

# Operational Semantics in a Domain-Specific Robot Control Language: a Pedagogical Use Case\*

William Steingartner and Valerie Novitzká

Technical University of Košice, Faculty of Electrical Engineering and Informatics  
Košice, Slovakia

{william.steingartner, valerie.novitzka}@tuke.sk

**Abstract.** In this paper, we focus on some aspects of structural operational semantics for a selected domain-specific language for robot control. After defining a syntax for two independent languages for control of a robot’s movements, we define a structural operational semantics for them. The integrated part of structural operational semantics is abstract implementation, which consists of defining abstract machine and transformation of a programming language to abstract machine instructions. The achieved results as well as the mentioned research are a part of the research in the field of semantic methods, where we focus on the formalization of semantic methods for software engineering. This area is also very important for the training of students and young IT experts because the semantic methods can help to understand program behavior and detect errors in program design. To make the teaching of formal semantics in the field of domain-specific languages more attractive, we have also prepared an application that serves to visualize the individual steps of the program on an abstract machine – simulation of translated code with visualization of a robot’s movement.

**Keywords:** Abstract machine, containerization, domain-specific language, formal semantics, micro-service, online teaching, teaching software, university didactic.

## 1. Introduction

The main aim of this paper is to define an abstract machine and abstract implementation as a part of structural operational semantics for a simple domain-specific language. After that, we design an application that serves as an emulation tool of the defined abstract machine. For its design and implementation, a modular approach was taken. Our approach is primarily dedicated to the students visiting our graduate course Semantics of Programming Languages.

Domain-specific languages (DSLs) constitute a new class of programming languages and they can be characterized as a new programming paradigm. These languages are designed as special-purpose programming languages with a higher level of abstraction to support a particular set of tasks, meant to be used in the contexts of specific domains [2]. DSLs are defined in [8] as programming languages or executable specification languages that offer expressive power focused on, and usually are restricted to particular problem domains. At present, there are several DSLs for various domains: domain-specific functional

---

\* This paper is an extended version of the conference paper “Abstract machine for operational semantics of domain-specific language” presented in a workshop 4th Workshop on Modern Approaches in Data Engineering and Information System Design organized in conjunction with the conference ADBIS 2022.

languages for dynamic geometry [28], for finance (for instance Marlowe [17] which allows users to create blockchain applications tailored for financial contracts, or Findel [3], designed for financial derivatives and boasts simplicity and expressiveness for complex derivative expressions); legislative, and juridical applications (e.g. ERGO [29] which focuses on legal contract execution logic, aiming for consistency between legal prose and executable logic, accessible to both lawyers and developers); for asset management (e.g. Psamathe [21]); for multidomain (e.g. Jabuti [9] catering the smart contracts in application integration, offering natural language-like constructors, ContextMapper [15] grounded in Domain-Driven Design principles aiding in context mapping and service decomposition, Adico [11] serving as the interface between human-readable institutional specifications and machine-readable Solidity contracts (referring to Ethereum, see [41]), Archetype [40] focused on developing smart contracts on the Tezos blockchain, with a specific focus on the formal verification of the contract, ink! [22] as a programming language dedicated to smart contracts (referring to Polkadot, see [42]), and iContractML [12] facilitating modeling and deploying smart contracts across multiple blockchain platforms); for smart contract formal analysis and verification (VeriSolid, Scilla), for modeling platform independent model specifications of a specific information system [18], for the development of software agents [6] or specifically for modeling application-specific functionalities of business applications [26] and many others. For more interested user, we recommend more comprehensive survey [1] in blockchain model. Because many of mentioned problem domains require deep knowledge of their principles and needs, they are not appropriate for teaching students because many details are not needed for explaining how to define the semantics of DSLs. Therefore we decided to use a simple DSL language Robot that is suitable for our purposes. We note that the first idea about this robot language was presented in [25].

A formal definition of each programming language consists of a formal syntax and formal semantics [16]. The formal syntax can be concrete and abstract. Generally, the concrete syntax is the set of rules that defines the combinations of symbols that are considered to be well-formed in a computer language. Abstract syntax defines the structure of syntactic entities without some details, and it is needed for defining the formal semantics of a language. Under semantics, we understand the meaning of such structures.

Formal semantics is an integral part of programming language's formal definition. It offers students studying computer science and IT experts to understand the meaning of the programs and the opportunity to better understand how the code execution works on the machine using abstraction, thus removing some unnecessary details which can be ignored.

Based on the above, this suggests that strong support for formal methods and principles in the field of software engineering and software development at the level of university education will help to acquire critical thinking and support abstraction in solving problems, whether from the point of view of designing more effective algorithms, verifying the properties of languages programs written in this language and thus prevent many (logical) errors in various phases of the design and development of software systems. Since all formal methods are rooted in formal semantics, we are also of the opinion that interactive support in the teaching of semantic methods will make it easier to learn the principles of these methods, as well as easier to navigate in the choice of a suitable semantic method, or to correctly interpret the results achieved at all levels of derivation. There-

fore, visualization software is one of the interactive aids that we present in an effort to modernize and make teaching more attractive. Orientation to domain-specific languages will make it easier for students to understand the principles that are standardly presented in imperative and functional languages. Students will more easily understand how these paradigms are related and how individual methods are modified when used across different paradigms. We would like to add that, from a practical point of view, support is still available when using the SLANG [31] tool. It is a software for the rapid prototyping of a programming language from its formal specification; which also allows belief in its semantic specification (currently support for denotational and natural operational semantics).

The students of our course become familiar with several semantic methods, such as natural semantics, denotational semantics, algebraic semantics, and axiomatic semantics. Natural semantics, or the semantics of big steps is the simplest kind of semantic method and is well understandable by students. Denotational semantics define the meaning of programs by functions and it does not keep track of execution details. Algebraic semantics specifies programs as a hierarchy of abstract data types and defines the semantics as heterogeneous algebras. Axiomatic semantics is suitable for verification purposes by defining preconditions and postconditions before and after executing the statements, respectively. One of the most popular semantics methods is structural operational semantics [20], also called the semantics of small steps, which defines how individual steps of programs are executed. This method requires to define also abstract implementation based on an abstract machine and we concentrate on this method in our paper.

Knowledge of structural operational semantics and the appropriate abstract machine is an important foundation in the teaching of programming languages. It offers students the opportunity to better understand how code execution works on the machine using abstraction, thus removing some unnecessary details which can be ignored. However, the related abstract machine theory can be unfriendly or difficult for students to learn.

To overcome this problem, we follow the idea of making formal semantic methods visual in particular steps of calculations. The current situation requires that students also acquire skills in working with this type of educational software, as stated by the author in [32]. In educating young IT experts, there is a need for tools that can illustrate the semantic methods to students and make it clearer how for example an abstract machine operates interactively for not only self-study for students but also for teachers to present abstract machine in their lectures. Many times, the introduction to programming is explained in languages similar to the original idea of Karel the Robot (or simply *Robot*). Therefore, we were also inspired by this approach and expanded our research in the field of domain-specific languages, where we work with a language for robot control. The domain-specific language we work with is the language Robot which is an entity moving in two-dimensional space by executing given commands.

