SEALS: A framework for building Self-Adaptive Virtual Machines

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

Olivier Barais
Univ. Rennes, Inria, IRISA
Rennes, France
olivier.barais@irisa.fr

Benoit Combemale
Univ. Rennes, Inria, IRISA
Rennes, France
benoit.combemale@irisa.fr

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

Abstract

Over recent years, self-adaptation has become a major concern for software systems that evolve in changing environments. While expert developers may choose a manual implementation when self-adaptation is the primary concern, self-adaptation should be abstracted for non-expert developers or when it is a secondary concern. We present SEALS, a framework for building self-adaptive virtual machines for domain-specific languages. This framework provides first-class entities for the language engineer to promote domain-specific feedback loops in the definition of the DSL operational semantics. In particular, the framework supports the definition of (i) the abstract syntax and the semantics of the language as well as the correctness envelope defining the acceptable semantics for a domain concept, (ii) the feedback loop and associated trade-off reasoning, and (iii) the adaptations and the predictive model of their impact on the trade-off. We use this framework to build three languages with self-adaptive virtual machines and discuss the relevance of the abstractions, effectiveness of correctness envelopes, and compare their code size and performance results to their manually implemented counterparts. We show that the framework provides suitable abstractions for the implementation of self-adaptive operational semantics while introducing little performance overhead compared to a manual implementation.

ACM Reference Format:
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 (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486608.3486912

SLE '21, October 17–18, 2021, Chicago, IL, USA

1 Introduction

Software systems increasingly need dynamic self-adaptation to best deliver their expected services [8], because these systems are evolving in complex environments on which they possibly are highly dependent. Self-adaptation requires a feedback loop to react to changes and trade-off analysis to determine the best course of action in the current context[24].

Expert developers may choose a manual implementation of the self-adaptation functionality, when self-adaptation is the primary system concern (*e.g.,* systems like autonomous cars (*e.g.,* Waymo[1]) and large-scale video streaming platforms (*e.g.,* Netflix[2])). However, there are many systems where self-adaptation is a secondary but nevertheless important concern for language users[3] to provide more tailored services to end users. In those cases, language users may not have the expertise or may not need (want) to explicitly deal with this complex concern. Hence, there is a need to abstract the self-adaption concern into high level language constructs for system development in the same way as software languages abstracted concerns like concurrency and parallelism [17, 44]. Language users of such systems can then exploit the enhanced support for this emerging concern, *i.e.,* the implementation of feedback loops and trade-off reasoning. For example, in the context of green IT [32], many systems like e-commerce applications can benefit from trade-offs related to sustainability and balance their provided services accordingly. More generally, there is a growing trend to provide systems capable of reasoning about trade-offs (*e.g.,* energy, time, cost, quality) to a large audience [26, 35].

We argue that language engineers will be expected to provide functionality for self-adaptation in their domain-specific languages. Therefore, language engineers can benefit from a framework at the language level, *i.e.,* a more formal approach to the specification of feedback loops and trade-off analyses to adapt a software language and better support the

[1]Cf. https://waymo.com/
[2]Cf. https://www.netflix.com/
[3]We use *language users* as a broad term that includes modelers and programmers using a given software (modeling or programming) language.

implementation of complex self-adaptable software systems. Over the last decades, the software engineering community proposed an important body of knowledge about the design and implementation of self-adaptable systems [8]. This body of knowledge is now mature enough to understand well the main concepts and associated architectures for feedback loops and trade-off analyses. Architectural patterns such as the MAPE-K loop have been proposed to structure their implementation [24]. Trade-off analysis is supported with dedicated modeling infrastructure (*e.g.,* models@runtime [6], goal modeling [40, 42, 48]).

This paper presents the *SElf-Adaptive LanguageS Framework* (SEALS), where the main objective is to abstract from the feedback loop of the self-adaptive system as much as possible, so as to free the language users from the detailed specification or implementation of the feedback loop. While not being model-driven approach per se, we choose to use appropriate models to describe parts of the framework (*e.g.* goal models for trade-off analysis). SEALS supports the definition of (i) the abstract syntax and the semantics of the language as well as the correctness envelope defining the acceptable semantics for a domain concept, (ii) the feedback loop and associated trade-off reasoning, and (iii) the adaptations and the predictive model of their impact on the trade-off. We use SEALS to build three languages with self-adaptive virtual machines: MiniJava (a representative of imperative general purpose languages), RobLANG (a representative of imperative domain-specific languages), and HTML (a representative of declarative domain-specific languages). We further discuss the relevance of the introduced abstractions, the effectiveness of the correctness envelopes, and compare their code size and performance results to their manually implemented counterparts. We conclude that the framework provides suitable abstractions for the implementation of self-adaptive operational semantics while introducing little performance overhead compared to a manual implementation.

In the remainder of the paper, we introduce the concept of *Self-Adaptable Virtual Machines* (SAVMs) and provide a motivating example in Section 2. We then present SEALS, our proposed framework, by giving an overview in Section 3 and detailing the implementation in Section 4. We report on the experiments we performed and the resulting evaluation of our framework in Section 5, respectively. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

2 Background and motivating example

In this section, we introduce the concept of Self-Adaptable Virtual Machines. SAVMs are a sub-set of Self-Adaptable Languages (SAL) [21] that incorporate a feedback loop in language operational semantics. Hence, we first provide background information on the implementation of language operational semantics and self-adaptive system design. Second, we present the self-adaptable HTML language which will be used as an example in the remainder of the paper.

2.1 Language operational semantics implementation

From a language user's point of view, a software language (for specification, modeling, or programming) is a specific syntax and a set of associated services (editor, checkers, simulator, compiler...). However, from a language engineer's point of view, all these services are usually obtained from abstract specifications of the language concerns that include the abstract syntax to define the language constructs, the concrete syntax to attach a textual or graphical representation to these constructs, and the semantics to provide precise meaning to these constructs. Since semantics, and more precisely operational semantics, are at the core of our approach, we review their implementation in this section.

Operational semantics are often defined as an interpreter that performs the computation by traversing the Abstract Syntax Tree (AST) and reports the result from the leaves of the tree to the root. Multiple patterns were proposed by the community and differ from one to another for reasons like performance, modularity, or reusability. Classical patterns such as Interpreter and Visitor patterns [15] provide a straight-forward way to implement the interpreter. The former facilitates the extension of the abstract syntax and the latter the extension of the operational semantics. The Eclipse Modeling Framework (EMF) also provides an implementation pattern, EMF's Switch, based on the Visitor pattern but uses Run-Time Type Information (RTTI) to select the method to call rather than the `accept` method . However, while patterns like Visitor or EMF's Switch ease the extension of the semantics, they hamper the extension of the abstract syntax (and vice versa for the Interpreter pattern). This problem called *The language extension problem* [30] led to the emergence of more modular ways to define semantics through new patterns (e.g., Object Algebras [36], Revisitor [29]) or dedicated meta-languages (e.g., MSOS [34], Kermeta [20]).

