Adaptive Structural Operational Semantics

Gwendal Jouneaux, Damian Frölich, Olivier Barais, Benoit Combemale,
Gurvan Le Guernic, Gunter Mussbacher, L. Thomas van Binsbergen

Adaptive Structural Operational Semantics

Gwendal Jouneaux
gwendal.jouneaux@irisa.fr
Univ. Rennes, Inria, IRISA
Rennes, France

Damian Frölich
dfrolich@acm.org
University of Amsterdam
Amsterdam, The Netherlands

Olivier Barais
Benoit Combemale
firstname.lastname@irisa.fr
Univ. Rennes, Inria, IRISA
Rennes, France

Gurvan Le Guernic
gurvan.le_guernic@inria.fr
DGA Maîtrise de l'Information, Univ. Rennes, Inria, IRISA
Rennes, France

Gunter Mussbacher
gunter.mussbacher@mcgill.ca
McGill University, Inria
Montreal, Canada

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
University of Amsterdam
Amsterdam, The Netherlands

Abstract

Software systems evolve more and more in complex and changing environments, often requiring runtime adaptation to best deliver their services. When self-adaptation is the main concern of the system, a manual implementation of the underlying feedback loop and trade-off analysis may be desirable. However, the required expertise and substantial development effort make such implementations prohibitively difficult when it is only a secondary concern for the given domain. In this paper, we present ASOS, a metalanguage abstracting the runtime adaptation concern of a given domain in the behavioral semantics of a domain-specific language (DSL), freeing the language user from implementing it from scratch for each system in the domain. We demonstrate our approach on RobLANG, a procedural DSL for robotics, where we abstract a recurrent energy-saving behavior depending on the context. We provide formal semantics for ASOS and pave the way for checking properties such as determinism, completeness, and termination of the resulting self-adaptable language. We provide first results on the performance of our approach compared to a manual implementation of this self-adaptable behavior. We demonstrate, for RobLANG, that our approach provides suitable abstractions for specifying sound adaptive operational semantics while being more efficient.

CCS Concepts: • **Software and its engineering → Specification languages**; **Semantics**; *Source code generation.*

Keywords: DSL, Operational Semantics, Self-Adaptation

ACM Reference Format:
Gwendal Jouneaux, Damian Frölich, Olivier Barais, Benoit Combemale, Gurvan Le Guernic, Gunter Mussbacher, and L. Thomas van Binsbergen. 2023. Adaptive Structural Operational Semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623517

SLE '23, October 23–24, 2023, Cascais, Portugal

1 Introduction

In a constantly evolving world, the need to design programs malleable to varying execution conditions (*i.e., self-adaptive* programs) is an important issue that has been worked on for many years by the adaptive systems community [16]. While there exist many frameworks (*e.g.,* [15, 37]) and architectural approaches (*e.g.,* [5, 21]), one of the difficulties in the field of domain-specific languages is for the average domain experts to be in a position to have to handle this concern, that is naturally outside their expertise. Indeed, this goes against the intuition that a domain-specific language (DSL) allows a domain expert to focus on their domain and generally leads to incorporating dedicated concepts for implementing self-adaptation in domain-specific languages.

To solve this problem, we propose a metalanguage, named **ASOS** (Adaptive Structural Operational Semantics), aimed at freeing the domain expert from the task of implementing self-adaptation. ASOS allows to define context-adaptable semantics for domain-specific languages. Built as an extension of MSOS [25], our approach aims to:

- Provide modular abstractions for defining and applying adaptations in the DSL's operational semantics;
- Leverage a static introduction mechanism for composing the definition of an adaptation;
- Ensure that the adaptations introduced within the semantics of a DSL do not break the fundamental properties of programs written in the DSL, such as determinism, completeness, and termination.

The contributions of this paper encompass (1) the definition of the ASOS metalanguage, (2) ASOS formal semantics for reasoning over self-adaptable operational semantics, and (3) a first implementation through a translational semantics from ASOS to SEALS [17], a framework for self-adaptable lan-

guages. This paper illustrates the concepts of the approach in the definition of a DSL for robot behavior (named RobLANG).

We then present the applicability of the approach on RobLANG. Next, we define the formal semantics of the ASOS metalanguage and discuss the alignment of the implementation with the formal semantics. Furthermore, we hint at the capability of verifying properties such as determinism, termination, and completeness of a program bound to an adaptive semantics. Lastly, we evaluate the performance overhead associated with the approach. We conclude that ASOS provides the foundations for specifying sound adaptive operational semantics while introducing little performance overhead compared to a manual implementation.

The remainder of the paper is as follows. Section 2 provides background and motivation, and introduces RobLANG. The following section gives an overview of our approach, including the definition of the abstract syntax and semantics of a self-adaptable language and the configuration of the self-adaptation loop. Section 4 details the syntax and semantics of ASOS. The last three sections present the evaluation, related work, and our conclusion and future work.

2 Motivation and Illustrative Example

While a manual implementation of the self-adaptation concern may be desirable when it is the main concern of the system, the required expertise and substantial development effort often does not justify a manual approach when it is only a secondary concern for the given domain. A more automated approach is needed. Hence, we first provide background information on self-adaptive system design and describe the limitations of the current approaches in the context of domain-specific languages (DSLs). Finally, we present the RobLANG DSL as an illustrative example that would benefit from the abstraction of self-adaptation at the language level.

2.1 Design of Self-Adaptive Systems

Self-adaptive systems are designed to adjust their behavior based on changes in the system or its environment. This changement of behavior is called an adaptation. They use sensors to gather data, analyze the current situation, decide on a new behavior if necessary, and implement the changes through effectors. This process is known as the feedback loop scheme, which consists of four main functions: Monitoring, Analysis, Planning, and Execution (MAPE-K) [19].

Previous work helps in the design of self-adaptive systems providing architectural solutions (*e.g.,* three layer architecture [21], MORPH [5], PLASMA [35]) and frameworks (*e.g.,* Executable Runtime Megamodels [37], DCL [27], ActivFORMS [15], Ponder2 [36]). While architectural solutions give guidelines, they generally do not support designers during the implementation. Meanwhile, frameworks provide support to the designer. However, those frameworks are restricted to the languages they were designed for, likely requiring re-implementation for other languages.

Another way to design a self-adaptive system is to define self-adaptation at the meta-level. SEALS [17], *e.g.*, supports language engineers in the implementation of self-adaptable virtual machines, providing abstractions and avoiding to re-implement a known framework for self-adaptation from scratch. SEALS provides facilities to support the definition of the language abstract syntax and operational semantics, the feedback loop and associated trade-off reasoning, and the adaptation semantics and the predictive model of their impact on the trade-off (Impact Model). However, SEALS does not readily enable formal verification of properties such as determinism, completeness, and termination.

2.2 Limitations of Current DSL Approaches

In the context of DSLs, the re-implementation of frameworks for self-adaptation can be tedious or impossible, depending on the expressiveness of the language. E.g, the difficult task of implementing a constraint solving algorithm for trade-off analysis or monitoring the execution environment (*e.g.,* CPU, RAM) may not be supported. Moreover, this implementation requires the language user to be an expert in the design of self-adaptive systems, which is often not the case. However, designing self-adaptation at the meta-level offers appropriate tooling to language engineers for the implementation.

2.3 Illustrative Example: Energy-Aware RobLANG

RobLANG[1] is a representative of procedural DSLs used, in this case, for specifying the actions of a robot. The domain concepts manipulated in RobLANG include speed changes, movement (forward and backward), orientation (turn left-/right), and use of the sensors (*e.g.,* battery, distance). To orchestrate these actions, RobLANG also has functions, control structure (if and while), arithmetic and boolean expressions.

Often, the domain of robotics requires developers to implement the behavior of the robot with energy efficiency in mind to avoid battery depletion. One way to reduce the energy consumption of a robot is to reduce the speed of the motors. This is due to the exponential increase in motor energy costs as a function of speed [2].

