Property-Driven Design and Executable Semantics of a Mission-Critical Domain-Specific Language

William Steingartner[1,2,*,†], Davorka Radaković[3,*,†], Samuel Novotný[1,†] and Monika Kohútová[1,†]

[1]Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, Košice, Slovakia
[2]Slovak Society for Applied Cybernetics and Informatics, Slovakia
[3]University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics, Division of Informatics, Novi Sad, Serbia

Abstract

Autonomous space missions place high demands on the reliability, safety, and predictability of software systems. In an environment where operator intervention is not possible, it is essential that task control sequences are correct and formally verifiable. In this paper, we present a simple domain-specific language (DSL) for space mission planning, designed using property-driven design principles and defined by denotational semantics. The language is minimalistic, but explicitly constructed to embed safety and correctness properties (e.g., no transmission without an activated antenna) directly in the semantics. Each command has a formally defined meaning as a state transformer that either produces a resulting state or signals an undefined outcome when a semantic precondition is violated. The language does not contain dynamic branching or recursion, which greatly simplifies static analysis. The proposed language and its semantics represent a foundation that can be extended with types, conditions, time constraints, and other features. The aim of further research is also to evaluate the proposed approach in educational and industrial contexts and to explore its use within verification and synthesis tools.

Keywords

denotational semantics, domain-specific language, executable semantics, mission-critical DSL, property-driven design, SLANG

1. Introduction

Nowadays, despite the constant technological progress and the dominance of general-purpose programming languages (GPLs), domain-specific languages (DSLs) continue to maintain their justified place. Their goal is not to compete with the universality of GPL languages, but to effectively solve tasks in narrowly defined domains. It is precisely in areas where special demands are made on reliability, predictability and analyzability – such as planning autonomous or safety-critical missions – that DSL languages prove to be significantly more suitable.

General-purpose programming languages such as C, Python or Rust are Turing-complete and allow the expression of any calculation. However, from the point of view of verification, this is a disadvantage: there is no general algorithm that would guarantee the security properties of a program written in this way. Moreover, GPL languages typically do not enforce domain restrictions. The language does not contain information about what is an admissible or safe sequence of commands in a given domain. For

SQAMIA 2026: 13th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 16–18, 2026, Niš, Serbia

*Corresponding author.
†These authors contributed equally.
william.steingartner@tuke.sk (W. Steingartner); davorkar@dmi.uns.ac.rs (D. Radaković); samuel.novotny@tuke.sk (S. Novotný); monika.kohutova@tuke.sk (M. Kohútová)
https://kpi.fei.tuke.sk/sk/person/william-steingartner (W. Steingartner);
https://informatika.pmf.uns.ac.rs/o-nama/imenik/davorkar/ (D. Radaković);
https://kpi.fei.tuke.sk/sk/person/samuel-novotny (S. Novotný); https://kpi.fei.tuke.sk/en/user/11335 (M. Kohútová)
0000-0002-2852-9403 (W. Steingartner); 0000-0001-8480-3211 (D. Radaković); 0009-0005-1675-2965 (S. Novotný); 0009-0003-6694-2551 (M. Kohútová)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-19

CEUR-WS.org/Vol-4336/paper16.pdf

example, it is technically possible to transmit data before the antenna has been deployed – an error that may only become apparent during simulation or real-world operation.

For these reasons, the use of formal specifications and so-called property-driven design is crucial. In formal specifications, both the syntax and semantics of the language are defined mathematically precisely, without ambiguities or undefined behavior. Property-driven design means that the language itself is designed to prevent the emergence of undesirable states – for example, the transmit command will be semantically undefined ($\perp$) unless the deploy operation has been performed beforehand. Security properties are thus guaranteed by the construction of the language itself.

The target domain is command scripts for autonomous Earth observation satellites, planning of planetary rovers task sequences, and robotic missions in confined or simulated environments. The proposed core serves as a basis for further development: language extensions, runtime verification, or secure code generation.

In this article, we present the design and formal specification of a simple domain-specific language for mission planning. The goal is to demonstrate that, using denotational semantics, it is possible to define a language whose commands are meaningful only in permissible states, thus ensuring correct execution already at the design level. The proposed language is conceived as a didactic core, suitable for further expansion – whether with verification mechanisms, code generation, or mission simulation. Part of our methodology is also its implementation in the Semantics-Based Language Generator (SLANG) [1] tool, which allows automatic generation of executable semantics from the specification. In this way, the DSL becomes not only analyzable, but also a directly executable prototype, suitable for development, testing, and mission verification.