Our proposed framework relies on parts of those design patterns and meta-languages to (1) have a base pattern in which we introduce the adaptation concern and (2) provide abstractions to the language engineer aligned with the existing interpreter implementation patterns.

2.2 Design of Self-Adaptive Systems

Self-Adaptive Systems are systems capable of adapting their behavior in response to changes in the environment or in the system itself, hence, can be classified as intelligent agents [38]. Intelligent agents perceive their environment through sensors, decide their actions with respect to the current environment, and act upon the environment through effectors. This succession of steps define a classic feedback loop scheme of *Collect, Analyze, Decide, and Act* [12].

The definition of a feedback loop has been investigated in the context of Self-Adaptive Systems [8], with a field mature enough to provide time-honored patterns such as the MAPE-K loop [24]. The MAPE-K loop provides a pattern

to implement a feedback loop in terms of four main functions (***M***onitoring, ***A***nalysis, ***P***lanning, and ***E***xcecution) and a common ***K***nowledge. All the functions can be supported by models, with each model playing one or more roles with respect to the model's purpose. It is *descriptive*, if its purpose is the documentation of current or past system aspects, thus facilitating understanding and enabling analysis. It is *predictive*, if its purpose is the prediction of information that one cannot or does not want to measure, hence creating new knowledge and allowing for decision-making and trade-off analyses. It is *prescriptive*, if its purpose is the description of the system to be built, hence driving the constructive process including runtime evolution in the case of self-adaptive systems. These roles apply to models of all types (*e.g.,* engineering models, scientific models, and machine learning models), and has been exemplified in the context of the MAPE-K loop [9].

In our approach, we use the MAPE-K Loop pattern to abstract the feedback loop implementation from the language engineer. This abstraction allows a good separation of concern between the steps performed and their interactions, and between the four main functions themselves.

2.3 Self-Adaptable HTML

Nowadays more than ever, information and communications technology (ICT) electricity consumption grows higher and higher. The electricity demand of the ICT is expected to represent 21% of the world electricity demand in 2030, ranging from 8% in the best case to 51% in the worst case [3]. Comparatively, the web electricity demand was representing around 2% of 2015 world electricity demand [16]. For this reason, we choose to build an HTML interpreter as a self-adaptable virtual machine that performs a trade-off between energy consumed to display the web page and the quality of the page rendering. The energy consumption to display a web page (display and data transfer) is difficult to assess due to the networking part. However, WebsiteCarbonCalculator [43] provides an algorithm that estimates this consumption, and this consumption is proportional to the size of the transferred data. Hence, the effective trade-off performed is between rendering quality and the size of data transferred. The expected trade-off is specified by the end-user due to the subjective character of the rendering quality.

Consider three adaptations for the self-adaptable HTML that serve as examples, two with loss of information and one without. The first is the conditional loading of resources depending on their URL. The idea for this adaptation is to keep the content from the website and remove external resources that are less prone to deliver important content. The second is HTML lists perforation according to their size. In HTML, lists tends to represent sets of items that are semantically similar. Often, those lists are generated from data that share the same nature, but are independent and self-sufficient, *e.g.,* blog posts or emails. The goal is to reduce the number of these elements and subsequent data requests like images.

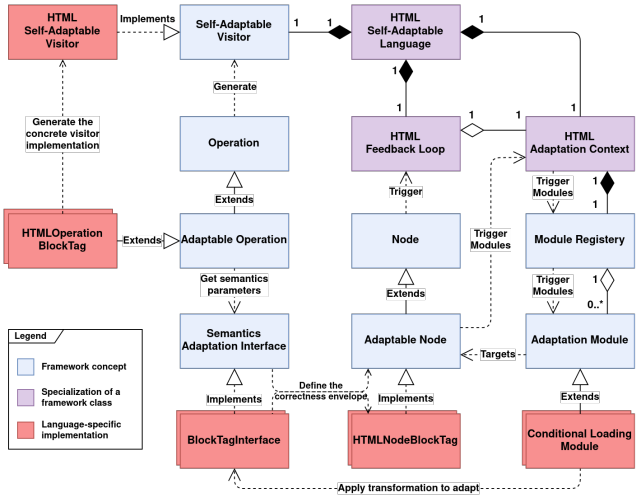


Figure 1. Approach overview on the HTML use case

The last one is the degradation of an image to reduce its size. The idea is to request a degraded version of an image if its size exceeds a certain threshold.

3 Approach overview

This section presents a general overview of our approach and describes the conceptual model of the framework. The HTML language introduced in Section 2 will be used in the remainder of this section to illustrate the use of the framework. Figure 1 presents the general overview of our approach on this example. In this figure, the three colors represent the three types of elements used to define a self-adaptable virtual machine with SEALS. The color blue is used to represent elements defined in the framework, the purple elements represent specializations of generic components provided by the framework, and red elements represent domain concepts defined using the framework constructs.

This section is structured around the self-adaptable virtual machine implementation workflow. First, we present the modeling of the domain concepts (Section 3.1). Then, the specialization of the adaptation process (Section 3.2). Finally, we discuss the coordination of all these components by the framework (Section 3.3).

In our approach, adaptations are developed as modules and are not attached to a fixed position in the presented workflow. To better understand their integration and considering that adaptations are domain-specific, we choose to present them with the modeling of the domain concepts in Section 3.1.

3.1 Modeling of domain concepts

The definition of a software language includes the definition of domain concepts in the form of an abstract syntax and semantics. To illustrate this aspect of the approach, we take the HTML domain concept of BlockTag (i.e., a tag that contains

child elements) as an example. In SEALS, a domain concept can be either a normal concept or an adaptable concept.

For each concept of the language, the language engineer starts by defining the abstract syntax using the *Node* and *AdaptableNode* constructs. The BlockTag concept abstract syntax is defined as an *AdaptableNode* named *HTMLNodeBlockTag*. In addition, an adaptable concept also needs to define its correctness envelope, i.e., the set of acceptable variations in its semantics, using the *SemanticsAdaptationInterface* abstraction. Since BlockTag is an adaptable concept, the *BlockTagInterface* needs to be defined. Finally, the language engineer implements the operational semantics using the *Operation* and *AdaptableOperation* constructs. An *AdaptableOperation* has access to a *SemanticsAdaptationInterface*, i.e., a valid configuration conforming to the correctness envelope and resulting from an adaptation and uses it to define the adaptable semantics of the concept. In our example, *HTMLOperationBlockTag* use the configuration specified by *BlockTagInterface* to define an adaptable operational semantics for the BlockTag concept. Those operations will be used by the framework to generate a concrete *Self-Adaptable Visitor* for the language, like the *HTML Self-Adaptable Visitor*.