However, speed is often also an important factor in the robot mission. Therefore, it is necessary to dynamically apply this speed reduction, depending on the trade-off between energy consumption and speed, taking into account various potentially dynamic pieces of information (*e.g.,* availability of the power supply, current level of the battery, time estimation to complete the current task, importance of the task).

In this context, RobLANG would benefit from the abstraction of this recurrent adaptive behavior in a metalanguage to free the language user from the implementation of the feedback loop and trade-off analysis.

[1]Implementation : https://www.gwendal-jouneaux.fr/SLE2023/RobLANG

3 Approach Overview

This section presents a general overview of our approach and describes ASOS (Adaptive Structural Operational Semantics), a metalanguage to specify and reason on self-adaptable operational semantics. Figure 1 depicts the design of a language with adaptive operational semantics, *i.e.,* a Self-Adaptable Language (SAL), and the generation of an interpreter in SEALS [17], an implementation framework for building self-adaptable virtual machines (see Section 2.1). SEALS is used as the target of the generation because of the suitable abstraction. The goal is to allow, through ASOS, to reason about such semantics, rather than on SEALS Java code. The execution of the ASOS semantics rules of the SAL are presented in Figure 2, starting from the *Evaluate* step, with first the feedback loop and then the rule executions.

The definition of a self-adaptable language, as in other languages, includes the definition of its abstract syntax and semantics. The abstract syntax specifies the domain concepts of the language and their relations, whereas the semantics define the meaning of those concepts. In this paper, with SEALS as the target implementation framework, we focus on operational semantics as a way to define language semantics.

3.1 Abstract Syntax Definition

In the modeling community, the abstract syntax is often expressed with a metamodel. Since SEALS relies on a Java-based definition of abstract syntax, we choose to use Ecore [34] as the metalanguage to express the metamodel. In addition, a generator from Ecore to Java classes using EMF [34] exists and can be modified to generate the SEALS-based interpreter. This allows a language engineer to define the abstract syntax as they would normally do for any Ecore-based DSL definition (*e.g.,* EcoreTools [34]). Figure 1 represents this by the language engineer defining the abstract syntax conforming to the Ecore metamodel, and the SEALS-based interpreter being generated for the language implementation.

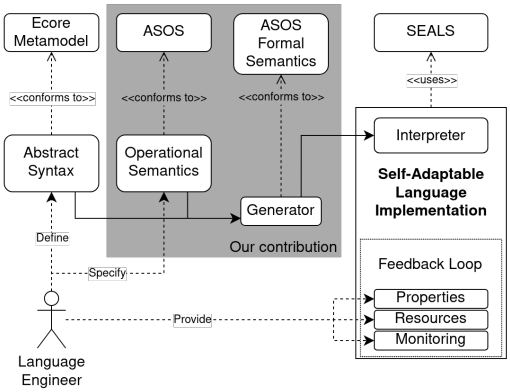

Figure 1. Approach overview

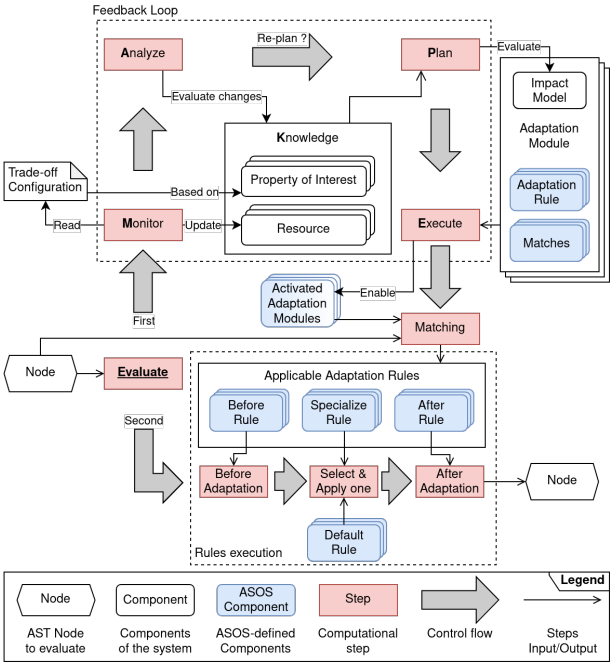

Figure 2. Overview of ASOS semantics execution

3.2 Operational Semantics Definition

Operational semantics define the meaning of the language concepts by expressing the computational steps (evolution of the state of the execution at runtime). The operational semantics of a language can be expressed using metalanguages such as Structural Operational Semantics (SOS) [29], and is typically implemented through an interpreter or a compiler.

In particular, SEALS defines operational semantics in the form of an interpreter composed of three components: a default semantics for the language, a feedback loop performing analyses, and modular adaptations modifying the default semantics. SEALS, through this decomposition, ensures a default behavior and allows the delegation of the development and/or configuration of adaptations to the language users, or even the end-users. ASOS keeps this decomposition for the same reasons. These three components are represented, in Figure 2, through the stack of *Default Rule* at the bottom, the *Feedback Loop* dashed box, and the stack of *Adaptation Module* on the right, respectively.

To support a modular approach and the ability to reason on operational semantics, we choose to base our definition of default and adaptation semantics on Modular SOS (MSOS) [25] and its extension I-MSOS [26]. Among the potential frameworks to formally specify semantics, such as $\mathbb{K}$ [32] and its matching logic [31], we choose MSOS for its focus on modular definition of operational semantics and the implicit propagation of auxiliary components of its extension I-MSOS, reducing redundancy in the rule writing and fitting the propagation of models through programs execution.

3.2.1 Default Semantics (Rules). The definition of the default semantics is a set of transition rules for all the concepts of the language. I-MSOS transition rules can be decomposed into several components, presented in Figure 3. First, the conclusion of the rule represents the effect of the transition on the state. This transition from a pattern used to match a particular term structure, *i.e.,* the *input pattern*, results in a new term to be evaluated or a computed value to return, *i.e.,* the *result* of the rule. When a transition is performed, we say that the term matching the *input pattern progressed* to the *result*. In addition, this transition can affect auxiliary entities such as the memory. In I-MSOS, these auxiliary entities are implicitly propagated and only expressed when a rule makes use of them. This conclusion is conditioned by two other components of the rule: side-conditions and premises.

Side-conditions allow to limit the application of the rule to a subset of the terms matching the structure defined in the conclusion of the rule. For example, a rule performing a division would only apply if the divisor is not zero. These conditions are also used to describe computations over computed values. For instance, a rule performing an addition would define a condition $n = n1 + n2$ such as $n1$ and $n2$ are the computed values of the left and right expressions. In this case, rather than evaluating the predicate, those conditions require an instance of n to make the predicate true, thus computing the value for n.

On the other hand, premises define assertions on the ability of subterms (terms contained in the main term) to *progress*, *i.e.,* perform transitions. It differs from the side-condition computations, as it represents computations of terms using rules.For example, a rule in charge of evaluating the condition of an if statement progresses to a term representing the same statement but replacing the original condition by the *result* of its progression. However, this makes sense only if the condition can progress. Hence, this rule is conditioned by a premise on the ability of the condition to progress.

ASOS reuses these concepts in the definition of its rules, which are detailed in Section 4.1. The key difference with I-MSOS is the clear separation between side-conditions and computations present in ASOS. In ASOS, computations are explicit and grouped, with assignment of values to/from the propagated auxiliary entities, in a dedicated section of the rule. Hence, making the side-conditions, as for the pattern matching, has no side effects. In the proposed implementation, these auxiliary entities are defined using a meta-model representing the structure of the semantic domain. The merge of the abstract syntax metamodel and the metamodel for the semantic domain forms the execution metamodel. Instances of this execution metamodel (execution models) represent the runtime state of the program, which is implicitly propagated in the runtime in the same way as the I-MSOS auxiliary entities.