The abstract implementation of a robotic language represents a natural continuation in the field of semantic methods, leveraging structural operational semantics for its definition. It is also a great motivation for us, as other semantic approaches are already formulated and derived: denotational and operational semantics are defined in [13], we defined natural semantics in [36] and we processed its visualization in [38]. It is also natural when extending the spectrum of semantic methods to verify that the new method is fully equivalent to the existing ones and that it provides the same results. Then we have in mind a

deterministic approach in the application of semantic methods. We formulated the proof of equivalence between the denotational semantics of the Robot language and the natural semantics defined by us in [37].

In this article, we present a practical approach to the visualization of the semantic method for a given DSL language, while the results achieved here are based on our previously mentioned works. In this way, we have, on the one hand, extended the field of semantic methods for a given language and thus given the user the opportunity to choose a method according to the pursued goal (only the result, detailed calculation procedure, verification of language properties, etc.), and on the other hand, we have also provided a visualization module that will enable the interactive use of the mentioned method, tracking the progress of the calculation or checking the results.

The aim of this paper is to define an abstract machine for the structural operational semantics of the robot language and for its abstract implementation. After that, we designed an application that serves as an emulation tool of abstract machine defined. For its design and implementation, a modular approach was taken. The application is divided into four micro-services: translation logic, translation graphical user interface, simulation logic, and simulation graphical user interface, with a database in the separate (fifth) container.

This paper is a part of our project to prepare young engineers and IT experts to deal with semantic methods. We have prepared the applications for teaching and learning denotational semantics [35], operational semantics [34] and abstract machine [33]. Our needs are so specific that we had problems finding similar applications. We followed the methods published in [30] with some modifications. The aim of our project is to provide an integrated system for learning and teaching semantic methods. This paper serves for illustrating how domain-specific languages differ in using operational semantics.

The purpose of the logic components is to parse the given input source code and to provide an output form (bytecode) for visual processing and calculation. The processed input for the translation logic is afterwards translated to abstract machine code and for the simulation logic, the data is used for simulating the execution on the abstract machine, returning simulation data. The web is used as a graphical user interface for both components using Spring with Thymeleaf template engine for displaying variable data on the HTML page which uses JavaScript scripts for additional calls. Web pages have both English and Slovak localization with the link from the compiler to the simulator and back.

The paper is structured as follows: in Section 2, we present a syntax of the language for robot and preliminaries for defining the semantics. Section 3 contains a definition of the abstract machine(s) for the robot language(s). Section 4 focuses on the design of an emulation of an abstract machine, and Section 5 presents its specification and functionality. Finally, Section 6 concludes our paper.

## 2. Simple Domain-Specific Language for Robot

A domain-specific language (DSL) is a programming language or executable specification language that, through appropriate notation and abstraction, offers expressive power focused on, and usually limited to, a specific domain of the problem [8]. Domain-specific languages are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application [19]. With the creation of a DSL, a user can focus

on the problem and its solution in a specific area for further processing. One of the possible benefits of using DSL is lower implementation time, reduced maintenance costs, as well as better portability, reliability, optimizability and testability [7]. The language of the robot is an external DSL which means it is a fully independent language with its syntax and semantics [14]. Language workbench is a great tool for creating such a language – it is easy to define not the only parser, but also a custom editing environment [10].

The operational semantics of the language of a robot is a base for abstract machine definition. The robot is an entity in two-dimensional space, a grid, with specified  $x$  (horizontal) and  $y$  (vertical) coordinates which are typically changed during the execution of a program. The robot can move horizontally and vertically, diagonal movement is not possible in this simplified version. Language has two syntactic domains [13]:

- $n \in \mathbf{Num}$  – numerals specifying number of steps which robot has to move;
- $C \in \mathbf{Comm}$  – commands.

The syntactic domain is specified by an abstract syntax giving the structure of the basic elements and the composite ones. The syntactic domain  $\mathbf{Num}$  has from the point of view of abstract syntax no internal structure (but syntactically numerals can be represented with a regular expression  $[0, \dots, 9]^+$ ). We need to define abstract syntax only for syntactic domain  $\mathbf{Comm}$  for commands. The structure of well-structured commands we define using BNF formalism by production rule as follows:

$$C ::= \mathbf{left} \mid \mathbf{right} \mid \mathbf{up} \mid \mathbf{down} \mid \mathbf{left} \ n \mid \mathbf{right} \ n \mid \mathbf{up} \ n \mid \mathbf{down} \ n \mid \\ \mathbf{skip} \mid \mathbf{reset} \mid C; C.$$

The commands **left** and **right** express the movement of a robot one position to the left or to the right, respectively. The commands **up** and **down** express the movement of a robot to the up or to the down, respectively. The commands **left**  $n$ , **right**  $n$ , **up**  $n$  and **down**  $n$  express the particular movements by  $n$  position in the specified direction. The **reset** statement invokes an immediate return to the initial, or default, position set by the user at the beginning (akin to *teleportation*). Next, the **skip** statement is void of action, serving primarily as a placeholder in proofs of semantic equivalence. The composition (sequence) of commands is expressed by the last pattern.

For the robot's position, we define the semantic domain

$$\mathbf{Point} = \mathbb{Z} \times \mathbb{Z}$$

with the elements  $p$  that are considered states. Each position has a form

$$p = (x, y)$$

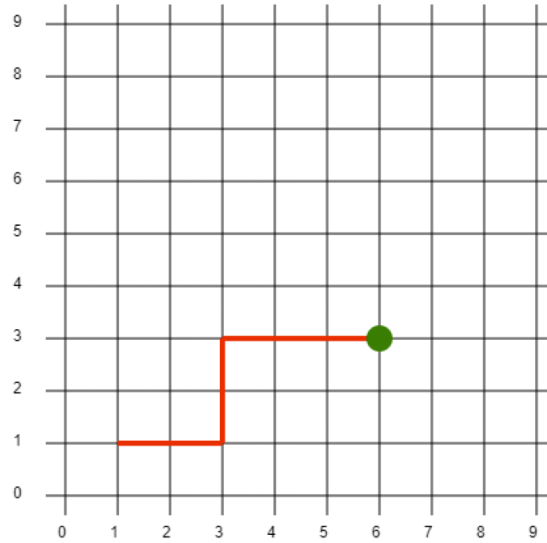
where  $x$  and  $y$  are the coordinates. The change in the position of the robot is considered a change of state. For simplification, we consider that our grid is not limited in any direction.

**Example 1:** Assume a simple program of robot:

```

right 2;
up 2;
right 3

```



**Fig. 1.** Movement of a robot

A robot moves two positions right, then two positions up and finally three positions right. The robot movement according to this program is illustrated as a path along the orthogonal grid (Fig. 1). The position

$$p^* = (1, 1)$$

is the starting position of a robot, i.e. the coordinates are  $x = 1$  and  $y = 1$ .