In our approach, we consider adaptations to be domain-specific, as they apply to the semantics of domain-specific concepts. To help with the definition of those adaptations, we provide a construct (*AdaptationModule*) to define adaptations as modules by specifying the adaptation and in which case one should call this adaptation. The *Conditional Loading Module* is an example of such a module that defines the condition to skip the loading of some elements, and this module is applied, among others, on the *script* elements represented as *HTMLNodeBlockTag*.

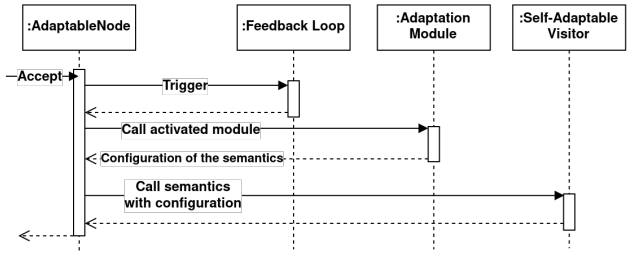
3.2 Specializing the adaptation process

At the core of self-adaptable virtual machines, there is a feedback loop that performs trade-off analysis based on a monitored environment. To support the language engineer in the conception of this adaptation process, we provide two constructs that need to be specialized. They represent the *Feedback Loop* and its *Adaptation Context*.

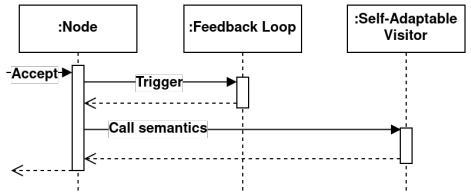
First, the language engineer needs to specialize the *Adaptation Context* by defining the monitored environment, the properties of interests, and the expected trade-off. The *HTML Adaptation Context* is the specialization for the HTML language. It specifies the "quality" and "energy" properties of interest, a void monitored environment, and delegates the expected trade-off to the end-user. In addition, the *HTML Adaptation Context* benefits from the capabilities of the generic *Adaptation Context* to register and manage the adaptation modules. These capabilities take the form of a *Module Registry*. Then, the language engineer specializes the *Feedback Loop* to implement how to perform the trade-off analysis for the language. For instance, the *HTML Feedback Loop* represents this implementation for the HTML language.

3.3 Coordination of all components

In order to coordinate and initialize the SAVM , the language engineer has to specialize the *Self-Adaptable Language* concept. This specialization needs to define how to instantiate the concrete *Feedback Loop*, *Adaptation Context*, and *Self-Adaptable Visitor*. Hence, the *HTML Self-Adaptable Language* defines how to create the *HTML Feedback Loop*, *HTML Adaptation Context*, and *HTML Self-Adaptable Visitor*.



(a) Execution of an *AdaptableNode*



(b) Execution of a *Node*

Figure 2. Sequence diagrams representing components interactions when executing a node semantics

The interactions of all the components presented in this section occur during the execution of a *Node*. Figure 2 presents the interactions of components during the execution of a node semantics in the form of two sequence diagrams, one for *AdaptableNodes* and the other for *Nodes*. Since our operational semantics implementation pattern is based on the Visitor pattern, the execution is launched by the call to an `accept` function. This `accept` function implementation is abstracted from the language engineer and implemented at the *AdaptableNode* and *Node* level. First, the `accept` function will trigger the feedback loop to perform, if needed, the monitoring and trade-off analysis. Then, if the node is an *AdaptableNode*, the function will call the activated adaptation modules that offer adaptations for this type of node to obtain an adapted configuration for the node semantics conforming to its correctness envelope. Finally, the `accept` function calls the semantics for this node in the visitor generated from the specified *Operation* and *AdaptableOperation*. In the case of an *AdaptableNode*, the function will also provide the configuration proposed by the modules to the visitor and the underlying *AdaptableOperation*.