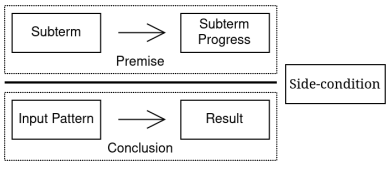

Figure 3. Structure of I-MSOS rule

3.2.2 Feedback Loop and Trade-Off Reasoning. At the core of a self-adaptable language, there is a feedback loop selecting the adaptations to perform depending on monitored resources and the desired trade-off between the addressed properties of interest. The resources represent the environment upon which the decision to adapt is taken, while properties of interest denote the specific properties that we seek to maximize through adaptations. To implement the feedback loop, SEALS requires the implementation of the Monitor, Analyze, Plan, and Execute phases (red boxes of the feedback loop in Figure 2) and Knowledge base, providing the resources to monitor, the properties of interest, and the function reading the desired trade-off. ASOS provides an implementation for the Analyze, Plan, and Execute phases of the feedback loop based on the modeling approach provided by the framework. However, the feedback loop still requires configuration. The resources and properties of interest needs to be configured in SEALS, and monitoring hooks must be implemented to update resources values. They are represented in the *Knowledge* box of the *Feedback Loop*, and by the "read" and "update" arrows in Figure 2 (see also the Feedback Loop in Figure 1). In this first version of ASOS, we do not provide abstractions for the configuration and monitoring hooks to retain flexibility to implement various strategies.

3.2.3 Defining Adaptations Modules. Finally, to express the adaptations of the operational semantics, ASOS requires adaptation modules.An adaptation module can be defined by the language engineer, or delegated to other stakeholders (*e.g.*, language users, end-users). However, while the inclusion of external adaptation modules is facilitated by SEALS and generated code from ASOS, the method used to manage external adaptations (*e.g.,* link in command line, dedicated folder) is left to the language engineer to implement.

An adaptation module is defined by three components: adaptation rules, matching clauses, and a model of expected impacts of the adaptation on the properties of interests. Adaptation rules are defined similar to default rules but with an additional description on how to introduce them in the operational semantics dynamically and under which conditions. We propose three ways to introduce adaptation rules: (i) *specialization*, where the new rule replaces an existing rule, (ii) *before*, where the new rule is executed before another rule, and (iii) *after*, where the new rule is executed after another rule. It means that before executing the default rule, as depicted in the *Rules execution* box of Figure 2, the applicable before adaptations are called, then either one of the

specialization rules or the default rule is applied, and finally the applicable after adaptations are called.

Of course, an adaptation may not always apply. To specify when the conditions are met and the adaptation can be introduced, we propose a matching system based on a structural matching on the Abstract Syntax Tree and conditions on the runtime values in the execution model. As shown in Figure 2, every adaptation module define *Matches*, that are used during a *Matching* step verifying if the adaptation is applicable on the current AST node.

In addition to this matching system, an adaptation also needs to be activated by a feedback loop evaluating its relevance to the current trade-off and environment. To do so, the adaptation module must declare a predictive model of its impact on the properties of interest, the *Impact Model*. This model is used in the *Plan* phase of the *Feedback Loop* (see Figure 2) to select the set of adaptations based on the trade-off given the current context. The adaptations are then enabled in the *Execute* phase.

The SEALS implementation for adaptation modules is derived from the ASOS specification for this module. However, the impact model for this module still needs to be defined. It is left as future work to provide the appropriate abstractions for the impact models in ASOS because there are multiple alternatives to implement the *Analyze* and *Plan* phases and define the impact models (*e.g.,* Goal Modeling, Machine Learning). When using the base implementation of the feedback loop provided by ASOS, the language engineer will have to define a goal modeling-based impact model using the constructs provided by the SEALS framework.

4 The ASOS Metalanguage

ASOS[2] is a declarative language to specify operational semantics based on MSOS [25]. ASOS extends MSOS by providing abstractions to define runtime dynamic adaptation of the operational semantics of the language. The definition of the operational semantics is done through transition rules in the same fashion as MSOS. The additional adaptation concern is managed using adaptation rules, *i.e.,* transition rules with additional information on how and when to introduce them in the set of applicable transition rules. This is defined using the ASOS matching system because it allows to express structural patterns that are not possible to express using typical MSOS rule format [9].

4.1 ASOS Syntax

This section describes the abstract syntax of the ASOS metalanguage. Figure 4 shows the main concepts of the ASOS metamodel. `Adaptive Operational Semantics` is the top-level concept representing the adaptive operational semantics of the implemented DSL, and is composed of a set of rules and a set of adaptation modules representing the default semantics and the adaptation semantics of the language. We use RobLANG[3] as an illustrative example with the concrete syntax provided by the implementation (Section 4.2).

4.1.1 Structure of Transition Rules.

The `Rule` concept is at the core of ASOS, describing the computation to perform for a given concept. These computations are mainly described using the `Transitions` components of the rule. Two types of transitions exist, the conclusion of the rule and the premises. Both represent the same concepts as the ones from MSOS described in Section 3.2.

```
1  rule IfCond,
2       RobLANG.If(cond, then, else)
3       ->
4       RobLANG.If(newcond, then, else)
5  resolve
6       cond -> newcond
7
8  rule IfTrue,
9       RobLANG.If(sd.ValueBool(b), then, else) -> then
10 where
11      b == true
12
13 rule IfFalse,
14     RobLANG.If(sd.ValueBool(b), then, else) -> else
15 where
16     b == false
```

Listing 1. Transition rules to compute an if condition

Listing 1 presents the definition of three `Rules` for the *If* concept. `Transitions` are represented using an arrow (→), with the conclusion defined as the first transition in the rule (*e.g.,* lines 2-4 for rule *IfCond*) and premises defined in the *resolve* section (*e.g.,* line 6). The LHS and RHS of transitions are `Terms` except that the LHS of the conclusion transition (*e.g.,* line 2) must be a `Configuration` defined in the abstract syntax of the DSL. Such a `Configuration` is prefixed with *RobLANG* and defines the concept on which to execute the rule. A concept prefixed with *sd* is also a `Configuration` but defined in the semantic domain structure and represents the computed values. The constructor notation[4] is used to denote a `Configuration` (*e.g., RobLANG.If(...)*), whereas `Symbols` are represented as identifiers (*e.g., cond*, *then*, *else*).

The subterms in the parenthesis of the constructor notation correspond to the elements contained in this concept as defined in the DSL's abstract syntax. This could represent a computed value constructor (*e.g., sd.ValueBool(...)* in line 9) or it could bind a name to the subterm of a configuration for further use in the premise (*e.g., cond*). The subterms allow us to represent part of the state of the evaluation of the concept, and update it. When defining a computed value constructor (prefixed with *sd*) for a subterm (see line 9), this implies that this subterm has been computed. Moreover, premises assert that a subterm, via a transition, can change state. In line 6 of *IfCond*, the state of evaluation of the if condition changed, and the *newcond* `Symbol` is bound to this new state.

[2]Implementation: https://www.gwendal-jouneaux.fr/SLE2023/ASOS

[3]Specification: https://www.gwendal-jouneaux.fr/SLE2023/ASOSRobLANG

[4]A *constructor notation* is the pattern : `prefix.Concept(subterms...)`

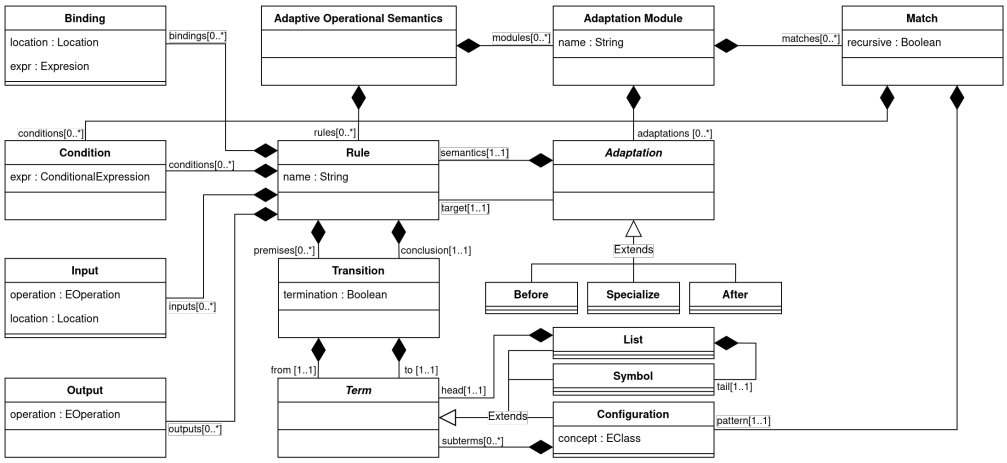

Figure 4. ASOS Metamodel (ConditionalExpression, Expression, and Location omitted to focus on main concepts)

The constructor notation akin to algebraic data allows us to express the impact of this change in the evaluation state of the *If* concept by changing *cond* to *newcond* in the output term (RHS of transition) using the `Symbol` as a reference. Just like in I-MSOS, memory and other auxiliary entities are propagated implicitly and do not need to be represented.