The structure of the paper is as follows. Next section discusses related work and comparable approaches. In Section 3, we define the syntax of the proposed domain-specific language and introduce its denotational interpretation. Section 4 provides a brief overview of property-driven design principles and their relevance in the context of our approach. The core contribution of the paper is presented in Section 5, where we define the denotational semantics of the language in a precise and modular way. Section 6 describes the executable prototype and the underlying execution model. A case study illustrating both a correct and an incorrect usage scenario is provided in Section 7. Finally, Section 8 concludes the paper and outlines directions for future work.

2. Related Work

Domain-specific languages (DSLs) are widely used in mission-critical systems to enable domain experts to express commands and plans in compact, high-level form. Several DSLs have been designed specifically for space and robotics applications.

PLEXIL [2] is a programming language (a formal DSL) for expressing plans. It is based on the Linda programming language and is designed to be more user-friendly and easier to learn than other robotic programming languages. PLEXIL was developed by NASA Ames Research Center and is used by a variety of companies and organizations, including iRobot and the United States Military. It has an XML-like syntax and precise operational semantics [3] but remains complex and difficult to analyze statically due to dynamic constructs.

Copilot [4] is a stream-based DSL (a runtime verification framework) for real-time embedded monitoring. Copilot monitors are written in a compositional, stream-based language with support for a variety of Temporal Logics (TL), which results in robust, high-level specifications that are easier to understand than their traditional counterparts. The framework translates monitor specifications into C code with static memory requirements, which can be compiled to run on embedded hardware [5]. While it offers a formal semantics and verification support, it is geared toward reactive monitoring rather than command sequencing.

FRET [6] supports (restricted) natural language-based formalization of system requirements and contracts, allowing high-level specification but not low-level execution semantics. These requirements are automatically provided with temporal logic semantics that is more amenable to analysis and

verification than the natural language requirements would be [7].

The Core Flight System (cFS) by NASA is a platform-independent, reusable software framework designed to support reuse and portability [8]. In contrast to the approach proposed here, mission-specific command sequences used within such a framework are not necessarily accompanied by an explicit denotational semantics, which can make formal reasoning about their behavior more difficult.

Executable Semantics Tools like Ott [9], K Framework [10], and PLT Redex [11] support executable specification of programming languages. However, they are general-purpose and not tailored to the lightweight modeling needs of safety-critical DSLs. The SLANG tool [1] strikes a balance between expressiveness and tractability, making it well-suited for prototyping semantically well-founded DSLs.

In the area of model-based development of safety-critical systems, SCADE [12] can be mentioned as a well-established industrial approach. SCADE represents a proven industrial approach to the development of safety-critical embedded software based on models. It is based on the Lustre synchronous data flow paradigm and combines formally defined computational models with graphical modeling, verification and generation of qualified code. In this regard, SCADE demonstrates how a rigorous semantic foundation can support the transition from high-level system models to executable safety-critical software. While SCADE focuses on the development of reactive embedded control systems within a mature industrial tool chain, our work explores a lightweight domain-specific approach in which mission-level constructs and their properties are explicitly defined through executable formal semantics.

A well-designed DSL can prevent dangerous or ambiguous programs by design. This eliminates entire classes of errors (e.g., “transmit before deploy”) before they are even executed. Since domain-specific languages are essentially small languages (designed for a specific domain without the ambition of replacing general-purpose programming languages), they are characterized by a limited syntax. This naturally results in simpler semantics and at the same time the possibility of implementing a parser with lower complexity [13].

DSLs also allow static analysis and property verification without the need for a full model checker [14, 15, 16]. Thanks to prototyping tools such as the Semantics-Based Language Generator (SLANG) [1], it is possible to quickly create and run simulations for DSLs with precisely defined semantics. This method of rapid prototyping eliminates the need to manually develop interpreters for each mission. This does not only apply to coding itself – interpreter development often involves a whole range of related processes, including architecture design, testing, integration and security certification.

When designing a DSL, correctness can be ensured relatively easily at the construction level (so-called correct-by-construction [17]). This means that the language design itself can guarantee that certain types of errors cannot occur at all. The semantics of the language must be precisely and correctly defined and should also cover boundary and error cases [18, 19], for example, violation of invariants should be modeled by returning the distinguished undefined value $s_\perp$ [20].

Finally, DSLs also have significant pedagogical and training value. They allow engineers to understand exactly what the system will do, without needing to know the details of the implementation or the general programming language. They can therefore be successfully used in training new teams, in verifying plans or in simulations. This trend has led university curricula to increasingly emphasize domain-specific languages, with particular attention to formal specification, principled syntax design, sound semantic definition, and interpreter construction. It goes without saying that syntax determines only the form of the program notation — but the meaning of the program, its behavior and credibility lie in the precise semantics. This should be given either as a semantic function (denotational approach), a transition relation (operational approach) or a system of preconditions and postconditions (axiomatic approach), depending on the choice of semantic model [18].