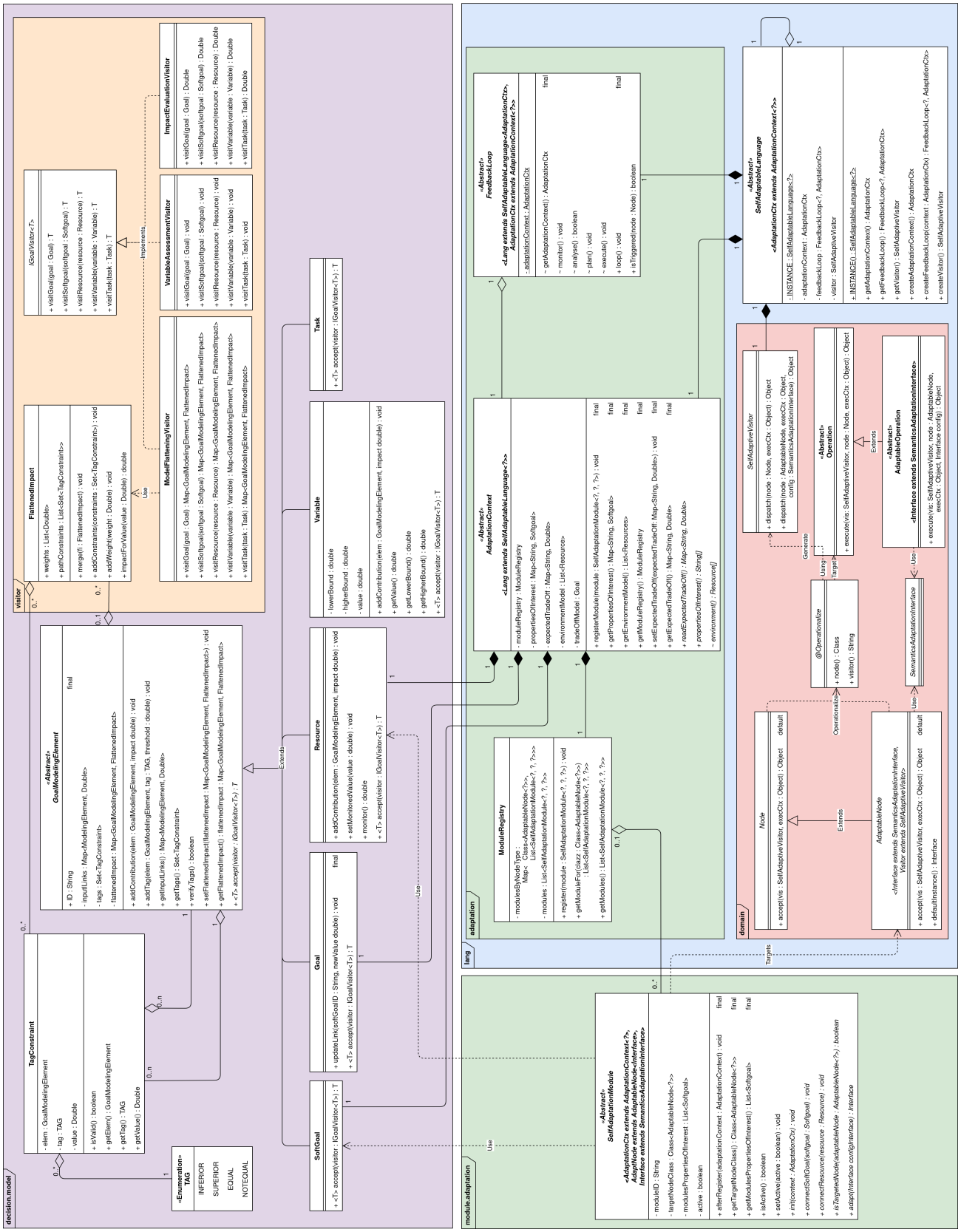


Figure 3. Class diagram of the SEALS framework

4 Implementation

In this section, we detail the current Java implementation of the SEALS framework. Figure 3 presents the class diagram of the whole framework. Colored boxes represent Java packages with particular functions. Purple (*decision.model*) encompasses the elements related to the decision model including, in yellow (*visitor*), the processing of the model. Blue (*lang*) represents the definition of the language with, in red (*domain*), the definition of the domain abstract syntax and semantics and, in green (*adaptation*), the language components related to the adaptation (e.g., feedback loop, adaptation modules). In this section, the use of *emphasis* denotes a references to a concept defined in Figure 3. The remainder of this section presents these elements and their interactions. As for Section 3, this section is structured around the workflow for the implementation of a self-adaptable virtual machine. First, we present the implementation of the domain concepts, using abstractions defined in the red `lang.domain` package of Figure 3 (Section 4.1). Second, we detail the specialization of the *Feedback Loop* and its *Adaptation Context* presented in Figure 3 as part of the green `lang.adaptation` package (Section 4.2). Then, we detail the implementation of adaptation modules, the second green `module.adaptation` package in Figure 3 (Section 4.3). Finally, we discuss the connection of all these components using the *Self-Adaptable Language* concept, presented in the blue `lang` package of Figure 3, and how to use a *Self-Adaptable Language* to create an executable for the language (Section 4.4).

4.1 Implementation of domain concepts

This section presents the concrete methodology to implement the domain concepts in SEALS, i.e., implementing the abstract syntax, the correctness envelope, and the operational semantics as presented in Section 3.1. The abstractions used to define those concepts are regrouped in the `domain` package represented in red in Figure 3.

Abstract Syntax. The definition of the abstract syntax is done by defining classes that represent the concepts of the language. These classes will later be instantiated in the form of an Abstract Syntax Tree (AST) to represent a program to compute. SEALS proposes two Java interfaces to define the classes that represent a node of the AST: *Node* and *AdaptableNode*. The former does not provide anything to the language engineer but is needed by the framework to understand that the specialization represents an AST node. On the other hand, the latter asks the language engineer to parameterize the node (using Java generics) by the Java type representing its correctness envelope. In addition, *AdaptableNode* also needs the language engineer to implement the `defaultInterface` function that returns an instance of the correctness envelope representing its values for the normal execution of the current instance of the node. This default interface represents the configuration that can be changed by adaptation modules and will be provided to the visitor, as mentioned in Section 3.1.

Correctness Envelope. The correctness envelope defined for an *AdaptableNode* takes the form of a Java class implementing the *SemanticsAdaptationInterface*. This class represents the variation points for the semantics of the node and provides the capabilities to manipulate them.

```
1  public class ForInterface implements
2                          SemanticsAdaptationInterface {
3      private int increment;
4
5      public ForInterface (int increment){
6          this.increment = increment;
7      }
8      public void changeIncrement(int inc) {
9          if(inc*increment > 0){//do not change sign
10             increment = inc;
11         }
12     }
13 }
```

Listing 1. Example of correctness envelope for a ForNode

For instance, Listing 1 presents a possible correctness envelope for a `ForNode`. The variation point presented is the progression of the loop represented by the variable `increment`. The `changeIncrement` function allows adaptations to manipulate the increment while a guard for the proposed value ensures a coherent value (e.g., do not change the sign of the increment to avoid going the wrong way). To manipulate this envelope, the adaptations need an instance of this type for the current node. The goal of the `defaultInterface` function is to provide this instance, with the particularity that the variation points are set to reflect the normal semantics of the current node. For instance, in Listing 2 the value of `increment` represents the value specified in the code (e.g., one if the progression is `i++`).

```
1 public ForInterface defaultInterface() {
2     int inc = getIncrementFromExpression(
3                 this.getProgression());
4     return new ForInterface(inc);
5 }
```

Listing 2. Example of `defaultInterface` implementation

```
1 public Object execute(SelfAdaptiveVisitor vis,
2   Node n, Object execCtx);
3 public Object execute(SelfAdaptiveVisitor vis,
4   AdaptableNode n, Object execCtx, Interface config);
```

Listing 3. Signatures of normal and adaptive operations

Operational Semantics. Complementary to the abstract syntax *Node* and *AdaptableNode*, the semantics of the language is defined by specializing the *Operation* and *AdaptableOperation* abstract classes. These classes define the signatures for the semantics of a normal node and for an adaptable node, respectively (see Listing 3).

The signature for normal semantics contains (1) the visitor to execute children of the current node in the Abstract Syntax Tree, (2) the current node to perform the semantics depend-

ing on the node attributes, and (3) the execution context. On the other hand, the signature for adaptable semantics also includes an instance of the correctness envelope representing the configuration of the semantics for this node's current execution. The configuration provided to the *AdaptableOperation* is the `defaultInterface` after potential transformation performed by adaptation modules as presented in Figure 2a.

To link the operations to the concerned node and to a concrete visitor, we provide the *Operationalize* Java annotation for the *Operation* class. This annotation takes two arguments: the operationalized node class and the fully qualified name of the visitor to generate including this operation. This generated visitor implements the *Self-AdaptableVisitor* interface and provides two dispatch functions, one for nodes and one for adaptable nodes, allowing the generic call to the operations from the *Node* and *AdaptableNode* interfaces and freeing the language engineer from implementing the dispatch by hand.

4.2 Implementing adaptation and decision process

At the core of self-adaptable virtual machines, there is a feedback loop. To abstract the implementation of this concern from the language engineer, we provide an abstract feedback loop based on the MAPE-K pattern that need to be specialized. This abstraction allows the language engineer to define the feedback loop in the form of the four steps of the MAPE-K loop, **M**onitor, **A**nalyse, **P**lan, **E**xecute when specializing the *FeedbackLoop* abstract class, and the **K**nowledge when specializing the *AdaptationContext*. Those abstractions are part of the `lang.adaptation` package represented in green in Figure 3. This *AdaptationContext* specialization needs to provide information on (1) a way to read the expected trade-off, (2) the environment monitored, and (3) the managed properties of interest. The first point allows the language engineer to specify directly a trade-off for the language in this function, or to delegate the trade-off definition to other stakeholders (e.g., language users, end users) by implementing the configuration system here. The other two points are specified using the modeling framework (purple box in Figure 3) that will be used to perform the trade-off analysis. The managed properties of interest are defined through a trade-off model, while the environment monitored is defined in the environment model.