In addition to the premises assertion, rules can be conditional. The *IfTrue* rule includes a `Condition` (line 13) in the *where* section of the rule. A condition could be used to define first-order logic predicates over values. For instance, the choice of the branch of the if statement to execute depends on the condition truth value.

```
1  rule Break,
2      RobLANG.BreakLoop()
3      ->
4      termination sd.BreakSignal()
5
6  rule LoopTrueBreak,
7      RobLANG.Loop(sd.ValueBool(b), body)
8      ->
9      sd.NilValue()
10 where
11     b == true
12 resolve
13     body -> termination sd.BreakSignal()
```

Listing 2. Abrupt termination using loop breaking

To ease handling abrupt termination (*e.g.,* errors, breaks, returns), transitions can emit and receive abrupt termination signals using the keyword `termination`. Conclusion transitions using the `termination` keyword emit a signal containing the usual output (e.g., line 4 in Listing 2). On the other hand, premises with a `termination` nature (e.g., line 13) are the only premises matching a transition emitting such signal. This allows the language engineer to receive and handle this abrupt termination, while remaining oblivious of the upward propagation of these signals when not managed. Furthermore, support of termination allows for default handling to be embedded in the ASOS metalanguage.

Transitions may require computations and/or storing capabilities to describe the arrival state (*e.g.,* arithmetic expressions, assignments). In ASOS, both are managed using the `Binding` construct. Bindings associate the result of an expression to a location. This location can be either a new `Symbol` (e.g., line 6 in Listing 4), allowing reuse of the expression result in the output of the transition, or a "dot" notation allowing to set values in the execution model propagated during the execution. Expressions support the "dot" notation to access the execution model, symbol reference, the usual arithmetic and boolean operators, and constants.

Finally, some concepts involve an external component either for *Input* or *Output*. For example, print statements require access to a communication interface, *i.e.,* the console. This access is managed through external functions conforming to a predefined signature in the semantic domain structure definition. In addition, the *on* keyword can be used to specify the object on which to call the function. An `Input` is denoted through the assignment of a function result (e.g., lines 8-9 in Listing 4), while an `Output` is just a function call (e.g., lines 10-12 in Listing 4).

4.1.2 Adaptation Modules Definition.

Adaptations are defined in modules. An `Adaptation Module` groups a set of transition rules representing the adaptation of a concept. These transition rules are defined like other transition rules as explained earlier. However, an adaptation developer needs to additionally define the conditions to apply the adaptation at the module level, and how the adaptation rules are added to the operational semantics at the rule level.

```
1 recursive match RobLANG.Loop(
2     RobLANG.GreaterEqual(
3         lhs,
4         RobLANG.DoubleConstant(d)),
5     body)
6 where
7     d == 0.0
```

Listing 3. Match clause of an adaptation module

The conditions are defined using a `Match` clause. This match clause is composed of a recursive (or not) nature, a structural match, and conditions, the latter two being similar to the input pattern (LHS of conclusion transition) and *where* section of a rule, respectively. Listing 3 gives an example of an adaptation module's match clause. In this example, the structural match targets a while loop of the form: *while(lhs* ≥ *d){body}*. Moreover, the *where* section condition ensures that the constant *d* is equal to 0. The match describes a configuration which leads to the dynamic introduction of the module adaptation rules in the operational semantics when the current term to evaluate is matched. The new rules introduced can be used for the evaluation of this term and/or all of its descendant in the AST. If a descendant of the matched term happens to be of the same nature (here the *Loop* concept) but is not valid with respect to the match clause, there are two possibilities. If the match clause possesses the recursive nature, nothing changes and the introduced rules remains. However, if the match is not recursive, the introduced adaptation rules are removed from the operational semantics for the descendant and the associated sub-tree of the AST.

To define how to introduce adaptation rules in operational semantics, we propose three types of adaptation rules presented in Listing 4. A `Specialization` adaptation defines a rule that will replace a target rule in the operational semantics. For instance, the adaptation rule *HalfSpeedForward* is a rule that replaces the *ForwardAct* rule to perform the moving forward action at half of the speed. `Before` and `After` adaptation also target an existing rule, but the rule defined is executed respectively before or after the target rule.

```
1  Specialize ForwardAct rule HalfSpeedForward,
2      RobLANG.MoveForward(sd.ValueDouble(d))
3      -> sd.NilValue()
4  bind
5      half = 0.5 * s
6  IO
7      ctx = RobLANG.WithContext.getContext();
8      s = sd.Context.getNominalSpeed() on ctx;
9      sd.Context.setSpeed(half) on ctx;
10     sd.Context.moveRobot(d) on ctx;
11     sd.Context.setSpeed(s) on ctx
12
13 Before TargetRule rule BeforeTargetRule,
14     ...
15 After TargetRule rule AfterTargetRule,
16     ...
```

Listing 4. Three types of adaptation rules

4.1.3 Well-Formedness Rules.

To specify a well-formed ASOS semantics, additional constraints on the abstract syntax need to be followed. First, all transitions in a rule are not defined in the same way. The transition representing the conclusion of the rule requires a `Configuration` defined in the abstract syntax of the DSL as left `Term` (from), which is not the case for premises. Second, `List` terms, representing subterms with cardinality greater than 2, can only be used as subterms to decompose, for instance, the list of statements in a loop. Finally, adaptation rules require well-formedness constraints to ensure their applicability. Thus, a `Before` adaptation requires, as result, a valid term that can be executed by the adapted rule.

4.2 ASOS Translational Semantics

In this section, we detail the current implementation of the ASOS language. This implementation takes the form of a translational semantics to a Java implementation based on Ecore [34] and the SEALS framework [17].

4.2.1 Derive Java Code from ASOS Transition Rules.

To derive a Java implementation of a transition rule, we divide the rule into two parts: effects and guards. Effects represent the effects of the rule on the state, such as the resulting term of the rule, bindings, inputs, and outputs. Guards represent the conditions to apply a rule, such as the input pattern, premises, and conditions. The overview of the rule guards and effect generation is presented in Figure 5. Solid arrows represent the generation of one element of the rule, while dashed arrows represent the generic generation repeated for all instances in a section. Arrows pointing to adaptations represent the call site of this type of adaptation.

To generate the effect of the rule, we first generate the inputs, then the bindings, the computation of the resulting term, and finally outputs. `Input` and `Output` are defined as callable functions in ASOS. In the implementation, we use Ecore *EOperations* to model these functions. We first generate the processing of the function arguments, then create a call to the appropriate *EOperation* for each input and each output. This is presented in Figure 5 by the dashed arrows from the `Input` and `Output` of the *IO* section. For `Input`, the result of each function is stored either in a temporary variable if the `Location` is a `Symbol`, or in the execution model by resolving the associated "dot" notation. Each `Binding` generates, as for `Input`, an assignment of the computed expression to a temporary variable or a call to the appropriate setter of the execution model. The expression of the `Binding` is translated to its Java equivalent, with execution model accesses resolved as calls to the appropriate getters.