As we showed in [37], for natural semantics of  $n$ -step statements, we can use direct evaluation, rewriting or inductive definition. Using the first approach, the semantics of the program is evaluated by the following derivation tree:

$$\frac{\langle \mathbf{right\ 2}, p^* \rangle \rightarrow p_1 \quad \frac{\langle \mathbf{up\ 2}, p_1 \rangle \rightarrow p_2 \quad \langle \mathbf{right\ 3}, p_2 \rangle \rightarrow p}{\langle \mathbf{up\ 2}; \mathbf{right\ 3}, p_1 \rangle \rightarrow p}}{\langle \mathbf{right\ 2}; \mathbf{up\ 2}; \mathbf{right\ 3}, p^* \rangle \rightarrow p}$$

and the internal states are:  $p_1 = (3, 1)$  and  $p_2 = (3, 3)$ . After performing all steps, the robot ends up at the position  $p = (6, 3)$ . □

The semantics of the commands is given by evaluation of the semantic function

$$\mathcal{C} : \mathbf{Comm} \rightarrow (\mathbf{Point} \rightarrow \mathbf{Point}).$$

A structural operational semantics is known as a small-step semantics. That means each step is defined by a transition relation between configurations. A configuration is in general a pair

$$\langle C, p \rangle,$$

and it describes a simple step of execution of a command  $C$  from a state  $p$ . The transition has the form

$$\langle C, p \rangle \Rightarrow \alpha,$$

where  $\alpha$  is a configuration and it can be either:

- a new state  $p'$ , if the command  $C$  is performed in one step. A typical case is a movement in some direction in one step or the teleportation to the initial position. That means, the execution has terminated in a state  $p'$ , or
- an intermediate configuration  $\langle C', p' \rangle$ , if the execution of command  $C$  is not completed and remaining computation continues.

Assume a state  $p = (x, y)$ . The transitions for the commands executed in one step are:

$$\begin{aligned} \langle \mathbf{left}, (x, y) \rangle &\Rightarrow (x \ominus \mathbf{1}, y), \\ \langle \mathbf{right}, (x, y) \rangle &\Rightarrow (x \oplus \mathbf{1}, y), \\ \langle \mathbf{up}, (x, y) \rangle &\Rightarrow (x, y \oplus \mathbf{1}), \\ \langle \mathbf{down}, (x, y) \rangle &\Rightarrow (x, y \ominus \mathbf{1}). \end{aligned}$$

Each command executing more movements can be considered as a sequence of commands:

$$\begin{aligned} \mathbf{left} \ n &= \mathbf{left}; \mathbf{left} \ m, \\ \mathbf{right} \ n &= \mathbf{right}; \mathbf{right} \ m, \\ \mathbf{up} \ n &= \mathbf{up}; \mathbf{up} \ m, \\ \mathbf{down} \ n &= \mathbf{down}; \mathbf{down} \ m, \end{aligned}$$

where  $\llbracket m \rrbracket = \llbracket n \rrbracket \ominus \mathbf{1}$ .

The commands that are not executed in one step have transitions expressing the first step of execution:

$$\begin{aligned} \langle \mathbf{left} \ n, (x, y) \rangle &\Rightarrow \langle \mathbf{left} \ m, (x \ominus \mathbf{1}, y) \rangle, \\ \langle \mathbf{right} \ n, (x, y) \rangle &\Rightarrow \langle \mathbf{right} \ m, (x \oplus \mathbf{1}, y) \rangle, \\ \langle \mathbf{up} \ n, (x, y) \rangle &\Rightarrow \langle \mathbf{up} \ m, (x, y \oplus \mathbf{1}) \rangle, \\ \langle \mathbf{down} \ n, (x, y) \rangle &\Rightarrow \langle \mathbf{down} \ m, (x, y \ominus \mathbf{1}) \rangle, \end{aligned}$$

where  $\llbracket n \rrbracket = \llbracket m \rrbracket \oplus \mathbf{1}$ , for  $\llbracket m \rrbracket, \llbracket n \rrbracket \in \mathbb{N}_0$  and  $\llbracket \cdot \rrbracket$  is a semantic function that sends numerals to naturals (or zero) [13]:

$$\llbracket \cdot \rrbracket : \mathbf{Num} \rightarrow \mathbb{N}_0.$$

The last two commands of a language have the following transitions:

$$\begin{aligned} \langle \mathbf{skip}, p \rangle &\Rightarrow p, \\ \langle \mathbf{reset}, p \rangle &\Rightarrow p^*, \end{aligned}$$

where  $p^*$  is the starting position and its coordinates are stated in beginning of a program.

To make this language more general (closer to real languages), we change its abstract syntax by defining new commands for turning into the given direction according to the

angle. Because the robot moves on the grid, the angles are multiples of 90 degrees and they define movement in the current heading. We define the abstract syntax of this language as follows:

$$C ::= \text{forward} \mid \text{forward } n \mid \text{turn left} \mid \text{turn right} \mid \\ \text{skip} \mid \text{reset} \mid C; C.$$

These commands enable the robot to rotate around its axis. The command **forward** causes the robot moves one step in the direction of the current heading. The command **forward**  $n$  defines that robot moves  $n$  steps in the direction of the current heading. The rotation of movement defines the commands **turn left**, which changes the direction of movement to the left, and **turn right**, which changes the direction of movement to the right according to the direction of the current heading. The commands **skip** and **reset** work as in the previous version of a language. The last item in production rule  $C$ ;  $C$  is the concatenation of commands. We define a new semantic domain for the angles

$$\mathbf{Angle} = \mathbb{Z}$$

with the elements  $a \in \mathbf{Angle}$ . We consider the following convention: if the robot is facing north, then the current angle is **0**, for east it is **90**, for south **180** and for west **270**. The **forward** command calculates the next position by trigonometric functions sine and cosine. For the potential input values of the angle, these functions respectively yield values of  $-1$ , **0**, or **1**.

Because an angle is an element that provides the direction of the robot, we need to modify the notion of state. A new semantic domain **State** is a product:

$$\mathbf{State} = \mathbf{Point} \times \mathbf{Angle}$$

with the elements  $(p, a) \in \mathbf{State}$ ,  $p \in \mathbf{Point}$  and  $a \in \mathbf{Angle}$ . Then a state is a tuple and has the following form

$$\langle C, p, a \rangle.$$

Now we have two groups of commands:

- the first group contains commands **left**, **right**, **up**, **down**, **left**  $n$ , **right**  $n$ , **up**  $n$  and **down**  $n$ ;
- the second group contains the commands enabling rotation, **forward**, **forward**  $n$ , **turn left** and **turn right**;
- for the second group of statements, we need to extend (redefine) also an initial state by adding an angle,  $s^* = (p^*, \mathbf{0})$ ;
- the commands **skip**, **reset** and  $C; C$  belong to both groups of commands.

Commands from different groups cannot be combined in one program, because of different definitions of configurations. But they can be executed on the same abstract machine – for this reason we create two ANTLR grammars during implementation, so it is possible to apply one implementation of abstract machine emulation to several input language specifications.