Space missions require highly reliable, predictable, and autonomous software systems capable of operating in inaccessible environments. Engineers must specify complex plans at the task sequencing level using commands that directly correspond to the operational domain — such as aiming at a target, activating sensors, collecting and transmitting data, and working with limited onboard resources.

General-purpose programming languages (GPLs) are often overly expressive for this purpose and therefore more prone to errors [13]. Their unrestricted control flow, shared state, and complex structures

make verification and analysis difficult. In contrast, domain-specific languages (DSLs) allow the design of constructs that correspond to specific domain concepts, limit expressiveness, and enforce desired properties and constraints directly in the language.

DSLs designed in this way simplify program analysis and verification, support safe and deterministic execution, and eliminate ambiguous or potentially dangerous constructs (e.g., transmitting before antenna deployment). Modern missions use autonomous task-specific command sequences – typically coded in simple scripting languages for satellites, planetary rovers or robotic probes. However, these languages often suffer from fundamental limitations:

- they lack formal semantics,
- they are focused only on execution, not on verification or correctness of the design,
- they are not easily simulated or verified due to dynamic features or lack of modularity.

In response to these challenges, we propose a minimal core DSL for mission planning that integrates semantic constraints into the language design and assigns each well-formed program an unambiguous mathematical meaning through denotational semantics. Our approach differs in its explicit *property-driven language design*, where the semantic model is deliberately crafted to enforce determinism, safety, and analyzability (P1–P4). Unlike prior work, we emphasize simplicity, executability, and pedagogical clarity as first-class goals.

3. Syntactic Structure and Semantic Foundations

In this section, we define the abstract syntax of our DSL and introduce the semantic domains and intuition behind its formal interpretation. These foundations prepare for the denotational semantics defined in Section 5.

The abstract syntax of the core language is defined in BNF as follows:

$$\begin{aligned} cmd ::= &\texttt{orient}(t) \mid \texttt{deploy} \mid \texttt{capture}(s) \mid \texttt{transmit}(d) \\ &\mid \texttt{wait}(\tau) \mid \texttt{repeat}(n, cmd) \mid cmd; cmd, \end{aligned}$$

We assume the following syntactic domains:

$t \in \mathit{Target}$	— symbolic name or identifier of a target (e.g., location, object).
$s \in \mathit{Sensor}$	— symbolic sensor identifier (e.g., `cam1`, `spec`).
$d \in \mathit{Datum}$	— data item or product returned by a sensor,
$\tau \in \mathit{Num}$	— numeric literal used as a waiting duration.
$n \in \mathit{Num}$	— numeric literal used as a repetition bound.

We take Num to be the set of non-negative integer literals. The non-terminal categories Target, Sensor, Datum, and Num represent atomic syntactic values such as string identifiers or numeric constants. Their concrete representation (e.g., identifiers, strings, integers) depends on the implementation but remains fixed throughout semantic evaluation.

Individual commands have their own (intuitive) meaning:

- `orient(t)`: rotate the spacecraft toward target t,
- `deploy`: deploy the antenna (a prerequisite for transmission),
- `capture(s)`: use sensor s to collect data,
- `transmit(d)`: send data d if the antenna is deployed,
- `wait`(τ): pause execution for τ time units,
- `repeat(n, c)`: repeat command c exactly n times,
- $\texttt{c}_1; \texttt{c}_2$: execute c_1 followed by c_2,

DSL syntax is intentionally limited to facilitate formal semantics, static reasoning, and predictable execution. These design choices enforce the following properties:

- *A small, syntax-directed set of command forms*, facilitating structural analysis.
- *Compositionality*, ensuring that the semantics of a compound command is derived solely from the semantics of its constituents.
- *Structured control flow*, excluding dynamic features such as unstructured jumps or exceptions.
- *Statically bounded iteration*, guaranteeing termination properties at the syntactic level.

We note that compositionality is a fundamental requirement for denotational semantics [21], the exclusion of unstructured control flow follows established principles of structured programming [22], and restricting expressiveness to support analysis is a common design principle in domain-specific languages [23]. This syntax supports properties P1–P4 introduced in Section 4 and is amenable to denotational interpretation (Section 5), simulation (Section 7), and comparison with related approaches (Section 2).

4. Property-Driven Design Principles

Property-driven design helps to minimize the risk of developing a system that does not satisfy the required properties, and to promote the reuse of hardware independent models [24, 25]. Property-driven design consists of four phases. The first phase of property-driven design consists in formally specifying the requirements of the system by stating its desired properties. In the second phase, a model of the system is created. In the third phase, the developer can use the model to guide the process of implementing the simulation. The last phase consists in deploying the system [26].

The design of our domain-specific language is guided by a set of key properties that are essential for safety, predictability, and formal verifiability in mission-critical software systems. Each property informs not only the semantics of individual commands, but also the structure and constraints of the language as a whole.