Finally, we generate the computation of the output `Term` depending on its form: a `Configuration` of the same concept as the input, a `Symbol`, or a `Configuration` with a different concept. In the case of a `Configuration` of the same concept, the rule is an update of the state of evaluation of the current concept. The update of the current state is generated from the difference between the two configurations. For instance, the `Term` resulting from a premise can be retained in the new state of evaluation of the current concept, as is the case in the *IfCond* rule (line 6 in Listing 1). If the transition resolves to a `Symbol`, we do not know what it represents (term or value). To manage this, we generate a conditional assignment to a *return* variable, that will affect the value if computed, or the term if not. At the end of the rule's semantics, the con-

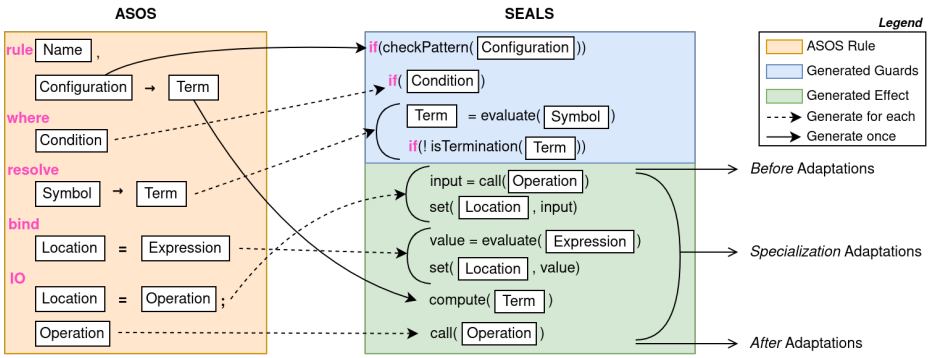

Figure 5. Overview of rule generation (ASOS to Pseudo-code)

tents of this variable are returned or executed, depending on whether it is a value or a term. Finally, if the output is another `Configuration`, we generate the expected structure using the object factory provided by the Ecore model. Subterms represented with a `Configuration` generate a new concept instance and subterms represented with a `Symbol` are resolved and set in the correct concept instance. This structure is assigned to the *return* variable, and is returned or executed depending on whether it is a value or a term.

To generate the guards, we first generate the input pattern matching condition, then check the conditions of the *where* section, and finally the premises. The generation of the input pattern condition is done by checking the type of the subterms of the associated `Configuration`. `Symbols` does not impose constraint, hence generating no condition. For `Configuration` and `List`, we generate the condition on the associated type, and recursively generate conditions for their subterms. This guard is the first generated, ensuring the structure required for valid symbol resolution in the remaining of the rule. The generation of a `Condition` is done by translating its expression to its Java equivalent. If there are multiple conditions, they are represented as nested in the order of definition in the ASOS rule.

Finally, the `Premise` generation produces three statements: a condition checking that the subterm has not been evaluated, a call for the evaluation of the `Symbol`, and a verification of abrupt termination signals. In Figure 5, the first statement is implicitly shown in the *evaluate* method as values cannot be evaluated. This condition is necessary as a premise is an assertion on the transitions of subterms, whereas a computed value can never transition. If it has not been computed, we compute it and store its result for reuse in the effect of the rule. If the right hand side of the premise is a `Configuration`, we also check that the resulting term is matching its pattern A final condition is generated to check the normal or abrupt termination of the premise computation.

If the premise does not expect a termination signal, or if this signal is different than the expected one, the rule is not applied, and the termination signal is stored for potential propagation if no other rules handle this termination. To avoid re-executing a premise when its result is the only difference in rule guards (*e.g., termination vs normal execution*), the result of premise evaluation is shared across those conflicting rules, as a failing guard does not apply a rule, hence does not change the state of execution.

4.2.2 Generate Default Semantics and Feedback Loop.

In SEALS, the default semantics is defined through *Operation* classes, providing the complete semantics for one concept in an "*execute*" function. From those *Operation* classes, SEALS provides a visitor who can evaluate the AST of the language. Moreover, SEALS allows adaptations on those *Operation* semantics by explicitly defining it as an *AdaptableOperation* and providing its interface with adaptations, requiring ASOS to generate one pair for each concept. Finally, SEALS requires the specialization of its *FeedbackLoop*, *AdaptationContext*, and *SelfAdaptableLanguage* concepts, generated by ASOS.

To generate the content of the *execute* function, presented by Figure 6, all the ASOS rules defined for the concept are retained from the set of rules defining the default semantics.

Since we consider the provided semantics definition as deterministic, the generated rules, presented in the yellow box in Figure 6, preserve the ASOS definition order. To store the computed value during the execution of the concept, we automatically generate a data class that contains one field for each subterm and the associated getters and setters. We have two instances of this data class, one for the data kept across rules, and one to store the result of computed premises.

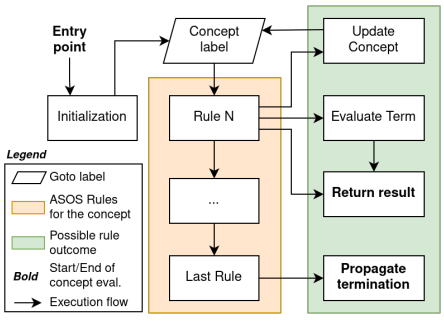

Figure 6. AdaptableOperation *execute* function overview

At the end of a rule execution, there are three cases: the rule returns a value, the rule reduces to a term to evaluate, or the current concept state is updated and needs to be executed further. All of these cases are shown in the green box on the right of Figure 6. In the first case, the computed value is simply returned as a result of the execution of the concept. In the second case, the visitor is called on this term to execute it, then the resulting computed value is returned. Finally, if the current concept requires more rule applications to complete, we jump back to the top of the rules list. In the case that no rule can be applied and because the semantics is considered complete, we know that a termination signal was not handled at this concept level and should be propagated to the calling concept. This is represented by "Propagate termination" at the end of the last rule in Figure 6.

Because adaptations are based on the introduction of new rules in semantics, we generate a language-wide interface defining three `adaptationrule` fields for each rule of the default semantics, corresponding to the `Before`, `After`, and `Specialization` adaptations. These `AdaptationRules` represent a callable adaptation rule, and therefore, will be detailed in the next section. The arrows on the right part of Figure 5 denote the call to these `AdaptationRules` if they exist for this rule in the current context. This is done inside the guards to ensure that we call adaptation only if the rule is effectively called. The effect of the original rule is generated in a way that either the specialization or the effect will be executed but not both. Finally, the `After` adaptations are called on the output of the current rule.

In addition, generic implementations of the three main classes of a SEALS language (*AdaptationContext*, *FeedbackLoop*, and *SelfAdaptableLanguage*) are generated. The *FeedbackLoop* implementation make use of SEALS proposed modeling approach for impact and trade-off analysis. However, the *AdaptationContext* still requires the definition of the resources and properties of interests, and implementation of the trade-off monitoring function.

4.2.3 Generate SEALS Adaptation Modules.

In ASOS, adaptation rules achieving part of the same adaptation logic are grouped in modules. These modules contain, in addition to the set of adaptation rules, a matching expression enabling the adaptation to occur on the term and its subterm. On the other hand, the concept of adaptation module of SEALS provides the *adapt* and *trigger* method. The *trigger* method verifies if the *adapt* method should be called on the current term, while the *adapt* method performs the adaptation.

To map the behavior specified in ASOS, we chose to generate a set of `AdaptationRule` classes representing each adaptation rule, implementing an *adapt* function with the code for the adaptation rule. To evaluate the adaptation rule, the implementation requires an access to the subterms, and a way to evaluate premises of the adaptation rule. The first point is managed by passing the node and the current state of execution of the node. The second is achieved by providing the store of computed premises and the visitor, giving access to computed premises and a way to evaluate the others.

These rules are the ones used in the pattern managing adaptation, at the right on Figure 5. When a match occurs, an instance of these classes will be created and added to the pool of executable rules by adding it to the interface for adaptations. However, SEALS recreates an interface at each step of the evaluation, hence this new set of rules is not propagated. This issue can be resolved by adding some information at the *AdaptationModule* level. Rather than performing the matching for the current node like SEALS, we save the state of the matching at the module level. When evaluating a term that matches the `Match` clause of the module, we save this information in a boolean in the module. Then for every node, if an ancestor node matched the clause, we add the `AdaptationRule` instance to the interface. Depending on the `Match` recursive nature, we potentially invalidate this match when going deeper in the structure. In the end, only the impact model of the adaptation remains for the adaptation designer to specify.