The semantic function for this language is defined as

$$\mathcal{C}^l : \mathbf{Comm} \rightarrow \mathbf{State} \rightarrow \mathbf{State},$$



and is defined by the following transitions:

$$\begin{aligned}
\langle \mathbf{forward}, (x, y), a \rangle &\Rightarrow ((x \oplus \sin a, y \oplus \cos a), a), \\
\langle \mathbf{forward } n, (x, y), a \rangle &\Rightarrow ((x \oplus \llbracket n \rrbracket \otimes \sin a, y \oplus \llbracket n \rrbracket \otimes \cos a), a), \\
\langle \mathbf{turn left}, p, a \rangle &\Rightarrow (p, (a \oplus \mathbf{270}) \bmod \mathbf{360}), \\
\langle \mathbf{turn right}, p, a \rangle &\Rightarrow (p, (a \oplus \mathbf{90}) \bmod \mathbf{360}).
\end{aligned}$$

We also need to redefine the commands **reset** and **skip**:

$$\begin{aligned}
\langle \mathbf{reset}, (p, a) \rangle &\Rightarrow (p^*, \mathbf{0}), \\
\langle \mathbf{skip}, (p, a) \rangle &\Rightarrow (p, a).
\end{aligned}$$

Both defined languages can be extended by other commands, for instance, **init**( $x, y$ ), which can initialize the starting position of the robot. Another useful extension can be conditional command and loop command that can serve for shortening the code and opening a lot of new options. Another extension can be an addition of flags on the environment which the robot can place and lift from. Furthermore, battery or energy management can be added, so each robot's movements deplete its energy which needs to be recharged later on [13]. The mentioned extensions are the aims for future work.

### 3. Abstract Machine for DSL – Theory and Development of Method

Abstract implementation of a program is used to verify the partial correctness of the implementation of programming languages and consists of

- the definition of an abstract machine;
- the translation of the program into a sequence of the instructions of an abstract machine;
- the execution of code of abstract machine and comparison with a structural operational semantics of a program.

Definition of the abstract machine [20] consists of

- the syntax of instructions;
- the semantics of instructions.

We define the new syntactic domains **Code** for a sequence of instructions and **Ins** for the instructions. An element  $c \in \mathbf{Code}$  is a sequence of instructions. We define the abstract syntax of instructions and code by the following production rules:

$$\begin{aligned}
\mathit{ins} ::= & \mathbf{LEFT} \mid \mathbf{LEFT-}n \mid \mathbf{RIGHT} \mid \mathbf{RIGHT-}n \mid \\
& \mathbf{UP} \mid \mathbf{UP-}n \mid \mathbf{DOWN} \mid \mathbf{DOWN-}n \mid \\
& \mathbf{FORWARD} \mid \mathbf{FORWARD-}n \mid \mathbf{TLEFT} \mid \mathbf{TRIGHT} \mid \\
& \mathbf{SKIP} \mid \mathbf{RESET}, \\
c ::= & \varepsilon \mid \mathit{ins} : c.
\end{aligned}$$

Our definition of abstract machine for structural operational semantics is defined for the domain-specific language of the robot, namely for both versions of the language. The state of the abstract machine determines the position of the robot in the environment

$$p \in \mathbf{Point} = \mathbb{Z} \times \mathbb{Z}$$

and possibly an angle of its rotation

$$a \in \mathbf{Angle} = \mathbb{Z}.$$

Hence, for the first version of the language containing the directional commands, the state is an element of the semantics domain **Point**. For the second version of the language, comprising the rotations, a state is an element of the semantic domain

$$s \in \mathbf{State} = \mathbf{Point} \times \mathbf{Angle}.$$

Evaluation stack is not used in this abstract machine(s) but will be needed for possible extensions of robot language.

Semantic of instructions is defining using configurations, which can be written as triples

$$\langle c, v, s \rangle,$$

where  $c$  is the code, a sequence of instructions,  $v$  is an evaluation stack of values and  $s$  is the state of the abstract machine.

Some of the instructions contain numeric parameters which represent the number of steps in a given direction (it simply means that command is performed  $n$ -times, where  $n = \llbracket n \rrbracket$ ). For situations, when numeral  $n$  is converted to boundary values (**0**) or not supported values (negative numbers), an abstract machine (and in a simulation program, as well) does not do anything. Hence, the compilation of such commands provides an error.

From the semantic equivalence (proved for natural semantics in [36]) it follows that (for any parametric instruction, denoted **INST- $n$** ):

$$\begin{aligned} \langle \mathbf{INST-}n, v, p \rangle &= \gg \langle \mathbf{INST}; \mathbf{INST-}m, v, p \rangle \\ \langle \mathbf{INST-}1, v, p \rangle &= \gg \langle \mathbf{INST}, v, p \rangle \end{aligned}$$

for  $\llbracket n \rrbracket = \llbracket m \rrbracket \oplus \mathbf{1}$ ,  $\llbracket n \rrbracket, \llbracket m \rrbracket \in \mathbb{N}_0$ . We note that all instructions with the value **0** of the parameter don't do anything and they are (semantically) equivalent to the instruction **SKIP**.

The semantics of abstract machine instructions (for the language with directional commands) expresses how a configuration is changed after execution of them. We define the semantics of instructions as follows:

$$\begin{aligned} \langle \mathbf{LEFT} : c, \varepsilon, (x, y) \rangle &= \gg \langle c, \varepsilon, (x \ominus \mathbf{1}, y) \rangle, \\ \langle \mathbf{LEFT-}n : c, \varepsilon, (x, y) \rangle &= \gg \langle \mathbf{LEFT-}m : c, \varepsilon, (x \ominus \mathbf{1}, y) \rangle, \\ \langle \mathbf{RIGHT} : c, \varepsilon, (x, y) \rangle &= \gg \langle c, \varepsilon, (x \oplus \mathbf{1}, y) \rangle, \\ \langle \mathbf{RIGHT-}n : c, \varepsilon, (x, y) \rangle &= \gg \langle \mathbf{RIGHT-}m : c, \varepsilon, (x \oplus \mathbf{1}, y) \rangle, \\ \langle \mathbf{UP} : c, \varepsilon, (x, y) \rangle &= \gg \langle c, \varepsilon, (x, y \oplus \mathbf{1}) \rangle, \\ \langle \mathbf{UP-}n : c, \varepsilon, (x, y) \rangle &= \gg \langle \mathbf{UP-}m : c, \varepsilon, (x, y \oplus \mathbf{1}) \rangle, \\ \langle \mathbf{DOWN} : c, \varepsilon, (x, y) \rangle &= \gg \langle c, \varepsilon, (x, y \ominus \mathbf{1}) \rangle, \\ \langle \mathbf{DOWN-}n : c, \varepsilon, (x, y) \rangle &= \gg \langle \mathbf{DOWN-}m : c, \varepsilon, (x, y \ominus \mathbf{1}) \rangle, \\ \langle \mathbf{RESET} : c, \varepsilon, (x, y) \rangle &= \gg \langle c, \varepsilon, p^* \rangle, \\ \langle \mathbf{SKIP} : c, \varepsilon, (x, y) \rangle &= \gg \langle c, \varepsilon, (x, y) \rangle, \end{aligned}$$

for  $\llbracket n \rrbracket = \llbracket m \rrbracket \oplus \mathbf{1}$ ,  $\llbracket n \rrbracket, \llbracket m \rrbracket \in \mathbb{N}_0$  and  $p^*$  stands for an initial (starting) position defined by user specification. Instructions defining the movement of  $n$  steps are performed according to the principles of structural operational semantics the first movement and the remaining code contains the same instruction for  $n \ominus \mathbf{1}$  steps.

