Section 4 analysis the action semantics and Section 5 concludes.

2. Analyses of two conventional semantic methods

2.1. Denotational Semantics

One of the most important features of denotational semantics is that semantic functions are defined compositionally. This means that there is a semantic clause for each of the basic elements of the syntactic category and for each composite element in the syntactic category there is a semantic clause defined in terms of the semantic function applied to the immediate constituents of the composite element. In this sense the semantic functions A and B defined in the previous section are examples of denotational definitions. The denotational semantics defines the meanings of the programs by mathematical objects that represent the effect of executing the program. The effect of executing a statement S is to change the state. As an example in the case of direct style semantics the meaning of statement S can be defined by the partial function

$$\mathbf{S_{ds}} : \mathbf{Stm} \rightarrow (\mathbf{State} \hookrightarrow \mathbf{State})$$

As an example the denotational definition of statement **while** b **do** S is as follows:

$$\mathbf{S_{ds}}\ |[\text{"while" b "}\mathbf{do}\text{" S }]| = \mathbf{FIX}\ F$$

where $F\ g = cond(B|[\text{b}]|, g \circ \mathbf{S_{ds}}|[\mathbf{S}]|, \text{id})$, F is continuous function in the sense of Scott and Strachey [8],

$cond$: $(\mathbf{State} \to \mathbf{T}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})$

$$cond(p, g_1, g_2)s = \begin{cases} g_1 & \text{if p s} = \mathbf{tt} \\ g_2 & \text{if p s} = \mathbf{ff} \end{cases}$$

FIX: $((\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})) \to ((\mathbf{State} \hookrightarrow \mathbf{State}))$ defines the least fixed point of F where the set $(\mathbf{State} \hookrightarrow \mathbf{State})$ with a partial order $\sqsubseteq$ is a chain complete partial order set [8]. Function **FIX** is a good example for some kinds of function composition. The detailed definition of this semantic function can be found in [7].

Denitational semantics is very good for purposes of static program analysis including for instance, constant propagation which is an analysis that determines whether an expression always evaluates to a constant value and if so determines that value. The detection of sign analysis is another example. In this case the sign of expressions is determined which is very useful information for code optimization process. During dependency analysis the idea is to regard some of the variables as input variables and others as output variables. Using this analysis we can determine whether or not the final values of the output variables only depend on the initial values of the input variables. A very useful dependency analysis method can be found in [7].

One of the most important drawbacks of denotational semantics is that the addition of new constructs to a described language can require reformulation of the already given description. If we extend our kernel language WK with new constructs where exceptions can be raised and handled we have to change direct style descriptions to a continuation style semantics where the continuations describe the effect of executing the remainder of the program. This requires the reformulation of the direct style description. The new semantic domain $\mathbf{Cont} = \mathbf{State} \hookrightarrow \mathbf{State}$ has to be introduced, which is a partial function from state to state.

Let's extend our language WK with blocks declaring local variables and procedures and denote the resulted language WE, where the new constructs are:

S ::= {rules od WK} | "**begin**" $\text{D}_\text{v}\text{D}_\text{p}$S "**end**" | "**call**" p
D_v ::= "**var**" x":="a; D_v | $\in$
D_p ::= "**proc**" p "**is**" S; D_p | $\in$

where D_v and D_p are meta-variables ranging over the syntactic categories $\mathbf{Dec_v}$ of variable declarations and $\mathbf{Dec_p}$ of procedure declarations, respectively, and

p is a meta-variable ranging over the syntactic category **Pname** of procedure names. Describing semantics of language WK states in **State** have been used to associate values with variables. In case of language WE we shall replace states with *stores* that maps locations to values and with *variable environments* that map variables to locations.

$$\mathbf{Loc} = \mathbf{Z}$$
$$\mathbf{Env_v} = \mathbf{Var} \to \mathbf{Loc}$$
$$\mathbf{Store} = \mathbf{Loc}\ \{\text{next}\} \to \mathbf{Z}$$

where 'next' is a special token used to denote next free location. The detailed descriptions of denotational semantic clauses for language WE using location can be found in[7].

This example demonstrates another drawback of denotational semantics namely introducing new language constructs the semantic functions map syntactic phrases into newer and newer primitive mathematical values, structured objects, and such higher-order functions. On the contrary action semantics uses three kinds of first-order entities as denotations: action, data and yielders (discussed in the next sections). Finally the semantic descriptions of concurrent and parallel programs are almost impossible task in the case of denotational semantics according to their characters of expressing meaning of programs using input-output relations.