4.3 ASOS Formal Semantics

In this section, we introduce a formalization for the ASOS meta-language and its alignment with the implementation (Section 4.3.1). The formalization allows us to reason about certain properties, in particular, how the adaptation process affects determinism (Section 4.3.2), termination (Section 4.3.3), and completeness (Section 4.3.4). To formalize ASOS we build upon earlier work of generalized transition systems (GTSs) as defined for MSOS by Mosses [25].

Definition 4.1. A GTS is a tuple $\langle \Gamma, \mathbf{A}, \longrightarrow, T \rangle$, where $\mathbf{A}$ is a category with morphisms A, $\longrightarrow \subseteq (\Gamma \times A \times \Gamma)$ is the transition relation, and $T \subseteq \Gamma$ are terminal configurations. $\langle \Gamma, A, \longrightarrow, T \rangle$ is a labeled terminal transition system[5] (LTTS) [30].

The category is referred to as a label category and is an (indexed) product category of component categories $\prod_{i \in I} \mathbf{A}_i$. For a full account of the different types of component categories we refer the reader to [25].

With the category, it is ensured that the labels of subsequent transitions compose. So when we have $\gamma \xrightarrow{\alpha_1} \gamma_1 \xrightarrow{\alpha_2} \gamma_2$ for some $\gamma, \gamma_1, \gamma_2 \in \Gamma$ and $\alpha_1, \alpha_2 \in A$, we have that $\alpha_1; \alpha_2$ holds in the category $\mathbf{A}$. Using the theory of MSOS, we define our formalization as follows.

Definition 4.2. Let $B = \langle \Gamma, \mathbf{A}, \longrightarrow, T \rangle$ be a GTS, then we define our formalization as a tuple $\langle B, \mathbf{\Delta}, \pi, \kappa, \zeta, \rightsquigarrow \rangle$ such that the discrete category with $\mathcal{P}(\Delta)$ as its objects is one of the component categories of $\mathbf{A}$; $\mathbf{\Delta} = (\Delta, lab : \Delta \to \{0, 1\})$ is a structured set of symbols that we call adaptation signals, and lab is a labeling function which maps adaptation

[5] An LTTS is simply an LTS with an extra component representing the terminal configurations, i.e. the configurations for which there are no transition.

signals to either 1 or 0, denoting to apply or not apply recursive activation, respectively; $\pi : \Gamma \to (\mathcal{P}(\Delta) \to \mathcal{P}(\Delta))$ is an adaptation projection function; $\kappa : O \to \mathcal{P}(\Delta)$ is an adaptation activation function with O being the set of objects; $\zeta : \mathcal{P}(\Delta) \to \delta$ is an adaptation selection function; and $\rightsquigarrow \in (\Gamma, \Delta) \times A \times \Gamma$ is an adaptable transition relation. Furthermore, we define the $\twoheadrightarrow$ relation inductively as follows, where $D \diamond D' = \{\delta \mid \delta \in D \cup D' \wedge lab(\delta) = 1\}$ with $D, D' \in \mathcal{P}(\Delta)$, and X denotes the label of the $\twoheadrightarrow$ transition, so X is a morphism of the category **A**.

$$\text{adaptation}\ \frac{\begin{array}{c}\pi(\gamma)(\kappa(source(X))) = D' \\ \zeta(D \cup D') = \delta' \\ D \diamond D' \vdash (\gamma, \delta') \rightsquigarrow \gamma'\end{array}}{D \vdash \gamma \twoheadrightarrow \gamma'}$$

$$\text{default}\ \frac{\begin{array}{c}\pi(\gamma)(\kappa(source(X))) = D' \\ \zeta(D \cup D') = \delta' \\ D \diamond D' \vdash (\gamma, \delta') \not\rightsquigarrow \\ D \diamond D' \vdash \gamma \longrightarrow \gamma'\end{array}}{D \vdash \gamma \twoheadrightarrow \gamma'}$$

These rules state that when there is an active adaptation for the current configuration and the current adaptation signal, then that rule is picked. Otherwise, the transition in the base GTS is used. In addition, we require that the $\rightsquigarrow$ relation respects the terminal configurations of the base GTS. I.e., for all $\gamma \in T$ and for all $\delta' \in \Delta$ we have $(\gamma, \delta') \not\rightsquigarrow$.

4.3.1 Alignment with the Implementation.

To explain our formalization in a bit more detail, we discuss the alignment of the ASOS model and our formalization.

Γ and T represent the terms to evaluate and is reflected in the implementation by a metaclass in the language metamodel. While Γ ranges over configuration with an arbitrary metaclass representing the term, the terminal configurations T are the subset that use metaclasses from the semantic domain structure definition. Adaptation signals(Δ), represent all the combinations of activated adaptation modules. Moreover, before and after adaptations obtain an annotated adaptation signal to differentiate between them within the $\rightsquigarrow$ relation. An adaptation signal is reflected in the implementation by an adaptation interface instance that contains adaptation rules. The addition of two module rules to the interface makes it contain the union of the two sets of rules, hence representing the same set of rules as the adaptation signal of the merge of the signals of the two modules. Thus, the composition in the interface is similar to the adaptation signal merge for composition in formal semantics.

The three functions κ, π, ζ model the feedback loop, adaptation activation, and adaptation selection. The matching on terms in the meta-language to activate an adaptation corresponds to the π function. To ensure that only one signal is active at a time, the ζ function selects one signal based on the current active adaptation signals.

The $\longrightarrow$ relation makes up the original semantics of the programming language and corresponds to the rules outside of adaptation modules. The $\rightsquigarrow$ corresponds to the specialization adaptations as defined in the adaptation modules. The syntax of $\rightsquigarrow$ is similar to $\longrightarrow$ with the addition of the adaptation component, corresponding to the adaptation module that captures the ASOS rule. The Δ component of the $\rightsquigarrow$ ensures that the specialization activates iff the module is activated. For both relations, the arrows in the premises correspond to the $\twoheadrightarrow$ relation. For the $\rightsquigarrow$ relation, premises can also contain before and after steps that model the before and after adaptations. The final semantics of the language is defined by the $\twoheadrightarrow$ relation, which combines the $\longrightarrow$ and $\rightsquigarrow$ relations. The $\twoheadrightarrow$ relation thus corresponds to the *Adaptive Operational Semantics* component in Figure 4.

Finally, the category **A** models the auxiliary entities available to rules. This corresponds to the *Binding*, *Input*, and *Output* components in Figure 4. In the implementation, we implicitly propagate these entities represented by the execution model, the feedback loop state, and the modules state. Therefore, successive modification of these elements forms a trace similarly to label composition in the category.

4.3.2 On Determinism in an ATS.

In our model, determinism is controlled by the designer and not introduced by the adaptation process. In other words, iff the three relations $\Rightarrow$, $\rightsquigarrow$ and $\longrightarrow$ are deterministic, then $\twoheadrightarrow$ is deterministic.

To demonstrate this, we give a proof sketch that shows that for every configuration and for all $\alpha \in A$ we have either $\gamma \not\twoheadrightarrow$ and $\gamma \in T$ or $\exists! \gamma' \in \Gamma$ such that $\gamma \twoheadrightarrow \gamma'$. This is only true whenever we have no derivation tree or we have a unique derivation tree. The first case holds by definition. For the second case, we show that under the assumption, the *adaptation* and *default* rules are deterministic. Let us assume they are not deterministic, then we can construct multiple derivations trees for some $\gamma \in \Gamma$ and some $\alpha \in A$. For the *adaptation* rule we can construct multiple derivation trees whenever one of premises $\pi(\gamma) \circ \kappa(source(X))$, $\zeta(D \cup D')$, or $D \diamond D' \vdash (\gamma, \delta') \rightsquigarrow \gamma'$ has multiple solutions and the resulting conclusion configurations are distinct. By definition, both the first and second component have exactly one solution. So, for some $\delta' \in \Delta$ we have $\exists \gamma_1, \gamma_2 \in \Gamma$ and $\gamma_1 \neq \gamma_2$ such that $D \diamond D' \vdash (\gamma, \delta') \rightsquigarrow \gamma_1$ and $D \diamond D' \vdash (\gamma, \delta') \rightsquigarrow \gamma_2$. However, this contradicts our assumption that $\rightsquigarrow$ is deterministic. The same process can be used for the *default* rule. Finally, the *adaptation* and *default* rule are non-overlapping by definition due to the conflicting premise of the $\rightsquigarrow$ relation. Hence, under the assumption, the $\twoheadrightarrow$ is deterministic.