For this modeling framework, we choose to use goal models to model the satisfaction of high-level goals depending on the contribution of other elements (e.g., adaptations, resources). These goal models are defined using GRL [40] to model the trade-off, as well as the contribution of the adaptations to the satisfaction of the underlying properties of interests. Three types of goal models are involved in the analysis: a trade-off model, an environment model, and one or more impact models. The first two are the previously mentioned parts of the adaptation context, while the last ones are defined within the modules and will be detailed in Section 4.3. When merged together those models form a global model representing the system allowing to perform the trade-off analysis on it. Our modeling framework uses goal models, however, any trade-off analysis process could replace it. The generalization of the modeling framework to use other trade-off analysis strategies is left as future work.

In addition to the *AdaptationContext* specialization, the *FeedbackLoop* specialization requires the language engineer to implement the four steps of the MAPE-K loop. **M**onitoring consists of observing the system and updating the environment model, **A**nalysis and **P**lanning consists of performing the trade-off analysis and choosing the adaptation modules to be applied, and **E**xecution activates or deactivates the modules. Moreover, the language engineer is also in charge of implementing the *isTriggered* function. In Figure 2, this function is hidden in the "Trigger" arrow and informs the framework if the monitor, analyze, plan, and execute functions need to be executed. The dynamic evaluation of the iteration launching condition allows the language engineer to define conditions based on the nodes executed or to be executed, as well as time related properties. In the end, the *FeedbackLoop* abstraction frees the language engineer from the interactions between the steps of the MAPE-K loop as well as the explicit calls to the MAPE-K loop.

4.3 Design of adaptation modules

In addition to the language in itself and the feedback loop, a self-adaptable virtual machine is composed of adaptations. To allow different stakeholders to implement the domain-specific adaptations, we choose a modular approach for the definition of the adaptations. To do so, we provide an abstract class representing a module that performs an adaptation (in the green `module.adaptation` package in Figure 3). When specializing, the adaptation developer specifies the class of the targeted nodes and needs to implement: (1) the initialization of the module, (2) the exposition of a predictive model of the impact on the properties of interest, (3) an additional filter on the targeted nodes, and (4) the adaptation.

The predictive model exposed by the module is an impact model representing how the activation of this module will affect the properties of interest. This model is connected at runtime to the trade-off model and environment model, allowing the adaptation developer to specify the impact on the properties of interest in the trade-off model based on the monitored resources of the environment model.

When calling the accept function of an adaptable node, the framework will prepend the call to the visitor with the module calls as presented in Figure 2a. First, we ask for the default correctness envelope instance and for the list of modules applicable on this node. Then, for each module, if the module is active and the additional filter allows the adaptation, the module adaptation function will transform the current instance of the correctness envelope. Finally, this instance is passed to the visitor during the call to the dispatch

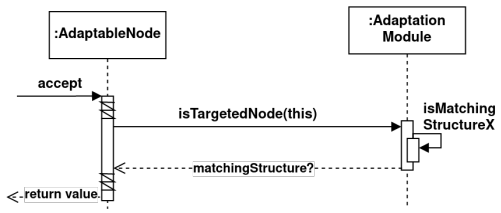


Figure 4. Sequence diagram of the module additional filter

function of adaptable nodes. This call mechanism frees the developers from calling the adaptation modules explicitly.

In addition, this mechanism allows fine-grained control over the nodes targeted by the adaptation, but also over the nodes that can be adapted by a module. In the first case, the filtering function allows the module developer to check properties of the node and apply or not the adaptation (Figure 4).

```
public class ForInterface implements
                        SemanticsAdaptationInterface {
    private int increment;

    public ForInterface (int increment){
        this.increment = increment;
    }
    public void changeIncrement(int inc) {
        if(inc*increment > 0){//do not change sign
            increment = inc;
        }
    }
}
public class ImmutableInterface extends ForInterface {
    @Override public void changeIncrement(int inc) {}
}
```

Listing 4. Mutable and immutable correctness envelopes

```
public ForInterface defaultInterface() {
    int inc = getIncrementFromExpression(
                this.getProgression());
    if(isMatchingStructure()){
        return new ForInterface(inc);
    } else {
        return new ImmutableInterface(inc);
    }
}
```

Listing 5. Implementation of `defaultInterface` filter

For the second case, the language engineer can define two types of correctness envelopes. The first to represent the adaptation possibilities, and the second that overrides it to remove the capacity to change values. Listing 4 presents the two correctness envelopes for the ForNode.

In addition to the definition of the correctness envelopes, the language engineer needs to filter the targetable nodes in the `defaultInterface` function to return the mutable/immutable version of the correctness envelope. Listing 5 provides an example of the implementation of this function.

As mentioned before, the adaptation is performed through the transformation of an instance of a correctness envelope. There are multiple ways to define and transform a correctness envelope, but we want to highlight two possible

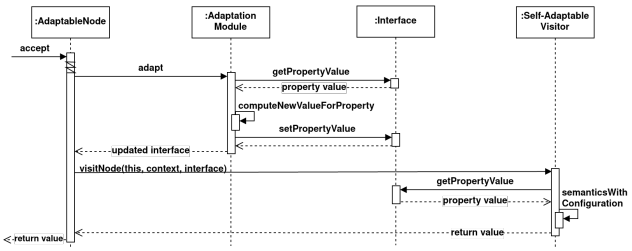


Figure 5. Sequence diagram of the node semantics parameterization

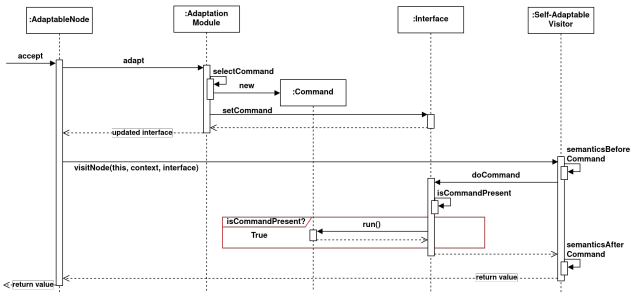


Figure 6. Sequence diagram of the command injection

ways that cover the majority of the possible adaptations. The first one is the semantics parameterization. The correctness envelope defines parameters for the semantics that can be changed by the adaptations (see Figure 5).

The second allows the injection of commands. In addition to the correctness envelope, the language engineer also defines a `Command` class with a `run` function to be called in the semantics. In this case, the module developer has more freedom to implement the adaptation, but the language engineer still controls the information given to the module and when to call it in the semantics. This pattern can be used to add commands before and after the semantics, in the middle, or even replace the whole semantics by re-implementing the operation in the command. Figure 6 depicts a sequence diagram for a generic use case of the command injection.