2.2. Operational Semantics

The operational semantics are based on abstract machines. They use the notion of configurations denoted by <S, s> or simple s, where <S, s> represents that the statement S is to be executed from the state s and s represents a terminal configuration called in another way final sate. In the case of operational semantics we define the meaning of a program by a transition relation which will then describe how the execution takes place. The difference between the *natural semantics* and the *structural operational semantics* (SOS) amounts to different ways of specifying the transition relation. In natural semantics we are concerned with the relationship between the initial and the final state of an execution. In this case a transition is expressed by <S, s> → s'. This means that the execution of S from s will terminate and the resulting state will be s'. The meaning of statements can be expressed by a partial function from state to state:
$\mathbf{S_{ns}}$**: Stm** → **(State** ↪ **State)** and this means that for every statement S we have a partial function $\mathbf{S_{ns}} \in \mathbf{State} \hookrightarrow \mathbf{State}$ given by

$$\mathbf{S_{ns}}\ \mathbf{[S]s} \begin{cases} s' & \text{if } <\text{S, s}> \to \text{s'} \\ undef & \text{otherwise} \end{cases}$$

The purpose of natural semantics is to express how the overall results of execution can be obtained. This type of semantics is called as big step operational

semantics [7]. The semantic description of parallel constructs is impossible in this case.

The *structural operational semantics* is small step semantics. In this case the transition relation has the forms:

<S, s> ⇒ <S', s'> or
<S, s> ⇒ s'

The transition <S, s> ⇒ <S', s'> means that the execution of S from s is not completed and the configuration <S',s'> expresses the remaining computation that is in structural operational semantics the focus is on the individual steps of the execution. The transition <S,s> ⇒ s' expresses that the execution of S from s has terminated and the final state is s'. The semantic function **S_{sos}: Stm → (State ↪) State** is given by

$$S_{\text{sos}}|[S]|s \begin{cases} s' & \text{if <S, s>} \Rightarrow^* \text{s'} \\ undef & \text{otherwise} \end{cases}$$

where the transition relation <S, s> ⇒* s' denotes that there is finite number of steps from the configuration <S, s> to terminal configuration s'. This transition relation can be defined in a similar manner in case of programming language **While** [7].

The expressive power of structural operation semantics is greater than the natural one. The structural approach can be extended to express semantics of modular language constructs.

The operational semantics of a programming language is useful when implementing it that is when one creates a compiler from the given programming language to a target language. The semantic functions can be used for proving correctness of a compiler. In [7] we can find a very good example for illustrating how to translate programming language **While** into a simple assembly language and the correctness of the implementation is proved as well. The modular structural operational semantics (MSOS) developed by P. D. Moses can be used as a framework for expressing semantic properties of component-based systems. MSOS is a simple, but very useful variant of structural approach. The MSOS uses so-called basic abstract constructs, they have fixed, language-independent abstract syntax and semantics and their syntax and semantics can be specified by formally. The set of basic abstract constructs is open-ended, that is new constructs may be added whenever the previous constructs are found to be insufficiently expressive [19]. One of the most important drawbacks of operational semantics is that the semantic functions S_{ns} and S_{sos} are not defined compositionally. They associate mathematical objects with each statement as well, but they can't be constructed by tools of composition. So the operational semantics do not scale up well to specifying the intended classes of implementations of larger programming languages used for industrial purposes.

3. Semantic problems of abstract data types

Abstract data types are very important during the design a software systems based on object-oriented paradigm [9, 10, 11, 5, 6, 15, 16, 17, 13]. The specification of a data type is a high level description of the data structure and of each operation of that data type. The semantics of an operation can be described by algebraic equations, by pre- and post-conditions, or by body of procedures given in a high level programming language. Abstract specifications of data types are very important not only in program design but in verification as well. The correctness of an implementation of an abstract data type should be proved and this verification is possible. Let us see the following circular list example where the abstract data type and its implementation are given by algebraic specifications.

clist = ({clist, elem, bool}, {create, insert, del, value, isempty, right, join})

syntax	**semantics**
create: → clist	del(create) = create
insert: clist x elem → clist	del(insert(c,i))=c
del: clist → clist	value(create) =**UNDEF**
value: clist → elem ∪ {**UNDEF**}	value(insert(c,i))=i
isempty: clist → bool	isempty(create) = **true**
right: clist → clist	isempty(insert(c,i)) = **false**
join: clist x clist → clist	right(create) = create
	right(insert(create,i)) = insert(create,i)
	right(insert(insert(c,i),j)) =
	insert(right(insert(c,j)),i)
	join(c, create) = c
	join(c,insert(c_1,i)) = insert(join(c,c_1),i)

where c, c_1 ∈ clist, i, j ∈ elem.