4.3.3 On Non-Termination in an ATS.

Non-termination in an ATS can arise due to the interplay between the $\rightsquigarrow$ and $\longrightarrow$ relations. For example, we might have $\gamma \to \gamma'$ and then $(\gamma', \delta) \rightsquigarrow \gamma$ for some $\delta \in \Delta$, which can result in non-termination. It might not, because the outside environment, *e.g.,* sensors, can change resulting in a different adaptation or

no adaptation being performed. In addition, such occurrences might be intentional. For example, when a *while* term is adapted but a term in the body of the *while* is not adapted.

So far, we have not yet identified a reasonable restriction on the adaptations that prevents this from occurring. Nevertheless, using our formal model, we can reason about such occurrences, and we aim to utilize model checking to identify adaptations that introduce such sequences.

4.3.4 On Completeness of the Original GTS. With our model, we wanted to retain the completeness of the original GTS This means that for all $\gamma \in \Gamma$ we either have $\exists \gamma'$ such that $\gamma \twoheadrightarrow \gamma'$ or $\gamma \in T$. This holds trivially in our formalization due to the $\twoheadrightarrow$ relation definition and by our requirement that the $\rightsquigarrow$ relation respects the terminal configurations.

5 Evaluation

To evaluate the proposed implementation, we discuss: the applicability of the approach on RobLANG, and the performance overhead.To discuss the applicability , we compare the number of actions performed by the robot with and without adaptations. Finally, we assess the performance of our approach, *i.e.,* self-adaptation at language level, to the performance of the same adaptive behavior written in the program, *i.e.,* self-adaptation at program level.

5.1 ASOS Applicability to RobLANG

To assess the applicability of ASOS to RobLANG, we compare the number of actions performed by the robot until battery depletion with and without adaptations, to show that we can express meaningful adaptations despite the fact that we abstracted the adaptation at language level. In addition, we also evaluate the ability to react to change in the environment by dynamically changing the trade-off at run-time. The adaptation used in this case is the reduction in motor speed discussed in Section 2.3, with the motor running at 75% speed. The action performed by the robot is a movement in square pattern, repeated until the battery depletion. The number of squares completed is 49898 without adaptation and 87475 with adaptation always active. In addition we also manually verified the change of semantic used when changing the trade-off from energy focused to performance focused, later refered as "Switch" configuration. The results shows that the application of the adaptation allowed the robot to perform 1.75 times the number of actions and that the language-level adaptations correctly change based on the context.

5.2 Assessing ASOS Performance

For this experiment, we choose RobLANG, presented in Section 2.3, as object of study. We use two implementations of this language, a self-adaptive one using ASOS, and a classic one using well known tools for DSLs implementation. For both implementations, the abstract syntax of the language was defined using an Ecore [34] metamodel and the concrete syntax using an Xtext [13] grammar. Since these are implementations of the same language, only small changes were made in the syntax, due to the operational semantics implementation method. Both metamodels define concepts that include: (1) Functions definition and calls, (2) Simple arithmetic and boolean predicates, (3) Access to the robot sensors, (4) Effectors to move the robot. The grammars for these concepts are the same for both implementations. However, in the case of ASOS, the structure of the semantic domain (*e.g.,* metaclasses of runtime values, attribute storing dynamic information) needs to be defined in the form of an Ecore metamodel, that is merged with the abstract syntax to define the execution metamodel. In addition, the abstract syntax of the ASOS RobLANG also defines a new statement allowing a language user to define their trade-off and change it at run-time. For the definition of operational semantics, ASOS is used to specify the adaptive version of RobLANG, while the classic version uses Xtend [4] dispatch to define a visitor.

5.2.1 Experimental Setup. To evaluate the overhead of our approach, we compare execution time of a program requesting adaptations. We compare the RobLANG ASOS implementation (*ASOS*) to a manual implementation of the self-adaptation concern at the program level (Program) using the classic implementation. The adaptation used is a reduction in motor speed in robot movement. This adaptation applies depending on the trade-off selected. If *Energy* is more important, the adaptation is applied. If *Time* is more important, it is not applied. We use a program that iteratively moves the robot in a square pattern, and we measure the overhead for three configurations: (i) the adaptation never applies (*Without*), (ii) the adaptation always applies (*With*), and (iii) the adaptation is activated and deactivated periodically at runtime (*Switch*). For each configuration, we measure 30 executions in a row, repeated three times with reboot between each repetition to mitigate the effect of the initial state [18]. Measurements were performed on a computer with 31Gb of RAM and an Intel(R) Core(TM) i7-10850H CPU (12 cores at 2.70GHz) with Manjaro 22.0.5. The language runtimes are executed using the OpenJDK Runtime Environment 11.0.18, and run alone on the computer.

Table 1. Mean time(ms) and 95% confidence intervals, relative speedups, and speedups geometrical mean

Conf.	Implementation		Speed-up
	Program	ASOS	
Without	1245.72 ms [1239.12, 1252.32]	1120.92 ms [1109.65, 1132.20]	x0.90
With	2075.72 ms [2065.50, 2085.95]	1979.91 ms [1968.42, 1991.40]	x0.95
Switch	1817.19 ms [1807.57, 1826.81]	1445.49 ms [1429.55, 1461.42]	x0.80
Speedups Geometrical Mean			x0.88

5.2.2 Results. Table 1 summarizes the performance of both the ASOS implementation and the manual adaptation implementation for each configuration. We compute the mean execution time, the speedup for each configuration, and provide the geometrical mean of these speedups when comparing the two implementations. With an overall speedup of 0.88, the implementation of the self-adaptation concern is more efficient with ASOS. The biggest speedup comes from the *Switch* configuration with a factor of 0.80, followed by the *Without* configuration (x0.90) and finally the *With* configuration (x0.95). These results shows that ASOS does not introduce problematic performances pitfalls, and can even surpass a manual implementation for non-optimized DSLs.

5.2.3 Discussion. First, we can observe the speedup of ASOS compared to the handcrafted adaptation. With the biggest speed-up coming from the *Switch* configuration, we deduce that the implementation of the feedback loop is more efficient using ASOS. This is probably because the usual RobLANG interpreter running the feedback loop is not optimized, whereas the JVM optimizes the feedback loop when using ASOS. The second observation is that the *With* configuration speedup is less important than the *Without* configuration. These two configurations make the same use of the feedback loop, as their trade-off does not change. In both cases, the handcrafted version performs a call to the speed-setter statement. Hence, this difference in the speedups comes from the difference in performance to call an adaptation compared to the original rule. To conclude that these hypotheses are true, further experimentation needs to be done.