4.4 Assembly and execution

The *Self-Adaptable Language* abstract class has two roles in the design of a self-adaptable virtual machine: assemble all the component together and ensure a correct initialization, and provide an interface to interact with the language runtime to create a standalone executable of the virtual machine or integrate the language in another application. The first role, already explained in Section 3.3, is achieved by defining how to instantiate the concrete *Feedback Loop*, *Adaptation Context*, and *Self-Adaptable Visitor* in the specialization.

For the second role, Listing 6 provides an example wrapper to create a standalone executable for a language. The first step is to create a language instance, this will set up the *Adaptation Context*, then the *Feedback Loop*, and create an instance of the visitor. A *Self-Adaptable Language* exposes two

components, the *Adaptation Context* to transfer the module registration to the *Module Registry* (l.5) and the visitor to start the execution (l.7).

```
1 SelfAdaptiveLang l = new SelfAdaptiveLang();
2 String source = readFile(filePath);
3 Parser parser = new Parser(source);
4 RootNode root = parser.parse();
5 l.getAdaptationContext().registerModule(new Module());
6 Object out = root.accept(
7     l.getVisitor(),
8     new ExecutionContext());
```

Listing 6. Example wrapper to create a standalone executable language

5 Evaluation

In this section, we present the evaluation of our approach. Section 5.1 details the studied languages, their manually implemented Self-Adaptable Virtual Machines (SAVMs), and SEALS-based counterparts. Then, we present our experimental setup and research questions in Section 5.2. Finally, the results are detailed in Section 5.3 and discussed in Section 5.4.

5.1 Studied Self-Adaptable Virtual Machines

This section presents the three studied languages : *HTML*, *RobLANG*, and *MiniJava*.

HTML allows to describe the structure of a web page and its content.

RobLANG is a domain-specific language (DSL) to specify the actions of a robot (*e.g.,* use sensors, movement).

MiniJava is an imperative and object-oriented language, subset of Java.

In addition, we present for each language: the trade-off, the environment model, the adaptations, and the implementations of the two Self-Adaptable Virtual Machines (SAVMs), i.e., using the framework or not. For some of the SAVMs we rely on other tools (e.g. Truffle, EMF, Xtext). However, the SEALS framework can be used without them and therefore should be seen as a contribution in itself. The pairs of Self-Adaptable Virtual Machines (Handcrafted/SEALS-based) were built to achieve the same trade-off analysis. However, when the feedback loop is called periodically, the time at which a decision is made can differ due to delay induced by the framework.

These three examples describe a broad range of application domains for *Self-Adaptable Virtual Machines* and are complementary in terms of language characteristics (from general-purpose to DSL, imperative to declarative) as well as the discussed feedback loop. A summary of this information is presented in Table 1.

Table 1. Summary of the languages, their type and trade-offs

VMs	Type of Language	Trade-off
HTML	Declarative DSL	Quality / Transfer size
RobLANG	Imperative DSL	Time / Energy consumption
MiniJava	Imperative GPL	Accuracy / Execution time

HTML. is a representative of declarative domain-specific languages. As presented in Section 2.3, the trade-off for this language is between the quality of the page rendering and the size of data transferred, which is proportional to the energy consumption (see Section 2.3). Moreover, the environment model is empty, i.e., we do not need to monitor the environment. We consider three modules for this language: conditional loading, HTML lists perforation, and image degradation, also detailed in Section 2.3.

HTML engines being complex hand-crafted pieces of software, we did not modify an existing engine nor did we create our own. Instead, we built the SAVMs as Pretty-Printers that change the HTML code received by the browser.

The handcrafted HTML pretty printer is built using the Truffle DSL [19], benefiting from its instrumentation framework [41] to provide the modularity required to call the adaptation modules (*i.e.*, using, among others, the framework's module registration and the dynamic injection capabilities in the interpreter of the VM). The feedback loop is implemented as a Truffle instrument (Truffle instrumentation clients who can observe and inject behavior into interpreters written using the Truffle DSL) and is called automatically by the instrumentation framework before each processed page. In addition, the adaptation context contains the trade-off model (between quality and consumption), but not the environment model which is unused in this case.

On the other hand, SEALS offers the same capabilities of module registration and frees the language engineer from the dynamic injection of module calls. Hence, the pretty-printer implemented using the framework does not need to rely on the Truffle DSL anymore. The adaptation process, *i.e.*, the trade-off analysis and the MAPE-K loop logic, remains the same but is now built with the framework constructs.

RobLANG. is also a representative of domain-specific languages, where the trade-offs are often more specialized and tailored to the application domain supported by the DSL. With RobLANG being a language for robotics, we choose to do a trade-off between time and energy spent to perform actions. For this trade-off, we choose a speed regulation adaptation. This adaptation is interesting because the consumption of a motor is deeply impacted by its speed: $Power_i = Power_{max}(\frac{Speed_i}{Speed_{max}})^3$ [1].

The handcrafted RobLANG SAVM was implemented on top of the abstract syntax generated from an Xtext [7] project. The adaptation context contains the list of modules, the trade-off model between time and energy, and the environment model containing the battery level. The feedback loop is triggered when a certain period of time has elapsed, to periodically monitor and adapt to the current battery level. At each iteration, the battery level is updated in the environment model. Then, we use the arithmetic semantics of goal models [13] to convert the goal model to a constraint model to find the relevant speed percentage to use. In our

implementation, we use the Choco Solver [22] to solve this problem. Using the solution of this problem, we configure the adaptation modules in the adaptation context.

Similarly, the SAVM implemented with SEALS uses the parser and abstract syntax generated by Xtext. The generated abstract syntax was modified to use the *Node*/*AdaptableNode* constructs. The adaptation process is converted to use SEALS' constructs, but keeps using the conversion to a constraint model for the Choco Solver as it is also part of SEALS.

MiniJava. represents the class of general-purpose languages, where trade-offs often revolve around the execution or analysis performance of the language versus the quality of the output taking the availability of computing resources into account. This trade-off is addressed by approximate computing techniques [33, 46]. Hence, we propose to apply the approximate loop unrolling technique [37] with a perforation rate depending on the CPU load of the computer.

The handcrafted MiniJava interpreter was designed on top of EMF [39] technologies. We created an adaptation context with a set of modules, a trade-off model with two properties of interest (accuracy and time), and the environment model containing the CPU load. The implemented feedback loop is triggered periodically to regularly monitor the CPU load.

As well as RobLANG, the framework-based SAVM implementation leverages on the EMF code generated for the handcrafted version and has been similarly modified. However, unlike the previous examples, the framework-based implementation of the adaptation process was slightly changed compared to the handcrafted implementation to reduce the number of expensive resolutions of the decision model.