The abstract type *circular_list* is very similar to the type *stack*, but the operations *right* and *join* introduce additional complexity by allowing the users to rotate the list of stored elements and to join two lists into one. Using the *right* operation makes it possible to access to both ends of the list.

The implementation of the abstract data type circular_list is given by data type array, the well-known data type nat with operations + and and data type bool.

array = (V,F), where V = {array, nat, elem}, F = { emptya, assign, read, shiftL, shiftR }

syntax

emptya:	→ array	//create any empty array
assing:	array x nat x elem→ array	//put an empty array to a given position
read:	array x nat → elem ∪ {undef}	//read an element from a given position
shiftL:	array → array	//the array is shifted left by one step
shiftR:	array → array	//the array is shifted right by one step

semantics

read(empty, a)	= undef
shiftL (emptya)	= emptya
shiftR (emptya)	= emptya
read (assign(a,i,e),j)	= if i=j then e else read(a,j)
shiftL (assign(a,i,e))	= if i = zero then shiftL (a) else assign (shiftL(a),i-1,e)
shiftR (assign(a,i,e))	= assign(shiftR(a),i+1,e)

where a ∈ **array**, i ∈ **nat**, e ∈ elem.

We will represent the abstract data type clist by a concrete type array, a nat and a bool. The representation function φ is defined by:
clist = φ(array, nat).
The operations of the abstract data type clist can be implemented as follows:

create	= φ(emptya, zero)
insert(φ(a,n),e)	= φ(assign(a,n,e),n+1)
del(φ(a,n))	= if n = zero then φ(emptya,zero) else φ(a,n-1)
value(φ(a,n))	= if n = zero then undef else read(a, n-1)
right(φ(a,n))	= if n = zero then φ(emptya, zero)
	else φ(assign(shiftR(a),zero, read(a, n-1)),n)
isempty(φ(a,n))	= if n = zero then **true** else **false**
join(φ(a,i),φ(b,j))	= if j = zero then φ(a,i)
	else join(φ(assign(a,i,read(b,zero)),i+1),φ(shiftL(b),j-1))

A proof of correctness of this above implementation consists of showing that all of the clist axioms are satisfied. For instance we have to prove the following lemma.

Lemma 3.1. *For every e ∈ elem the axiom right(insert(create, e)) = insert(create, e) is satisfied substituting the implemented versions of operations create, insert and right in it.*

Proof. First we use the implementation equations of operations create, insert and right, then we apply semantic axioms of array. Finally using the implementation equations of operations insert and create again, we can get the result.
right(insert(create, e)) = right(insert(φ(emptya, zero), e) =
= right(φ(assign(emptya, zero, e), 1)) = if 1 = zero then ...
else φ(assign(shiftR(emptya), zero, read(assign(emptya, zero, e), zero)), 1) =
= φ(assign(emptya, zero, e), 1) = insert(φ(emptya, zero), e) =
= insert(create, e). ■

During the proving process we can use clauses of an implementation, the semantic axioms of the concrete data types and structural induction method, but we can't use the semantic axioms of abstract data types. The representation function φ has very important role from point of view of correctness. The whole proving process fails if the representation function is incorrect. We have

to check whether the function is correct or not in the sense that each c ∈ clist there exists at least one a ∈ array and n ∈ nat so that c = φ(a, n). We can prove the following lemma in case our example.

Lemma 3.2. *The abstract data type* ***clist*** *is correctly represented by function* φ.

Proof. We have to prove that for every c ∈ clist there exists at least one a ∈ array and n ∈ nat so that c = φ(a, n). Using the results 3.3 Lemma it is enough to prove that c = create and c = insert(c', e) are represented well are proved by induction on the structure of clist.