Similarly, the semantics of abstract machine instructions (for the language with rotations) is defined as follows:

$$\begin{aligned} \langle \mathbf{FORWARD} : c, \varepsilon, ((x, y), a) \rangle &\Rightarrow \langle c, \varepsilon, ((x \oplus \sin a, y \oplus \cos a), a) \rangle, \\ \langle \mathbf{TLEFT} : c, \varepsilon, ((x, y), a) \rangle &\Rightarrow \langle c, \varepsilon, (p, (a \oplus \mathbf{270}) \bmod \mathbf{360}) \rangle, \\ \langle \mathbf{TRIGHT} : c, \varepsilon, ((x, y), a) \rangle &\Rightarrow \langle c, \varepsilon, ((a \oplus \mathbf{90}) \bmod \mathbf{360}) \rangle, \\ \langle \mathbf{RESET} : c, \varepsilon, ((x, y), a) \rangle &\Rightarrow \langle c, \varepsilon, (p^*, \mathbf{0}) \rangle, \\ \langle \mathbf{SKIP} : c, \varepsilon, (p, a) \rangle &\Rightarrow \langle c, \varepsilon, (p, a) \rangle, \end{aligned}$$

The transition for moving  $n$  steps is defined similarly. We note, that in this specification, an initial configuration (state) is a tuple consisting of the position  $p^*$  and the angle  $\mathbf{0}$ . The initial value for the angle can be changed according to the user specification.

Generating of abstract machine code from the input language is done by the translation function

$$\mathcal{TC} : \mathbf{Comm} \rightarrow \mathbf{Code}$$

which sends an input source code written in the robot language into a sequence of instructions of abstract machine. Although we defined two forms of robot language, only one translation function for our purposes is defined:

$$\begin{aligned} \mathcal{TC}[\mathbf{left}] &= \mathbf{LEFT} & \mathcal{TC}[\mathbf{left } n] &= \mathbf{LEFT-}n \\ \mathcal{TC}[\mathbf{right}] &= \mathbf{RIGHT} & \mathcal{TC}[\mathbf{right } n] &= \mathbf{RIGHT-}n \\ \mathcal{TC}[\mathbf{up}] &= \mathbf{UP} & \mathcal{TC}[\mathbf{up } n] &= \mathbf{UP-}n \\ \mathcal{TC}[\mathbf{down}] &= \mathbf{DOWN} & \mathcal{TC}[\mathbf{down } n] &= \mathbf{DOWN-}n \\ \mathcal{TC}[\mathbf{turn left}] &= \mathbf{TLEFT} & \mathcal{TC}[\mathbf{turn right}] &= \mathbf{TRIGHT} \\ \mathcal{TC}[\mathbf{forward}] &= \mathbf{FORWARD} & \mathcal{TC}[\mathbf{forward } n] &= \mathbf{FORWARD-}n \\ \mathcal{TC}[\mathbf{reset}] &= \mathbf{RESET} & \mathcal{TC}[\mathbf{skip}] &= \mathbf{SKIP} \\ \mathcal{TC}[C_1; C_2] &= \mathcal{TC}[C_1] : \mathcal{TC}[C_2] \end{aligned}$$

The colon symbol “:” serves as an instruction separator delineating distinct instructions (syntactic elements) in the abstract machine instruction sequence listing.

For example, let

**forward 2; turn right; forward; turn right;  
forward 1; turn left; forward 4;**

be a correct program in robot language. A code of abstract machine for this program is the following:

**FORWARD-2 : TRIGHT : FORWARD : TRIGHT :  
FORWARD-1 : TLEFT : FORWARD-4**

As the next step, we define the meaning of the abstract machine code using a partially defined execution function:

$$\mathcal{M} : \mathbf{Code} \rightarrow \mathbf{Point} \rightarrow \mathbf{Point},$$

defined as follows:

$$\mathcal{M} \llbracket c \rrbracket p = \begin{cases} p', & \text{if } \langle c, \varepsilon, p \rangle = \gg^* \langle \varepsilon, \varepsilon, p' \rangle, \\ \perp, & \text{otherwise.} \end{cases}$$

A symbol  $\gg^*$  represents a finite number of steps in transition relation.

By composing the translation and execution functions, we get the semantic function for the abstract machine:

$$\begin{aligned} \mathcal{C} &: \mathbf{Comm} \rightarrow \mathbf{Point} \rightarrow \mathbf{Point}, \\ \mathcal{C} \llbracket c \rrbracket &= (\mathcal{M} \circ \mathcal{TC}) \llbracket c \rrbracket. \end{aligned}$$

This definition applies to the language version with directional commands. We note that the definition of the execution ( $\mathcal{M}'$ ) and semantic function ( $\mathcal{C}'$ ) for the second types of language variants (with rotations) are analogous, taking into account the semantic area:

$$\mathcal{M}' \llbracket c \rrbracket s = \begin{cases} s', & \text{if } \langle c, \varepsilon, s \rangle = \gg^* \langle \varepsilon, \varepsilon, s' \rangle, \\ \perp, & \text{otherwise.} \end{cases}$$

$$\begin{aligned} \mathcal{C}' &: \mathbf{Comm} \rightarrow \mathbf{State} \rightarrow \mathbf{State}, \\ \mathcal{C}' \llbracket c \rrbracket &= (\mathcal{M}' \circ \mathcal{TC}) \llbracket c \rrbracket. \end{aligned}$$

## 4. Application Design

We followed the idea of containerizing the application. We based on the methodological design, where we separated the individual layers (front-end, back-end) and implemented each functional unit as a separate Docker module. Docker is an open-source containerization platform which enables developers to package applications into containers and offers isolation of applications into containers which run safely [5] and independently thanks to Docker Engine directly on the host computer's operating system [27]. Docker containers also offer scalability and portability [24], so the system is easy to extend or change, and reusability of containers in other systems thereby achieving reproducible research [4]. The application consists of five components:

- *data structures* – contain data structures that are used in multiple components to avoid code repetition,
- *execution logic* – performs code simulation on an abstract machine, the output of which is the states and intermediate states of the abstract machine that occurred during execution,
- *GUI execution* – visualization of the simulation with which the user will interact,
- *translator logic* – translates the code from the language of the robot into the language of the abstract machine,
- *translator GUI* – the implementation of the page through which the user will enter input and perform the translation.