5.2 Experimental Setup

The goal of the SEALS framework is to provide first class constructs for the language engineer to support the definition of domain-specific feedback loops in the definition of the DSL operational semantics. To validate the relevance of the abstraction provided, we propose to discuss:

1. The framework's applicability to the three examples
2. The assurances provided by the correctness envelopes
3. The reduction of the development costs
4. The performance overhead

To support the discussion on the reduction of the development costs, we propose to compare the code size, in terms of lines of code, of framework-based SAVMs to their equivalent handcrafted versions. While performance is not the goal of the SEALS framework, we propose to also compare the performance of SAVMs implemented using SEALS to their manually implemented counterparts. This first estimation of the overhead of the framework will provide insight on the usability of the SAVMs created. However, the optimization of this framework is left as future work.

To compute correctly the number of lines of code, all the sources use the same coding conventions. The performance measurements are run on a computer with 31Gb of RAM and an Intel(R) Core(TM) i7-10850H CPU (12 cores at 2.70GHz) with Manjaro 21.1.0. The SAVMs are run alone on the computer, and using GraalVM CE version 20.3.2 as JVM due to its necessity for the handcrafted HTML pretty-printer. For each SAVM, we measure 30 program runs in a row repeated three times with reboot between each repetitions.

5.3 Results

Based on our experimentation, we are evaluating 1) the applicability of the approach on the aforementioned examples, 2) the assurances provided by the correctness envelopes, 3) the benefits of our approach in terms of reduction of the development costs, and 4) the performance overhead due to the use of the framework.

5.3.1 Applicability of the approach. When evaluating the applicability of the approach, we look at the use of the constructs of the language. In particular, are all the constructs used and has the language engineer fall back to handcrafted solutions for uncovered parts of the development? When implementing the Self-Adaptable Virtual Machines using the framework, the two mandatory components to define are the feedback loop and the operational semantics. For the first, we provide 1st-class entities to define the feedback loop in the form of a MAPE-K Loop with the associated *Adaptation Context* that acts as the **K**nowledge of the MAPE-K Loop. On the other hand, SEALS supports the definition of operational semantics through the *Operation* and *AdaptableOperation* concepts. These concepts require the language engineer to use the *Node* and *AdaptableNode* constructs to define the abstract syntax of the languages, and consequently, the *SemanticsAdaptationInterface* construct for *AdaptableNode*. With the *SelfAdaptiveVisitor* generated from the operation and *Self-Adaptable Language* required by the framework, we use all the constructs provided to define the language. In addition, this definition of the language implies the use of the *Module Registry* construct and the *Adaptation Module* construct to implement the adaptations. The SAVMs studied thus use all the concepts provided by the framework.

Moreover, in our implementations, we relied on the decision model concepts provided to implement the decision process, hence, we did not have to manually implement the complex analysis of the model. More generally, the parts of the implemented SAVMs that were manually implemented are out of the scope of the framework (e.g., parsers). In other words, we did not need other constructs to implement the self-adaptable behavior with SEALS.

5.3.2 Correctness envelopes assurances. To ensure the validity of the possible adapted semantics, the framework provides the concept of correctness envelope in the form of an arbitrarily complex Java type definition. The definition of *AdaptableOperation* according to a proposed configuration of these semantics (i.e., an instance of the correctness enve-

lope) ensures that the semantics changes are bounded by the set of possible instances of the correctness envelope. On the other hand, the definition of the correctness envelope as a Java type definition protects the language semantics from invalid configuration by blocking all adaptations by default. The language engineer has to define subjects of adaptation (i.e., attributes) and operators to manipulate them (i.e., methods) for the type representing the correctness envelope, thus explicitly allowing some adaptation to be made.

5.3.3 Impact on development costs. To evaluate the impact of the framework on the development costs of the SAVMs, we measure the number of lines of code needed to define the language and the adaptation process. When comparing code size of Handcrafted versus SEALS-based implementation, we compare similar implementations, *i.e.*, the RobLANG implementation is based on Xtext in both cases but uses the SEALS framework conjointly in the second implementation. The same applies to the EMF-based MiniJava. On the other hand for HTML, we do not use Truffle and SEALS at the same time because the structure of SEALS is not fully compatible with the Truffle implementation. Those measurements are summarized in Table 2

Table 2. Summary of the number of lines of code for handcrafted and framework-based SAVMs, and size factor for each language.

Language	Handcrafted	Framework	Size factor
HTML	1969	543	x0.276
RobLANG	2069	1682	x0.813
MiniJava	13096	13672	x1.044

For HTML we observe a relatively high reduction in size. However, most of this reduction is due to the verbosity of the tag system of the Truffle instrumentation framework [41] (~700 lines of code), and Truffle DSL specific code. In the case of RobLANG, the reduction is very representative of the impact of the use of the framework because almost all the code in the framework-based implementation is identical to the handcrafted version. This reduction is mainly due to the abstraction of the components interaction and decision model implementation. Finally, MiniJava sees its size grow by ~600 lines. This augmentation is mainly due to the change in the operational semantics definition. The handcrafted version of MiniJava operational semantics is defined using the interpreter design patterns, hence, is defined as functions whereas SEALS requires to create a new class for each concept semantics. The overhead introduced by the classes' definition represent ~900 lines. In the end, it means that SEALS reduces the number of lines of code to implement the adaptation process by ~300 lines which is in line with what we observe for RobLang.

5.3.4 Introduced performance overhead. This performance study aims only at observing the overhead related to the use of the framework. We measure for each SAVM the time spent to run a program with adaptations registered but not applied. We do not apply the adaptations to measure the overhead introduced by the framework, *i.e.*, feedback loop abstractions and general architecture of the framework, to avoid being influenced by the difference in modules implementations. Table 3 summarize the average time measured and the time factor between handcrafted and framework-based versions ($Handcrafted * Timefactor = Framework$).

Table 3. Performance of the handcrafted and framework-based SAVMs and associated time factor.

Language	Handcrafted	Framework	Time factor
HTML	0.944 ± 0.010	0.685 ± 0.006	x0.726
RobLANG	1.095 ± 0.005	1.044 ± 0.007	x0.953
MiniJava	17.115 ± 0.521	19.829 ± 0.519	x1.159

These results are very different from one another and cannot be used to describe a general trend. However, even for MiniJava which introduces the highest overhead of the three experiments, its 16% overhead seems to be acceptable for the framework. Furthermore, the results for HTML provide anecdotal evidence for significant performance gains when compared to approaches that rely on general purpose capabilities. Yet, the results for MiniJava emphasize the limits of our framework abstractions compared to handcrafted adaptation processing. These results point to future work on performance to determine the reasons for the introduced overhead and find optimizations.