P1: Determinism. For any given initial state, the evaluation of a well-formed command yields at most one resulting state. That is, the semantic function $[\![\cdot]\!] : \mathit{Cmd} \to (\mathit{State} \to \mathit{State}_\perp)$ is a total function into the lifted state domain and never introduces nondeterminism.

P2: Safety Preconditions. Commands may have semantic preconditions that are encoded explicitly in their semantic definitions. If a precondition is not satisfied in the current state, evaluation returns $\perp$. This enforces safety by construction. For example, `transmit(d)` requires that the antenna be deployed ($\sigma.\mathit{antenna} = \mathbf{1}$).

P3: Compositionality. The semantics of compound constructs (e.g., sequences, loops) is defined strictly in terms of the semantics of their subcomponents. This enables reasoning and verification to scale modularly and supports interpreter reuse.

P4: Static Checkability. The language is designed so that violations of safety properties (such as P2) can potentially be detected without execution. This includes explicit constructs with predictable behavior, avoidance of dynamic control structures (e.g., unstructured jumps), and a small, syntax-directed command grammar that facilitates static analysis.

These properties are embedded into the denotational semantics defined in Section 5 and demonstrated in the case study of Section 7. In particular, P2 is enforced by returning $\perp$ when a semantic precondition is violated. P1 and P3 follow from the functional and compositional nature of the semantics. P4 is enabled by the simplicity and structure of the syntax and command forms.

5. Denotational Semantics

Denotational semantics is one of the traditional semantic methods [21, 27, 28, 29]. Its principle is that for a given construct we find a (mathematical) function that provides the same results as the running program, regardless of the actual order or repetition of instructions within the given construct [18]. Due to its exactness and mathematical precision, we have chosen it as the starting method in our approach.

We define the semantics of our domain-specific language using a denotational approach, where the meaning of a command is given as a mathematical function from input state to output state. This choice ensures compositionality, determinism, and forms a basis for later tool implementation.

5.1. Denotational Interpretation

Each command in the DSL is interpreted as a state transformer. We define the semantic function:

$$[\![\cdot]\!] \colon \mathit{Cmd} \longrightarrow (\mathit{State} \longrightarrow \mathit{State}_{\perp}).$$

For a command c and an input state σ, the expression $[\![c]\!](\sigma)$ denotes the resulting state, or $\perp$ if a semantic precondition is violated. Thus, each command denotes a total function into the lifted domain $\mathit{State}_{\perp}$, while invalid command executions are represented explicitly by $\perp$.

The function is defined compositionally: for compound commands such as sequencing and `repeat`, the meaning is built from the meanings of their subcommands. Primitive operations such as `deploy` and `capture` act as atomic state transformations. The semantic equations for all command forms are given in Section 5.3.

5.2. State Domain

We define an abstract domain of program states capturing the relevant aspects of mission execution.

The program state is modeled as a (simplified) record:

$$\mathit{State} = \{\mathit{orient} : \mathit{Target}, \quad \mathit{antenna} : \{\mathbf{0}, \mathbf{1}\}, \quad \mathit{data_ready} : \mathbb{B}, \quad \mathit{log} : \mathit{List}[\mathit{Datum}]\}$$

where:

- Target is the set of possible target orientations,
- $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ is the Boolean domain,
- log stores, in order, the data items that have already been transmitted (e.g. by sensors).

Each element of log is of type Datum, corresponding to a syntactic value from the domain $d \in \mathit{Datum}$.

To account for semantic errors (e.g., violation of command preconditions), we extend the state domain with a distinguished undefined element:

$$\mathit{State}_{\perp} = \mathit{State} \cup \{\perp\}.$$

The value $\perp$ represents an undefined or erroneous semantic result. We denote by $\mathcal{S}$ the set of all semantic results, i.e., $\mathit{State}_{\perp}$.

5.3. Semantic Function

We now define the meaning of each command form. Let $\sigma \in \mathit{State}$ denote an input state. The equations below use the semantic function introduced above; consequently, command syntax and semantic state variables are kept distinct.

Orient to target

$$[\![\texttt{orient}(t)]\!] = \lambda\sigma.\ \sigma[\mathit{orient} \mapsto t]$$

Deploy antenna

$$[\![\texttt{deploy}]\!] = \lambda\sigma.\ \sigma[\mathit{antenna} \mapsto \mathbf{1}]$$

Transmit data For $\ell \in \mathit{List}[\mathit{Datum}]$ and $d \in \mathit{Datum}$, let $\mathit{append}(\ell, d)$ denote the list obtained by appending d to the end of ℓ, while preserving all existing elements. Then

$$[\![\texttt{transmit}(d)]\!] = \lambda\sigma.\ \begin{cases} \sigma[\mathit{log} \mapsto \mathit{append}(\sigma.\mathit{log}, d)], & \text{if } \sigma.\mathit{antenna} = \mathbf{1}, \\ \bot, & \text{otherwise.} \end{cases}$$