The last four components are containers and have their own *Dockerfile* defined, which downloads the openjdk version 11 image, defines a new user and working directory, copies

the JAR archive and runs it in the given container on the specified port. The database has its own container and a *Dockerfile* with a *postgres* version 13.3 image, a user and database set up and running it. For easier work with containers, a *docker-compose* file was also defined, which is common to all components with described image name, source code path, ports, container name, dependencies and environment variables. The interface between the front end and back end parts is obvious.

The components of applications are running on Spring Boot which is being used in a big amount of projects in areas like cloud computing, big data, reactive programming and client applications development [39]. Spring Boot data can be displayed on web pages using a template engine like Thymeleaf which is used for our GUI components. Visualization, HTTP requests and other logic are implemented using JavaScript, jQuery and other libraries.

For reading and transforming the user input, a technology of ANTLR has been used [23]. Three ANTLR grammar files are designed for our application for compiler and simulator logic components:

- simple directional commands (without rotations) for translator – translates input source code to abstract machine language where only **up**, **down**, **left**, **right** are present, then commands with parameter *n* as number of steps and **reset** commands are valid;
- commands with rotations for translator – translates input source code to abstract machine language where only **forward**, **forward *n***, **turn left**, **turn right** and **reset** commands are valid;
- parser for simulator – parses the abstract machine code into data structures (bytecode) on which simulation can be performed.

Finally, we give a brief overview of the technologies that were used in the development of the presented application. As a development environment, the IntelliJ IDEA environment of the Jet-Brains company with a valid university license, the Git version management tool and the university Gitlab platform, as well as the Docker Desktop container management tool, which was available for free for school non-commercial work at the time of software development, were used. Maven version 3.6.3 was used for compilation. The languages Java version 11.0.11, JavaScript, HTML together with the Thymeleaf templating tool and the Spring framework were used for the implementation, for which the IntelliJ IDEA environment provides the generation of basic source and configuration files. For parsing and translation, ANTLR 4 was used as an IntelliJ IDEA plugin, installable from the plugin store in the environment settings, the grammar files associated with them, the generated Java files, and the auxiliary “interp” and “tokens” files. Other files are css style files, icons from the Flaticon site by Freepik, Kiranshastry, Vectors Market and Chanut, and configuration files such as *Dockerfile* and *docker-compose* to define containers. The jQuery library was used to manipulate the HTML page and provide a simpler programming interface. The jsPDF JavaScript library developed by MrRio was used to create PDF documents. Creating archives in ZIP format was handled by the JSZip open source library.

## 5. Functionality of Application

An application can be accessed through a standard web browser (the application is optimized for the Mozilla Firefox browser). This allows users to work with the application

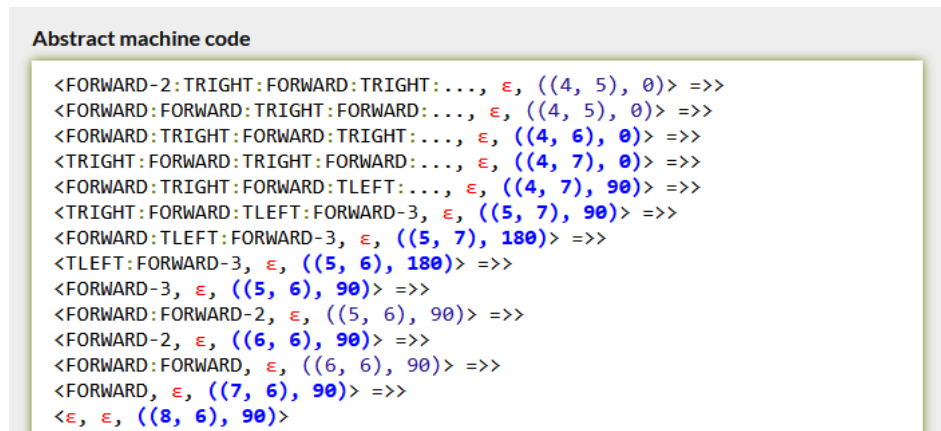
from anywhere. An application allows to insert the user's input source code or to load a code into the application from external storage. The idea is that user input is sent to the back-end layer and compiled. If no error occurs, the source is correct and compiled bytecode is sent back for performing the visualization.

The user interface helps the user to enter the input source by providing also hints for writing the code in the robot language. The hint disappears when the user starts to write the code. Two main functionalities are available – simulation (visual computation) of abstract machine and compilation (translation) of input source to internal bytecode (a data structure according to the specification).

For the implementation of the compilation module, the ANTLR tool was used. The compiler reads an input source and identifies in which variant of language the program is written:

- if code contains rotation commands, appropriate grammar is applied;
- if code does not contain rotation commands, only directional-stepping commands, then grammar for this language, is applied;
- otherwise, no code is generated and the program provides error information (since the languages cannot be mixed).

If the commands are not mixed, but during the compilation, the lexical or syntactic error occurs, the compilation does not provide a bytecode, only information about the compilation error. Then, abstract machine code is annotated by adding HTML tags and sent back to the front end where all errors are highlighted.



The screenshot shows a window titled "Abstract machine code" containing a list of commands. Each command is enclosed in angle brackets and followed by an epsilon symbol and a coordinate pair. Some numbers in the coordinate pairs are highlighted in red, indicating errors. The commands are as follows:

```

<FORWARD-2:TRIGHT:FORWARD:TRIGHT:..., ε, ((4, 5), 0)> =>>
<FORWARD:FORWARD:TRIGHT:FORWARD:..., ε, ((4, 5), 0)> =>>
<FORWARD:TRIGHT:FORWARD:TRIGHT:..., ε, ((4, 6), 0)> =>>
<TRIGHT:FORWARD:TRIGHT:FORWARD:..., ε, ((4, 7), 0)> =>>
<FORWARD:TRIGHT:FORWARD:TLEFT:..., ε, ((4, 7), 90)> =>>
<TRIGHT:FORWARD:TLEFT:FORWARD-3, ε, ((5, 7), 90)> =>>
<FORWARD:TLEFT:FORWARD-3, ε, ((5, 7), 180)> =>>
<TLEFT:FORWARD-3, ε, ((5, 6), 180)> =>>
<FORWARD-3, ε, ((5, 6), 90)> =>>
<FORWARD:FORWARD-2, ε, ((5, 6), 90)> =>>
<FORWARD-2, ε, ((6, 6), 90)> =>>
<FORWARD:FORWARD, ε, ((6, 6), 90)> =>>
<FORWARD, ε, ((7, 6), 90)> =>>
<ε, ε, ((8, 6), 90)>