1. step. Let us see the case when c = create. Consulting the implementation of clist we have create = φ(emptya, zero) so emptya is a good array and zero is a good pointer.
2. step. Suppose there exist a good representation for c' that is an array a' and a pointer n'. According to the implementation of operation insert we have insert(φ(a,n), e) = φ(assign(a, n, e), n+1), so c = insert(c', e) = insert(φ(a', n'), e) = φ(assign(a', n', e), n'+1). This gives the results: the array assign(a', n', e) and pointer n'+1 are good representation of c ∈ clist.

The proof can now be completed by using the results of 3.3 Lemma. ■

Lemma 3.3. *For every c ∈ clist, c = create or there exists a c' ∈ clist and e' ∈ elem such that c = insert(c', e').*

This lemma was proved by Guttag, Horowitz and Musser [11].
From point of view of software design it is very important feature of algebraic specification of abstract data types that the inheritance relations can be expressed which are very important during design phase of software systems [3, 13].

4. Action semantics

The action semantics is a combination of denotational, operational and algebraic semantics. The action semantics follows the tradition of denotational semantics in the sense that the syntactic entities are mapped compositionally by semantic functions into semantic domains (semantic entities) that act as the denotations of the syntactic objects. The main differences are the following. The semantic functions of denotational semantics map syntactic phrases into several primitive and higher order mathematical objects, the action semantics uses three kinds of first-order entities as denotations: actions, data and yielders. Actions are dynamic (computational) entities. An action represents information processing behaviour and reflects the gradual step-wise nature of

computation. They are mathematical entities in the sense that they are abstract and formally-defined entities, analogous to abstract machines in the same way of operational semantics. In contrast, items of data essentially static entities, representing pieces of information. Data consists of mathematical values, such as integers, Boolean values, and abstract cells representing memory locations similarly in denotational semantics. A yielder represents an unevaluated item of data, whose value depends on the current information embodying the state of computation that is yielders are entities - depending on the current storage and environment - that can be evaluated to yield data. In action semantics, we specify semantic functions by semantic equations. Each equation defines the semantics of a particular phrase in terms of the semantics of its components that is the semantic functions are defined compositionally. For instance, the semantic functions of kernel language WK are *execute* _, *evaluate* _, the *operation-result of* _, the *value of* _. First we give the semantic functions of expressions and then of statements. The semantic functions of expressions are as follows.

evaluate _ :: Expression → action [giving a value].
evaluate N:n = give the value of n.
evaluate x:Identifier = give the value bound to x or give the
number stored in the cell bound to x
evaluate |[a1:Expression O:Operator a2:Expression]| =
(evaluate a1 and evaluate a2) then give the operation-result of O.

the operation-result of _ :: Operator → yielder [of a value] [using the given value2]
the operation-result of "+" = the sum of (the given number#1,
the given number#2).
...
the operation-result of "=" = the given vale#1 is the given value#2
the operation-result of "∧" = both of (the given truth-value#1,
the given truth-value#2)
...
the value of _ :: Numeral → number.
the value of n:Numeral = number & decimal n.

In action semantics it is usual to specify the functionality of each semantic function. For instance, evaluate _ :: Expression → action [giving a value] asserts that for every abstract syntax tree E for an expression, the semantic entity evaluate E is an action which when performed gives a value.

Each semantic equation defines the result of applying a particular semantic function to any abstract syntax tree whose root node has the indicated form, in terms of applying (perhaps different) semantic functions to the branches of the node. We use notations |[...]| informally as separating syntactic symbols from semantic symbols only in the left hand side of a semantic equation. We do not write |[E]| and |[S]| on the right hand side of a semantic equation, since E and S by themselves already stand for tree.

The semantic functions of statement for the language WK is the e following:
execute _ :: Statements → action [completing | diverging | storing].
execute |[x:Identifier "=" a:Expression]| = (give the cell bound to x and evaluate a) then store the given number #2 in the given cell #1.
execute <S_1:Statement ";" S_2:Statement> = execute S_1 and then execute S_2.
execute |["if" b:Expression "then" S_1: Statement "else" S_2 : Statement]| =

$$evaluate\ b\ then \begin{cases} check\ the\ given\ truth-value\ and\ then\ execute\ S_1 \\ or \\ check\ not\ the\ given\ truth-value\ and\ then\ execute\ S_2 \end{cases}$$

execute |["while" b:Expression "do" S:Statement]| = unfolding

$$evaluate\ b\ then \begin{cases} check\ the\ given\ truth-value\ and\ then\ execute\ Sand\ then\ unfold \\ or \\ check\ not\ the\ given\ truth-value\ and\ then\ complete. \end{cases}$$

The semantic entities are actions, data and yielders.