Sequence of commands

$$[\![c_1; c_2]\!] = \lambda\sigma.\ \begin{cases} [\![c_2]\!](\sigma'), & \text{if } [\![c_1]\!](\sigma) = \sigma' \text{ for some } \sigma' \in \mathit{State}, \\ \bot, & \text{if } [\![c_1]\!](\sigma) = \bot. \end{cases}$$

Repeat n times

$$[\![\texttt{repeat}(0, c)]\!] = \lambda\sigma.\ \sigma$$

$$[\![\texttt{repeat}(n, c)]\!] = \lambda\sigma.\ \begin{cases} [\![\texttt{repeat}(n{-}1, c)]\!](\sigma'), & \text{if } [\![c]\!](\sigma) = \sigma' \text{ for some } \sigma' \in \mathit{State}, \\ \bot, & \text{if } [\![c]\!](\sigma) = \bot, \end{cases} \qquad n > 0.$$

The remaining primitive constructs are defined analogously:

Capture sensor data

$$[\![\texttt{capture}(s)]\!] = \lambda\sigma.\ \sigma[\mathit{data_ready} \mapsto \mathbf{true}]$$

We assume that a successful capture operation has no side effect except marking the `data_ready` flag as true.

Wait for time units

$$[\![\texttt{wait}(\tau)]\!] = \lambda\sigma.\ \sigma$$

We do not model time passage explicitly in the state. For now, the `wait`(τ) operation acts as a semantic no-op.

This approach supports formal reasoning, validation, and simulation. It also allows for correct-by-construction tools to reject programs that would otherwise be invalid or dangerous. A more refined model would include a temporal field in the state and advance a clock or consume execution budget. We leave this extension for future work.

6. Prototype and Execution Model

While the denotational semantics provides a mathematical foundation for the meaning of programs, our long-term goal is to derive executable interpreters and analyzers directly from these definitions. To support this, we have initiated a prototype implementation of the language semantics using the SLANG toolset [1], which supports structured semantic definitions and evaluation models.

6.1. Executable Semantics in SLANG

In SLANG we formalize the denotational semantics of the language in a compositional and executable manner. SLANG allows language designers to:

- specify abstract syntax in a declarative form,
- define semantic domains and functions in a strongly-typed, modular framework,
- execute and test semantic definitions directly,

- extend models with static analysis and type systems.

Based on the formal language specification, SLANG automatically generates:

- Parser – converts the text representation of the program into an abstract syntax tree (AST) using the ANTLR4 tool,
- Printer – creates a text representation of the program from the AST,
- Type checker – implements static type checking based on inference rules,
- Interpreter – implements the language semantics defined via big-step rules or denotational semantics.

Its support for denotational and operational semantics makes it well suited for the executable modeling of DSLs also in mission-critical domains.

The SLANG tool is not the only one of its kind and there are other similar software (even at a higher level or with greater processing capabilities, e.g. Maude or K). We have so far successfully used this tool to implement an executable specification of the traditional While/IMP language [1], the NBL (Natural/Boolean Language), the robot control language [30], or a simple interpreter for stack calculations – language StackLang [31] (in the context of teaching the formal semantics course, SLANG provided an executable view of language specifications as a complement to verification in the Rocq tool). The above applications were designed and developed primarily for pedagogical reasons as support for the courses Formal Semantics of Programming Languages, or Formal Languages and Compilers. In addition, SLANG was used to develop and rapidly prototype a substantial subset of the SQL language [32] (which is also a DSL), as well as to implement a machine controller language EFSM which is based on the principle of extended finite state machines [33]. Experience to date confirms its simplicity and usefulness in developing specific solutions, where it is sufficient to implement an interpreter in the style of executable semantics for a correctly defined syntax. SLANG also supports the definition of denotational functions and transition relations in big-step semantics (natural semantics).

SLANG is a formal framework for the executable specification of programming languages, based on denotational semantics. In SLANG, the language is defined using a context-free grammar and a set of semantic equations that assign mathematical meaning to each syntactic construct. In the case of big-step semantics, the meaning of syntactic objects is defined using transition relations. These semantic definitions are written in a declarative meta-language with a functional structure and interpreted as compositional mappings between state and meaning domains.

Based on this specification, SLANG automatically generates, in conjunction with the ANTLR tool, an executable interpreter in Java whose operational behavior exactly corresponds to the formally defined meaning. This approach allows for rapid prototyping of formally defined domain-specific languages, immediate verification of behavior through simulation, and provides a semantic foundation for future tools such as static analysis, property checking, or certified code generation.

In the context of our domain-specific mission planning language, SLANG will be used to define executable semantics of basic commands over an abstract state model. The resulting interpreter will represent an accurate implementation of the language's meaning that can be directly tested, analyzed, and systematically extended without compromising semantic correctness.