```

**Fig. 2.** A computational sequence of an abstract machine code (cropped screenshot)

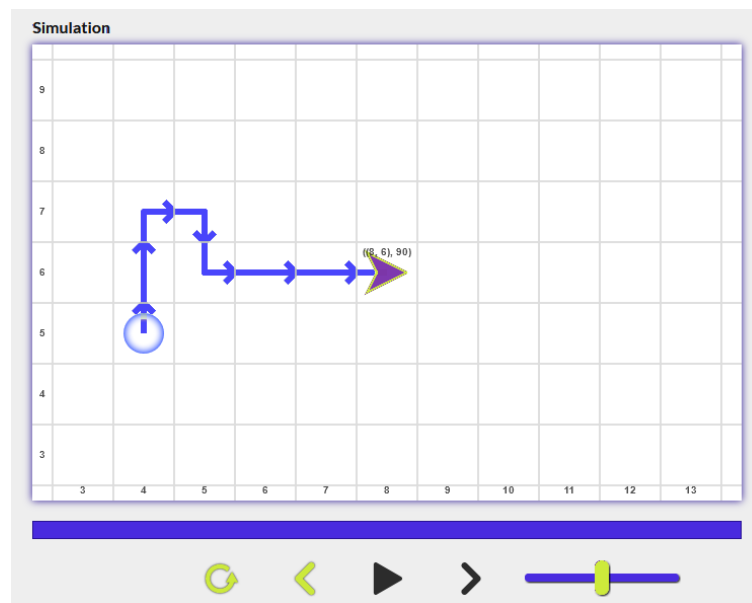
The successful compilation, if no error appears, provides a bytecode for abstract machine computation steps and visualization. The result of the compilation is then stored in an internal database and sent back to the GUI component.

The main screen of the application contains two visual areas. An abstract machine code is displayed on the right-hand side of an application window (Fig. 2). The application

allows also to save the source code. The user interface supports being displayed in English and Slovak languages, a color theme can be changed from light to dark, as well.

By clicking on the execution button, the user is redirected to the simulator page with the id of the latest translation request. On the simulator web page, the given id is sent to the back end and then to the simulator logic. There the id is searched in the database for a given request. After finding the request, the abstract machine code is parsed according to the third grammar to a list of commands from which the initial configuration is created and execution of these commands is started. The result of an execution is a list of states or rather configurations of the abstract machine and this list is added to the execution request which is sent back to the user and front-end.

On the left-hand side, there is the visualization canvas showing the robot's position and rotation in an orthogonal grid (Fig. 3).



**Fig. 3.** Visualization canvas

Below (under the canvas area) is the simulation controller with control buttons (play, pause, step back, step forward and reset buttons) and a simulation speed slider. On the right-hand side, there are transitions of configurations from the initial configuration to the current configuration. On the top, in the middle, a currently performed command from the translated abstract machine code is highlighted. The user can download a record of the simulation in four different formats:

- standard CSV format with position, rotation, value and code stacks for each configuration in time,
- XML format with structured data similar to CSV data,

- PDF file with a title, current time, a snapshot of the canvas and transitions of configurations which are generated using jsPDF library by MrRio<sup>1</sup>,
- a TeXsource format and picture with a snapshot of the canvas which looks similar to the PDF and is stored in a ZIP archive using JSZip<sup>2</sup>.

Visualization is performed using JavaScript following states or configurations from simulator logic to move the robot in its environment and draw its path.

## 6. Conclusion

In this paper, we presented our approach to define structural operational semantics and its abstract implementation for selected domain-specific language describing the controlling of robot. We developed and defined our semantic approach for both versions of the language for natural semantics in [36]. For the existing approach in structural operational semantics (defined in [13]), we defined our approach of abstract implementation. The abstract implementation consists of a definition of an abstract machine and from translation function that transforms each command of robot language into a sequence of abstract machine instructions. We defined one abstract machine consisting of instructions suitable for both versions of the robot language, together with their semantics expressing changing configurations during instruction execution.

For this semantic approach, we also developed a visualizing software that provides a compilation of input source code written in robot language and emulates the calculation of abstract machine for both versions of the language, and it accepts direct input in abstract machine code and provides also the calculations and emulations.

The design of the micro-service system is created by using containerization, so each micro-service can be reused in other systems later (if we assume the future work is oriented to a complex software visualizing environment).

The created application offers many possible extensions, also thanks to the modular approach applied by containerization, and improvements. The list of commands can be extended in future based on a new specification. Such developed application is ready to be integrated into the teaching process for the courses oriented to formal semantics, and (possibly) formal languages. Its added value is a significant degree of interactivity, clarity and illustrativeness and, above all, the possibility to use the application in the process of present and distance teaching as well as during the independent preparation of students.

Because we see the potential for expanding and consolidating formal methods for software engineering, mainly because all formal methods are based on formal semantics, we will focus our research on semantic methods with the possibility of integrating the results into practice and into teaching young IT specialists and experts.

**Acknowledgments.** This work was supported by the project 030TUKÉ-4/2023 – “Application of new principles in the education of IT specialists in the field of formal languages and compilers”, granted by Cultural and Education Grant Agency of the Slovak Ministry of Education *and* in the frame of the initiative project “Semantics-Based Rapid Prototyping of Domain-Specific Languages” under the bilateral program “Aktion Österreich-Slowakei, Wissenschafts- und Erziehungskooperation” granted by Slovak Academic Information Agency.

<sup>1</sup> <https://mrrio.github.io/jsPDF/>

<sup>2</sup> <https://github.com/Stuk/jszip>



The authors express their gratitude to Prof. Marjan Mernik for the original idea of how to formulate a domain-specific language for a given application domain. The authors also would like to thank Dániel Horpácsi and Judit Horpácsiné Kőszegi for their approach to DSL semantics for the robot that motivated us in our research.