The properties of *actions* are performance, non-determinism, information classification, facets and combinators.

The *performance* of an action directly represents information processing behaviour and reflects the gradual step-wise nature of computation. A performance can be part of an enclosing action: completes, escapes, fails or diverges corresponding to normal termination, to exceptional termination, to abortion or to non-termination, respectively.

An action may be *non-deterministic*. Non-determinism represents implementation dependence, where the behaviour of a given program may vary between different implementations or between different instants of time on the same implementation.

The *information* processed by an action performance can be *classified* according to how far it tends to be propagated:

transient: tuples of data, corresponding to intermediate results
scoped: bindings of tokens to data, corresponding to symbol tables
stable: data stored in cells, corresponding to the values assigned to variables
permanent: data communicated between distributed actions

The different kinds of information give rise to so-called *facets* of actions, focusing on the processing of at most one kind of information at a time:

the basic facet, processing independently of information
the functional facet, processing transient information
the declarative facet, processing scoped information
the imperative facet, processing stable information, action reserve or unreserved cells of storage and change the data stored in cells
the communicative facet, processing permanent information, action send and/or receive messages

We can use primitives for specifying actions including action *combinators*, which operate on sub actions.

The semantic entity *data* can include different mathematical objects, such as numbers, truth-values, characters, strings, lists, maps and sets. It can include computational entities as well, such as tokens, cells, some compound entities with data components including messages and contracts. It is very important property of action semantics that new kinds of data can be introduced ad-hoc, for representing special pieces of information. This property increases the expressive power of action semantics while the low number of semantic entities makes it easy to use this semantics.

The semantic entities *yielders* can be evaluated to yield data during action performance. Compound yielders can be formed by the application of data operations to yielders. The data yielded by evaluating a compound yielder are the result of applying the operation to the data yielded by evaluating the operands.

A very good introduction to action semantics can be found in [18]. Action semantics uses abstract contracts similar to MSOS and the set of basic abstract constructs is open-ended in this case as well. When further language constructs are added to a given language, the action semantics of each old language constructs remains well-formed and meaningful that is addition of new constructs to a described language can't require reformulation of the already given description.

The action semantics can be used not only for describing semantic properties of programming languages, but it is suitable for describing semantics of languages in Model Driven Engineering (MDE) as well. Languages in MDE are generally defined by metamodels, which specify the structural aspects of models but do not capture their dynamic semantics. The computational meanings (the dynamic semantics) of modelling constructs can be described well by action semantics as it allows modular semantic specifications and provides an intuitive textual notation at the same time. G. Stuurman and I. Kurtev developed a compiler for translating models in MDE to action tree and a simulator for executing these action trees. The advantage of these tools is in the fact that in this way the models become executable and their behaviour can be studied at the early modelling phase [4]. In [20] a method is presented of Meta Object Facility (MOF 2.0) to include action semantics for supporting behavioural modelling. A new approach is presented in [1] for automatically generating test cases from UML state machine. Achieving this aim they give UML a formal semantics by developing a new and non-trivial mapping from UML state machines to an object-oriented version of action systems.

In the MDE environment there are two kinds of semantic languages. The first kind is an Action Semantics Language (ASL) with a whole new platform-independent syntax. The second kind is when an existing programming lan-

guage is used as an ASL but in that case it can lead to platform dependencies. There are many Action Semantic Languages to describe more precisely the software systems at a higher abstraction level but there is a lack of a formal standardized ASL. Many different types of Action Semantic Languages were created by different vendors like the Object Action Language (OAL) by MentorGraphics for xtUML or Kennedy Carter declares xUML and provides Intelligent UML (iUML). Action languages can be used also to specify mapping rules to generate a Platform Specific Model (PSM) from a Platform Independent Model (PIM). An Action Semantics Language can be built based on OCL like in OCL4X[12] with the support of actions with side effects.

5. Conclusions

The definition of semantics of programming and recently MDE languages is crucial for understanding and using these languages. In our paper we analysed some approaches to conventional semantics and to action semantics as well. We suggest to use action semantics for describing semantic properties of software systems including programming and MDE languages as well. Action semantics framework support creating specifications given in a natural language-like notation and uses concepts familiar to programmers. There are further approaches to express semantics of MDE languages. For instance, in [2] authors presented semantic descriptions for automatic transformation of UML statechart diagram into its equivalent finite state automata and a method is given to generate regular grammar for the generated finite state automata. Using this grammar you can generate various test cases to verify UML statechart diagram with respect to its various test conditions.