6.2. Prototype Structure and Planned Integration

In our current prototype, we define:

- *syntax*: a minimal grammar representing core commands such as `orient`, `deploy`, `transmit`, and composition (`sequence`, `repeat`),
- *domains*: a `State` record domain with fields for orientation, antenna status, and data log,
- *semantics*: a semantic function mapping a command to a state transformer.

The fragment below shows a partial abstract-syntax domain definition; the `State` record and semantic equations are not included in this excerpt:

```
domains{
    Number = Unsigned[NUM];
    Command = Orient[t] + Deploy + Capture[s] + Transmit[d]
        + Wait[ta] + Seq[Command,Command] + Repeat[Number,Command];
    Program = Prog[Command];
}
```

The SLANG prototype, currently in development, is designed as a powerful implementation of a formally specified language, with a view to practical use in various phases of mission design and verification. It serves as a reference interpreter, making an accurate semantic model available to both end users of the language and its developers. It enables simulation and validation of mission scenarios in the early design stages, thus supporting the detection of errors and inaccuracies before deployment. Given the formal basis, it also provides the possibility of early static verification of safety properties, which is crucial especially in critical domains such as autonomous systems control. In addition, it creates the prerequisites for future development and extension of the language.

7. Case Study: Example Mission Plan

We illustrate the semantics and design intent of our domain-specific language using a simplified example of a mission plan involving typical satellite operations. The following DSL program represents a structured sequence of operations:

```
orient(Target1);
deploy;
transmit(Img1);
repeat(3, capture(SensorA))
```

This mission script performs the following actions in order:

- orient the satellite toward `Target1`,
- deploy the communication antenna,
- transmit an image labeled `Img1`,
- capture data from sensor `SensorA` three times.

We assume that $\texttt{Idle} \in \mathit{Target}$ is a distinguished initial orientation.
For this situation, we assume the initial program state:

$$\sigma_0 = \{\mathit{orient} = \texttt{Idle},\ \mathit{antenna} = \mathbf{0},\ \mathit{data_ready} = \mathbf{true},\ \mathit{log} = []\}$$

We interpret the mission script step by step using the previously defined denotational semantics:

$\sigma_1 = [\![\texttt{orient(Target1)}]\!](\sigma_0) \Rightarrow \sigma_1.\mathit{orient} = \texttt{Target1}$
$\sigma_2 = [\![\texttt{deploy}]\!](\sigma_1) \Rightarrow \sigma_2.\mathit{antenna} = \mathbf{1}$
$\sigma_3 = [\![\texttt{transmit(Img1)}]\!](\sigma_2) \Rightarrow \sigma_3.\mathit{log} = [\texttt{Img1}]$
$\sigma_4 = [\![\texttt{repeat(3, capture(SensorA))}]\!](\sigma_3) \Rightarrow$ (sensor data captured three times)

The final state σ_4 reflects a successful execution of the mission sequence with the expected updates to orientation, antenna deployment, data log, and sensor data.

7.1. Preventing Unsafe Commands by Construction

We now consider an incorrect version of the program:

```
transmit(DataX);
deploy
```

Evaluating this program yields:

- $[\![\texttt{transmit(DataX)}]\!](\sigma_0) = \bot$, since $s_0.antenna = \mathbf{0}$
- The program fails before the `deploy` command is executed.

This program behavior illustrates how denotational semantics enforces safety by construction: an unsafe command, such as `transmit` deployment yields $\bot$. In our formalization, $[\![\texttt{transmit}(d)]\!]$ is total as a function into $State_\bot$, while $\bot$ represents the undefined outcome obtained when the precondition `antenna = 1` is not satisfied.

As a result, any program that violates this constraint cannot evaluate to a defined state. This makes such errors observable at runtime and amenable to static analysis. More importantly, the safety property is not enforced externally (e.g., by testing or runtime checks), but is internally encoded in the semantics of the language.

This is one of the key advantages of denotational semantics: already at the language design stage, or when formally verifying its properties using mathematical methods, it allows for the detection of hidden problems that would otherwise only occur during program execution. Where the program could not continue correctly, the semantic function provides an undefined value $\bot$, thereby explicitly marking the boundary between defined and error behavior.

8. Conclusion and Future Work

In this paper, we presented a minimal domain-specific language for spacecraft task sequencing. The language was designed using property-driven principles and defined through denotational semantics. The language enforces safety and correctness properties by construction, and its semantics can be directly executed or used for verification. The proposed language and semantics are intentionally minimal. This design philosophy brings several benefits, each construct has a well-defined, compositional meaning, and property P2 is enforced semantically by returning $\bot$ when its precondition is violated. The SLANG prototype allows immediate testing of semantics and early feedback from domain experts. Moreover, the absence of loops with dynamic bounds, recursion, or mutable environments facilitates lightweight analysis.