## References

1. Alam, M.T., Chowdhury, S., Halder, R., Maiti, A.: Blockchain domain-specific languages: Survey, classification, and comparison. In: 2021 IEEE International Conference on Blockchain (Blockchain). pp. 499–504 (2021)
2. Alam T., e.a.: Blockchain domain-specific languages: survey, classification, and comparison. 2021 IEEE Int.Conf. on Blockchain p. 499–504 (2022)
3. Arusoaie, A.: Certifying findel derivatives for blockchain. *Journal of Logical and Algebraic Methods in Programming* 121, 100665 (2021), <https://www.sciencedirect.com/science/article/pii/S2352220821000286>
4. Chamberlain, R., Schommer, J.: Using docker to support reproducible research. DOI: <https://doi.org/10.6084/m9.figshare.1101910>, 44 (2014)
5. Combe, T., Martin, A., Di Pietro, R.: To docker or not to docker: A security perspective. *IEEE Cloud Computing* 3(5), 54–62 (2016)
6. Demirkol, S., Challenger, M., Getir, S., Kosar, T., Kardas, G., Mernik, M.: A DSL for the development of software agents working within a semantic web environment. *Computer Science and Information Systems* 10(4), 1525–1556 (2013), <https://dk.um.si/IzpisGradiva.php?lang=eng&id=66708>
7. Deursen, A., Klint, P.: Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology* (2002)
8. Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. *SIGPLAN Notices* 35, 26–36 (01 2000)
9. Dornelles, E., Parahyba, F., Frantz, R., Roos-Frantz, F., Reina-Quintero, A., Molina-Jiménez, C., Bocanegra, J., Sawicki, S.: Advances in a dsl to specify smart contracts for application integration processes. In: *Anais do XXV Congresso Ibero-Americano em Engenharia de Software*. pp. 46–60. SBC, Porto Alegre, RS, Brasil (2022), <https://sol.sbc.org.br/index.php/cibse/article/view/20962>
10. Fowler, M.: *Domain-specific languages*. Pearson Education (2010)
11. Frantz, C., Nowostawski, M.: From institutions to code: Towards automated generation of smart contracts. pp. 210–215 (09 2016)
12. Hamdaqa, M., Met, L.A.P., Qasse, I.: iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Information and Software Technology* 144, 106762 (2022)
13. Horpácsi, D., Kőszegi, J.: Formal semantics. [https://regi.tankonyvtar.hu/en/tartalom/tamop412A/2011-0052\\_05\\_formal\\_semantics/index.html](https://regi.tankonyvtar.hu/en/tartalom/tamop412A/2011-0052_05_formal_semantics/index.html) (2014), accessed: Dec 14th, 2020
14. Johanson, A.N., Hasselbring, W.: Hierarchical combination of internal and external domain-specific languages for scientific computing. In: *Proceedings of the 2014 European conference on software architecture workshops*. pp. 1–8 (2014)
15. Kapferer., S., Zimmermann., O.: Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. In: *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*. pp. 299–306. INSTICC, SciTePress (2020)
16. Kollár, J., Porubán, J., Chodarev, S.: Modelovanie a generovanie softvérových architektúr [Modeling and generation of software architectures]. *elfa s.r.o* (2012), (*in Slovak*)

17. Lamela Seijas, P., Nemish, A., Smith, D., Thompson, S.: Marlowe: Implementing and analysing financial contracts on blockchain. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) *Financial Cryptography and Data Security*. pp. 496–511. Springer International Publishing, Cham (2020)
18. Luković, I., Pereira, M.J.V., Oliveira, N., Cruz, D.d., Henriques, P.R.: A DSL for PIM Specifications: Design and Attribute Grammar based Implementation. *Computer Science and Information Systems* 8(2), 379–403 (2011)
19. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 316– (12 2005)
20. Nielson, Riis, H., Nielson, F.: *Semantics with Applications: An Appetizer*. Springer Science & Business Media (2007)
21. Oei, R., Coblenz, M.J., Aldrich, J.: Psamathe: A DSL with flows for safe blockchain assets. *CoRR abs/2010.04800* (2020), <https://arxiv.org/abs/2010.04800>
22. Parity Technologies: Ink! (2020), <https://github.com/paritytech/ink>, [Online; accessed February 2024]
23. Parr, T.: *The Definitive ANTLR 4 Reference*. Pragmatic Programmers, LLC, The, Raleigh (2013)
24. Patil, S.: Study of container technology with docker. In: *International Journal of Advanced Research in Science, Communication and Technology* pp. 504–509 (2021)
25. Pereira, M., Mernik, M., Cruz, D., Rangel Henriques, P.: Program comprehension for domain-specific languages. *Comput. Sci. Inf. Syst.* 5, 1–17 (12 2008)
26. Popovic, A., Luković, I., Dimitrieski, V., Djukic, V.: A dsl for modeling application-specific functionalities of business applications. *Computer Languages, Systems & Structures* 43 (05 2015)
27. Rad, B.B., Bhatti, H.J., Ahmadi, M.: An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)* 17(3), 228 (2017)
28. Radaković, D., Herceg, D.: Towards a completely extensible dynamic geometry software with metadata. *Computer Languages, Systems & Structures* 52, 1–20 (2018)
29. Roche, N., Hernández, W., Chen, E., Siméon, J., Selman, D.: Ergo - a programming language for smart legal contracts. *CoRR abs/2112.07064* (2021), <https://arxiv.org/abs/2112.07064>
30. Schreiner, W.: Theorem and algorithm checking for courses on logic and formal methods. In: Quaresma, P., Neuper, W. (eds.) *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018*. EPTCS, vol. 290, pp. 56–75 (2018), <https://doi.org/10.4204/EPTCS.290.5>
31. Schreiner, W., Steingartner, W.: *The SLANG Semantics-Based Language-Generator — Tutorial and Reference Manual (Version 1.0.\*)*. Technical report 23-13, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria (September 2023), [doi:10.35011/risc.23-13](https://doi.org/10.35011/risc.23-13)
32. Seidametova, Z.: Some methods for improving data structure teaching efficiency. *Educational Dimension* 58, 164–175 (Jun 2022), <https://journal.kdpu.edu.ua/ped/article/view/4509>
33. Steingartner, W.: Compiler module of abstract machine code for formal semantics course. In: *2021 IEEE 19th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*. pp. 000193–000200 (2021)
34. Steingartner, W.: On some innovations in teaching the formal semantics using software tools. *Open Computer Science* 11(1), 2–11 (2021), <https://doi.org/10.1515/comp-2020-0130>
35. Steingartner, W., Gajdoš, E.: The visualization of a graph semantics of imperative language. *Polytechnica: Journal of Technology Education* 2(5), 7–14 (2021), <https://doi.org/10.36978/cte.5.2.1>

36. Steingartner, W., Novitzká, V.: Natural semantics for domain-specific language. In: New Trends in Database and Information Systems. pp. 181–192. Springer International Publishing, Cham (2021)
37. Steingartner, W., Novitzká, V., Schreiner, W.: Proof of Equivalence of Semantic Methods for a Selected Domain-Specific Language. *Journal of Applied Mathematics and Computational Mechanics* 23(2) (2024)
38. Steingartner, W., Zsiga, R., Radaković, D.: Natural semantics visualization for domain-specific language. In: 2022 IEEE 16th International Scientific Conference on Informatics (Informatics). pp. 293–298 (2022)
39. Walls, C.: *Spring Boot in action*. Manning Publications (2016)
40. Wohrer, M., Zdun, U.: From domain-specific language to code: Smart contracts and the application of design patterns. *IEEE Software* 37(5), 37–42 (2020)
41. Wood, G.: Ethereum: A secure decentralized generalized transaction ledger. *Ethereum project yellow paper* 151(2014), 1–32 (2014)
42. Wood, G.: Polkadot: Vision for a heterogeneous multi-chain framework (2016), 21(2327): 4662

**William Steingartner** works as Associate Professor of Computer Science at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. He defended his PhD thesis "The Rôle of Toposes in Computer Science" in 2008. His main fields of research are semantics of programming languages, category theory, compilers, data structures and recursion theory. He also works with cybersecurity and software engineering.

**Valerie Novitzká** works as a Full Professor of Informatics at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. Her fields of research include semantics of programming languages, non-classical logical systems and their applications in computing science. She also works with type theory and behavioural modeling of large program systems based on categories.

*Received: December 01, 2023; Accepted: April 28, 2024.*