References

[1] **Aichering, B.K., H. Brandl, E. Jöbstl and W. Krenn,** UML in Action: A Two–Layered Interpretation for testing, *ACM SIGSOFT Software Engineering Notes*, Vol. **36**, No. 1, January 2011, pp. 1–8.

[2] **Arora, D., B. Hazela and V. Saxena,** Semantics for UML Model Transformation and Generation of Regular Grammar, *ACM SIGSOFT Software Engineering Notes*, Vol. **37**, No. 3, May 2012.

[3] **Parisi–Precise, F. and A. Pierantonio,** An algebraic theory of class specification, *ACM Trans, On Soft., Eng., and Meth.*, Vol. **3**, No., 2, 1994, pp. 166–199.

[4] **Stuurman, G. and I. Kurtev,** Action semantics for defining dynamic semantics of modelling languages, in: *BM–FA'11: Proc. of the Third Workshop on Behavioural Modelling*, June 2011, pp. 64–71.

[5] **Ehrig, E. and B. Mahr,** *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*, Springer Verlag, 1985., ISBN 3–540–13718–1.

[6] **Ehrig, H and B. Mahr,** *Fundamentals of Algebraic Specification 2, Module Specifications and Constraints*, Springer Verlag, 1990., ISBN 3–540–51799–5.

[7] **Nielson, H.R. and F. Nielson,** *Semantics with Applications, A Formal Introduction*, John Wiley & Sons, 1992.

[8] **Stoy, J.E.,** *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[9] **Guttag, J.V., E. Horowitz and D.R. Musser,** *The Design of Data Type Specifications*, Research report, University of Southern California Maria del Rey Information Sciences Institute, November, 1976.

[10] **Guttag, J.V., E. Horowitz and D.R. Musser,** Abstract data types and software validation, *Comm. ACM*, Vol. **21(12)**, (1978), 1048–1064.

[11] **Guttag, J.V., E. Horowitz and D.R. Musser,** The design of data type specifications, In: *Current Trend sin Programming Methodology*, R. T. Yeh, Ed., Prentice–Hall, Englewood Cliffs, N. J. 1978. pp. 60–79.

[12] **Ke Jiang, Lei Zhang, and Shigeru Miyake,** 2007. OCL4X: An Action Semantics Language for UML Model Execution, In: *Proceedings of the 31st Annual International Computer Software and Applications Conference*, Volume 01 (COMPSAC '07), Vol. 1. IEEE Computer Society, Washington, DC, USA, 633–636. 2007.

[13] **Kozma, L. and L. Varga,** *A Szoftvertechnológia Elméleti kérdései*, ELTE Eötvös Kiadó, 2006, ISBN 963–463–648–9 (in Hungarian).

[14] **Slonneger, K. and B.K. Kurtz,** *Formal Syntax and Semantics of Programming Languages:A Laboratory Based Approach*, Addison–Wesley, 1994.

[15] **Varga, L.,** On the verification of abstract data types, *Acta Cybernetica*, Tom. **6**, Fasc. 1, Szeged, 1983., 7–12.

[16] **Varga, L.,** Típusspecifikációk helyességének vizsgálata, *Alkalmazott matematikai Lapok*, **13** (1987–88), 57–68.

[17] **Kozma, L.,** Proving the correctness of implementations of shared data abstractions, *Lecture Notes in Computer Science (LNCS)*, **137** (1982), 227–241.

[18] **Mosses, P.D.,** *A Tutorial on Action Semantics*, FME'94.

[19] **Mosses, P.D.,** Component-Based Semantics, SAVCBS'09, In: *Proc. of the 8th International Workshop on Specification and Verification of Component–Based Systems*, 2009.

[20] **Paige, R.F., D.S. Kolosov and F.A.C. Polack,** An action semantics for MOF 2.0, SAC'06, In: *Proc. of the 2006 ACM Symposium on Applied Computing*, April, 2006, pp. 1304–1305.

L. Kozma and Gy. Orbán
Faculty of Informatics
Eötvös Loránd University
Pázmány P. sétány 1/C.
H-1117 Budapest
Hungary
kozma@ludens.elte.hu
o.gyorgy@gmail.com