6 Related Work

Adaptable Systems. In this paper, we have introduced the idea of adaptable structural operational semantics to capture the essence of adaptable languages. Earlier work on describing adaptable systems exists. Adaptable interface automata [6] are an extension of interface automata [11] with atomic propositions that model state observations. Adaptations are then transitions where the two states of the transition give different results for some (or all) proposition(s). Self-Adaptive Abstract State Machines [3] use multi-agent abstract state machines to formalize self-adaptable systems. Compared to our approach, the adaptive system is not centralized but is distributed among several agents. There are two types of agents: managing and managed. Synchronous Adaptive Systems [1] is another approach to the formalization of adaptive systems. In this approach, a system is divided into several modules which can be in different configurations — each representing a different behavior. Configurations are activated and deactivated via guards — somewhat resembling the matching in our approach. A rewriting approach [7] is used by modeling self-adaptive systems in Maude [10] relying on computational reflection [22]. The approach takes an unbounded layered approach in which (partial) knowledge flows downwards and effects flow upwards. A layer can modify the rules of the layer below it, modeling adaptation in the approach. Recurring in these different approaches is a separation of an adaptable system in two or more layers. This idea is also present in our approach, exemplified by the two transition relations in our approach. The idea behind ASOS clearly falls within the line of adaptable interpreters [8] that enable the creation of dynamic systems. The ASOS approach, by proposing semantics based on MSOS, additionally enables the construction of verification tools for language engineers.

Abstraction at Language Level. The quest to abstract non-functional properties (*e.g.*, adaptability, security) and incorporate their effects in the behavior of an existing application has been a longstanding endeavor. This approach is rooted in Model-Driven Engineering methodologies that aim to provide a software creation process through a series of transformations, enabling the specialization of such properties [33]. It is also rooted in modern programming languages where annotations/attributes may be used to abstract non-functional features as first level entities [28]. Additionally, this is also observed in the community of dynamic software product lines [12, 14] and their implementation based on Aspect-Oriented Programming (AOP) [20], thereby allowing the weaving of non-functional concerns based on software design decision. In these approaches, we recognize the quest to abstract non-functional concerns from the design phase carried out by a domain expert. While certain approaches have focused on investigating the correctness of system behavior for various configurations [24], the ability to reason compositionally about this correctness remains limited [23]. Abstracting the adaptation concern at the language level, while providing a clear semantics for the composition of adaptation modules with a base program, allows the language designer to reason about the impact of an adaptation module on a set of behavioral properties of a base program written using an ASOS-defined DSL.

7 Conclusion and Future Work

This paper proposes the ASOS framework to define modular and adaptable semantics of a DSL. ASOS paves the way for checking determinism, completeness, and termination properties based on the proposed formal semantics. ASOS also provides the possibility of generating an implementation of a modular and adaptable interpreter based on SEALS, an implementation framework for adaptable interpreters.

Perspectives of this work are (1) evaluating the complexity for a language designer to use ASOS , (2) allowing the definition of correctness envelopes at the rule level, (3) allowing the configuration of the feedback loop in ASOS, and (4) showing that the declarative nature of ASOS rules allows language composition, facilitating the construction of self-adaptable language fragments, enabling the scenarios where a DSL is built by assembling existing language fragments.

References

[1] Rasmus Adler, Ina Schaefer, Tobias Schüle, and Eric Vecchié. 2007. From Model-Based Design to Formal Verification of Adaptive Embedded Systems. In *Formal Methods and Software Engineering, 9th International Conference on Formal Engineering Methods, ICFEM 2007, Boca Raton, FL, USA, November 14-15, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4789)*, Michael J. Butler, Michael G. Hinchey, and María M. Larrondo-Petrie (Eds.). Springer, 76–95. https://doi.org/10.1007/978-3-540-76650-6_6

[2] Anwar Al-Mofleh, Soib Taib, Wael Salah, and Mokhzaini Azizan. 2008. Importance of Energy Efficiency: From the Perspective of Electrical Equipments. In *Proceedings of the 2nd International Conference on Science and Technology (ICSTIE)*.

[3] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. 2015. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, Paola Inverardi and Bradley R. Schmerl (Eds.). IEEE Computer Society, 13–23. https://doi.org/10.1109/SEAMS.2015.10

[4] Lorenzo Bettini. 2011. A DSL for writing type systems for Xtext languages. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*. 31–40. https://doi.org/10.1145/2093157.2093163

[5] Victor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel. 2015. Morph: A reference architecture for configuration and behaviour self-adaptation. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*. 9–16.

[6] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. 2013. Adaptable transition systems. In *Recent Trends in Algebraic Development Techniques: 21st International Workshop, WADT 2012, Salamanca, Spain, June 7-10, 2012, Revised Selected Papers 21*. Springer, 95–110.

[7] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. 2015. Modelling and analyzing adaptive self-assembly strategies with Maude. *Sci. Comput. Program.* 99 (2015), 75–94. https://doi.org/10.1016/j.scico.2013.11.043

[8] Walter Cazzola and Albert Shaqiri. 2016. Dynamic software evolution through interpreter adaptation. In *Companion Proceedings of the 15th International Conference on Modularity*. 16–19.

[9] Martin Churchill and Peter D Mosses. 2013. Modular bisimulation theory for computations and values. In *Foundations of Software Science and Computation Structures: 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 16*. Springer, 97–112.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott (Eds.). 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, Vol. 4350. Springer. https://doi.org/10.1007/978-3-540-71999-1

[11] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, A Min Tjoa and Volker Gruhn (Eds.). ACM, 109–120. https://doi.org/10.1145/503209.503226

[12] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. 2010. A dynamic software product line approach using aspect models at runtime. In *5th Domain-Specific Aspect Languages Workshop*. Citeseer.

[13] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: implement your language faster than the quick and dirty way. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 307–309. https://doi.org/10.1145/1869542.1869625

[14] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic software product lines. *Computer* 41, 4 (2008), 93–95.

[15] M Usman Iftikhar and Danny Weyns. 2014. Activforms: Active formal models for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 125–134.

[16] Paola Inverardi and Massimo Tivoli. 2009. The future of software: Adaptation and dependability. *Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures* (2009), 1–31.

[17] Gwendal Jouneaux, Olivier Barais, Benoit Combemale, and Gunter Mussbacher. 2021. SEALS: a framework for building self-adaptive virtual machines. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. 150–163.

[18] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005. Benchmark precision and random initial state. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)*. 484–490.

[19] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan 2003), 41–50.

[20] Gregor Kiczales. 1996. Aspect-oriented programming. *ACM Computing Surveys (CSUR)* 28, 4es (1996), 154–es.

[21] Jeff Kramer and Jeff Magee. 2007. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE'07)*. IEEE, 259–268.

[22] Pattie Maes. 1988. Computational reflection. *Knowl. Eng. Rev.* 3, 1 (1988), 1–19. https://doi.org/10.1017/S0269888900004355

[23] Andreas Metzger and Klaus Pohl. 2014. Software Product Line Engineering and Variability Management: Achievements and Challenges. In *Future of Software Engineering Proceedings* (Hyderabad, India) *(FOSE 2014)*. Association for Computing Machinery, New York, NY, USA, 70–84. https://doi.org/10.1145/2593882.2593888

[24] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jézéquel. 2009. Taming dynamically adaptive systems using models and aspects. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 122–132.

[25] Peter D Mosses. 2004. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming* 60 (2004), 195–228.

[26] Peter D Mosses and Mark J New. 2009. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science* 229, 4 (2009), 49–66.

[27] Hiroyuki Nakagawa, Akihiko Ohsuga, and Shinichi Honiden. 2012. Towards dynamic evolution of self-adaptive systems based on dynamic updating of control loops. In *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 59–68.

[28] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.

[29] Gordon D Plotkin. 1981. *A structural approach to operational semantics*. Aarhus university.

[30] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 17–139.

[31] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. 2010. Matching logic: An alternative to Hoare/Floyd logic. In *International Conference on Algebraic Methodology and Software Technology*. Springer, 142–162.

[32] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[33] Douglas C Schmidt et al. 2006. Model-driven engineering. *Computer-IEEE Computer Society-* 39, 2 (2006), 25.

[34] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework.* Pearson Education.

[35] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. 2010. PLASMA: a plan-based layered architecture for software model-driven adaptation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 467–476.

[36] Kevin Twidle, Naranker Dulay, Emil Lupu, and Morris Sloman. 2009. Ponder2: A policy system for autonomous pervasive environments. In *2009 Fifth International Conference on Autonomic and Autonomous Systems*. IEEE, 330–335.

[37] Thomas Vogel and Holger Giese. 2012. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 129–138.

Received 2023-07-07; accepted 2023-09-01