Nonetheless, the model also has limitations: it does not yet model time constraints or concurrent execution, and it lacks formal type checking and type inference. Furthermore, some realistic mission logic (e.g., conditional branching) is not yet expressible. A case study demonstrated the practical application of the language and its semantics to typical mission operations. Furthermore, we outlined an executable semantics prototype using the SLANG framework, enabling future integration into simulation, verification, or code generation pipelines. This work provides a foundation for future extensions that preserve semantic transparency while increasing expressiveness.

In our future research, we will focus on extending the language with types, conditionals, and time constraints, and on enhancing the SLANG prototype with full support for language constructs and analysis capabilities. We also plan to evaluate the approach in both educational and industrial settings, and to explore its integration with runtime verification and synthesis frameworks as part of a broader model-based development methodology. The simplicity and formal precision of the proposed language also make it suitable as a pedagogical tool for teaching formal methods and executable semantics in safety-critical domains.

Acknowledgments

This work was supported by KEGA project 030TUKE-4/2023 “Application of new principles in the education of IT specialists in the field of formal languages and compilers”, granted by the Cultural and Education Grant Agency of the Slovak Ministry of Education. The second author gratefully acknowledges the financial support of the Ministry of Science, Technological Development and Innovation of the Republic of Serbia (Grants No. 451-03-33/2026-03/ 200125 & 451-03-34/2026-03/ 200125).

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] W. Schreiner, W. Steingartner, The SLANG Semantics-Based Language-Generator: Tutorial and Reference Manual (Version 1.0), Technical Report Technical Report 23-13, Research Institute for Symbolic Computation, Johannes Kepler University Linz, 2023. doi:10.35011/risc.23-13.

[2] V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, K. Tso, Plan execution interchange language (PLEXIL) for executable plans and command sequences, in: Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS), Munich, Germany, 2005, pp. 1–8. ESA SP-603.

[3] G. Dowek, C. Muñoz, C. Rocha, Rewriting logic semantics of a plan execution language, Electronic Proceedings in Theoretical Computer Science 18 (2010) 77–91. URL: http://dx.doi.org/10.4204/EPTCS.18.6. doi:10.4204/eptcs.18.6.

[4] I. Perez, A. E. Goodloe, F. Dedden, Runtime verification in real-time with the copilot language: A tutorial, in: A. Platzer, K. Y. Rozier, M. Pradella, M. Rossi (Eds.), Formal Methods, Springer Nature Switzerland, Cham, 2025, pp. 469–491.

[5] I. Perez, F. Dedden, A. Goodloe, Copilot 3, NASA Technical Memorandum NASA/TM–2020–220587, National Aeronautics and Space Administration, Langley Research Center, 2020. URL: https://ntrs.nasa.gov/citations/20200003164.

[6] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Schumann, Automated formalization of structured natural language requirements, Information and Software Technology 137 (2021) 106590. URL: https://www.sciencedirect.com/science/article/pii/S0950584921000707. doi:https://doi.org/10.1016/j.infsof.2021.106590.

[7] G. Vázquez, A. Mavridou, M. Farrell, T. Pressburger, R. Calinescu, Robotics: A new mission for fret requirements, in: N. Benz, D. Gopinath, N. Shi (Eds.), NASA Formal Methods, Springer Nature Switzerland, Cham, 2024, pp. 359–376.

[8] NASA Goddard Space Flight Center, core Flight System (cFS), 2026. URL: https://etd.gsfc.nasa.gov/capabilities/capabilities-listing/cfs/, accessed 31 July 2026.

[9] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, R. Strniša, Ott: effective tool support for the working semanticist, in: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 1–12. URL: https://doi.org/10.1145/1291151.1291155. doi:10.1145/1291151.1291155.

[10] G. Roșu, T. F. Șerbănuță, An overview of the k semantic framework, The Journal of Logic and Algebraic Programming 79 (2010) 397–434. URL: https://www.sciencedirect.com/science/article/pii/S1567832610000160. doi:https://doi.org/10.1016/j.jlap.2010.03.012, membrane computing and programming.

[11] M. Felleisen, R. B. Findler, M. Flatt, Semantics engineering with PLT Redex, Mit Press, 2009.

[12] J.-L. Colaço, An overview of scade, a synchronous language for safety-critical software (keynote), in: Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 1. doi:10.1145/3427763.3432350.

[13] T. Kosar, N. Oliveira, M. Mernik, M. Pereira, M. Črepinšek, C. Da, P. Rangel Henriques, Comparing general-purpose and domain-specific languages: An empirical study, Computer Science and Information Systems 7 (2010) 247–264. doi:10.2298/CSIS1002247K.

[14] J.-P. Bodeveix, M. Filali, J. Lawall, G. Muller, Formal methods meet domain specific languages, in: Integrated Formal Methods, volume 3771 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, 2005, pp. 187–206. doi:10.1007/11589976_12.

[15] E. Fülöp, N. Pataki, A dsl for resource checking using finite state automaton-driven symbolic execution, Open Computer Science 11 (2021) 107–115. URL: https://doi.org/10.1515/comp-2020-0120. doi:doi:10.1515/comp-2020-0120.
[16] I. Ruiz-Rube, T. Person, J. M. Dodero, J. M. Mota, J. M. Sánchez-Jara, Applying static code analysis for domain-specific languages, Software and Systems Modeling 19 (2020) 95–110.
[17] M. Sun, G. Bakirtzis, H. Jafarzadeh, et al., Correct-by-construction requirement decomposition, Software and Systems Modeling (2025). doi:10.1007/s10270-025-01291-4.
[18] H. Nielson, F. Nielson, Semantics with Applications: An Appetizer, Undergraduate Topics in Computer Science, Springer, 2007.
[19] D. Patterson, N. Mushtak, A. Wagner, A. Ahmed, Semantic soundness for language interoperability, arXiv preprint arXiv:2202.13158 (2022). doi:10.48550/arXiv.2202.13158. arXiv:2202.13158.
[20] W. Steingartner, V. Novitzká, M. Bačíková, S. Korečko, New approach to categorical semantics for procedural languages, Computing and Informatics 36 (2018) 1385–1414. URL: https://www.cai.sk/ojs/index.php/cai/article/view/2017_6_1385.
[21] G. Winskel, The Formal Semantics of Programming Languages, MIT Press, 1993.
[22] E. W. Dijkstra, Go to statement considered harmful, Communications of the ACM 11 (1968) 147–148.
[23] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (2005).
[24] R. Neugebauer, U. Frieß, J. Paetzold, M. Wabner, M. Richter, Approach for the development of energy-efficient machine tools, Journal of Machine Engineering 9 (2009) 51–62.
[25] C. Weber, T. Deubel, New theory-based concepts for PDM and PLM, in: A. Folkeson, K. Gralén, M. Norell, U. Sellgren (Eds.), Proceedings of the 14th International Conference on Engineering Design (ICED 03), Stockholm, Sweden, 2003, pp. 429–430. URL: https://api.semanticscholar.org/CorpusID:55392041.
[26] M. Brambilla, C. Pinciroli, M. Birattari, M. Dorigo, Property-driven design for swarm robotics, in: V. Conitzer, M. Winikoff, L. Padgham, W. van der Hoek (Eds.), Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012), International Foundation for Autonomous Agents and Multiagent Systems, Valencia, Spain, 2012, pp. 139–146.
[27] D. A. Schmidt, Denotational Semantics: A Methodology for Language Development, William C. Brown Publishers, USA, 1986.
[28] W. Schreiner, Thinking Programs, Texts & Monographs in Symbolic Computation, 1st ed., Springer, Cham, Switzerland, 2021. URL: https://doi.org/10.1007/978-3-030-80507-4.
[29] J. E. Stoy, Denotational semantics : the Scott-Strachey approach to programming language theory, Series in Computer Science, No. 1, MIT Press, Cambridge, MA, 1977.
[30] W. Steingartner, V. Novitzká, W. Schreiner, Proof of equivalence of semantic methods for a selected domain-specific language, Journal of Applied Mathematics and Computational Mechanics 23 (2024) 79–92. doi:10.17512/jamcm.2024.2.07.
[31] W. Steingartner, W. Schreiner, Executable Semantics for Teaching Concatenative Stack-Based DSLs: The Case of StackLang, in: P. K. Chrysanthis, K. Nørvåg, K. Stefanidis, Z. Zhang, E. Quintarelli, E. Zumpano (Eds.), New Trends in Database and Information Systems, ADBIS 2025 Short Papers, Doctoral Consortium and Tutorials, Tampere, Finland, September 23-26, 2025, Proceedings, volume 2676 of *Communications in Computer and Information Science (CCIS)*, Springer, Cham, Switzerland, 2025, pp. 248–263. URL: https://doi.org/10.1007/978-3-032-05727-3_23.
[32] W. Schreiner, W. Steingartner, Semantics-Based Rapid Prototyping of a Subset of SQL, Technical Report 25-02, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2025. doi:10.35011/risc.25-02.
[33] W. Schreiner, W. Steingartner, Semantics-based rapid prototyping of a machine controller language, in: V. Novitzká, A. Szakál (Eds.), Proceedings of the 17th IEEE International Scientific Conference on Informatics, IEEE, Poprad, Slovakia, 2024, pp. 348–353. doi:10.1109/Informatics62280.2024.10900792.