5.4 Discussion

Based on the applicability to the three examples studied and the gain in development costs granted by the abstractions, we conclude that our framework provides useful constructs to support the development of Self-Adaptive Virtual Machines. However, as for any framework, the gain in development costs needs to be balanced with the time necessary to understand and master the framework, which needs to be investigated in future work. In addition, the correctness envelopes provides control over the possible adaptations that can be performed on the language semantics to the language engineer. This mechanism allows a safe delegation of the adaptations' implementation to language users not involved in the language development. Finally, the results show a reasonable overhead for the framework, but further investigations into the performance of SEALS and possible optimizations are definitely needed.

6 Related Work

Abstracting non-functional features as first level entities of a programming language or a domain specific language is a common source of evolution. Thus Modula [45], followed by the first object-oriented languages [10], allowed to think about the notion of modular programming. ArchJava [2],

twenty years later, showed the interest to put forward architectural features as an entity of programming languages. For another non-functional concern, taking variability into account has introduced particular abstract concepts within programming languages [31]. The Erlang language [5] finds its specificity among others in having introduced at the heart of its semantics the problem of concurrency and fault tolerance. If we take example from general purpose programming languages, it is common to see the same trend in DSLs. Our work is clearly part of this trend, aiming at introducing a non-functional concern (i.e., self-adaptation) at the heart of the semantics of new programming languages in providing a framework for building self-adaptive virtual machines.

The techniques used to build self-adaptable software systems have been widely studied for more than 20 years. Several surveys [23, 28] summarize the main techniques that have been designed. Then, the definition of a feedback loop has been investigated in the context of Self-Adaptive Systems [8], with a field mature enough to provide time-honored patterns such as the MAPE-K loop [24]. Within the taxonomy defined by Krupitzer *et al.* [28] to classify self-adaptive techniques, the authors propose five main criteria to classify self-adaptation approaches (*Reason*, *Level*, *Time*, *Technique*, *Adaptation control*). For the *Technique* part, the authors introduce three sub-categories: *Parameter*, *Structure*, and *Context*. *Parameter* refers to adaptation through the change of parameters. *Structure* subsumes change in the structure of the technical system, *e.g.*, new composition of components, removal/addition of components. Further, changes in the relation between elements, technical resources, or the environment/user(s) are also structural adaptations.*Context* refers to any changes in the context, *e.g.*, modifying the state of context variables via actuators. The main approaches interested in adaptation techniques used at the implementation level are: Architecture-based, Reflection, Programming Paradigms (Component-Based Software Engineering [27, 49], Generative Programming [11, 14], Context-Oriented Programming [18], and Aspect Oriented Programming [25, 47]).

Introducing self adaptation as a first class entity in a new programming language could be seen in a way quite close to Aspect Oriented Programming (AOP). We share the vision of removing the concern of self-adaptation for the language user when she is defining her program. However while the use of AOP is a good way to abstract the concern of adaptation [47], it has limitations. First, the approach relies on a pointcut definition which is focused on method call only. In this context, adaptation of language domain-specific primitives is impossible. Second, because it relies on function calls, the adaptation can only be written by the domain expert (language user) that knows which ones to target. Finally, the domain expert will have to deal manually with the feature interaction of the different aspects used. In proposing a framework allowing within a virtual machine (VM) to specialize the semantics of a language feature in order to adapt to the execution context, we could easily change the normal behavior of a program running on top of our specialized VM. Thus, a language engineer can specify the semantics of her language but also the validity envelope of the language. This allows either another language engineer or a domain expert to specialize the semantics of the language, while ensuring by construction that the semantics remain within a correctness envelope defined by the original language designer. Besides, this semantics variation becomes available for all the programs running on top of this VM.

7 Conclusion

The *SElf-Adaptive LanguageS Framework* (SEALS) aims to support the building of self-adaptive virtual machines for domain-specific languages. This framework provides first-class entities for the language engineer to promote domain-specific feedback loops in the definition of the DSL operational semantics. In particular, the framework supports the definition of (i) the abstract syntax and the semantics of the language as well as the correctness envelope defining the acceptable semantics for a domain concept, (ii) the feedback loop and associated trade-off reasoning, and (iii) the adaptations and the predictive model of their impact on the trade-off. In building three languages with self-adaptive virtual machines, we highlight the relevance of the abstractions and the effectiveness of correctness envelopes. The implementation of the adaptive behavior in these three languages used all constructs provided by the framework and did not need additional constructs. We also discuss the cost in terms of lines of code and provide a first comparison of the performance results with their manually implemented counterparts.

This initial performance study aimed only at observing the overhead related to the use of the framework. However, the HTML use case led us to discuss the potential benefits of using the framework for an implementation on top of Truffle/GraalVM rather than relying on its instrumentation framework. These first experiments pave the path to investigate performance benefits for integrating the autonomic loop at the level of the operational semantics of DSLs. Furthermore, future work has to be done at the level of concrete syntax or language tooling (editor) in order to make the language user aware of the parts of the program impacted by adaptable semantics. Finally, the problem of feature interactions [4] between adaptation modules remains. For the moment, the application order of the modules impacting the same AST node must be specified by the language designer.

References

[1] 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).*

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: connecting software architecture to implementation. In *ICSE '02: Pro-*

ceedings of the 24th International Conference on Software Engineering (Orlando, Florida). ACM, New York, NY, USA, 187–197.

[3] Anders SG Andrae and Tomas Edler. 2015. On global electricity usage of communication technology: trends to 2030. *Challenges* 6, 1 (2015), 117–157.

[4] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of Feature Interactions Using Feature-Aware Verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, USA, 372–375.

[5] Joe Armstrong. 2010. erlang. *Commun. ACM* 53, 9 (2010), 68–75.

[6] Nelly Bencomo, Robert B France, Betty H C Cheng, and Uwe Aßmann. 2014. *Models@run.time: foundations, applications, and roadmaps*. Vol. 8378. Springer.

[7] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[8] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–26.

[9] B. Combemale, J. A. Kienzle, G. Mussbacher, H. Ali, D. Amyot, M. Bagherzadeh, E. Batot, N. Bencomo, B. Benni, J. Bruel, J. Cabot, B. H. C. Cheng, P. Collet, G. Engels, R. Heinrich, J. Jezequel, A. Koziolek, S. Mosser, R. Reussner, H. Sahraoui, R. Saini, J. Sallou, S. Stinckwich, E. Syriani, and M. Wimmer. 2020. A Hitchhiker's Guide to Model-Driven Engineering for Data-Centric Systems. *IEEE Software* (2020), 0–0.

[10] Brad J. Cox and Andrew Novobilski. 1991. *Object-Oriented Programming; An Evolutionary Approach* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[11] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., USA.

[12] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. 2006. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 1, 2 (2006), 223–259.

[13] Yuxuan Fan, Amal Ahmed Anda, and Daniel Amyot. 2018. An arithmetic semantics for GRL goal models with function generation. In *International Conference on System Analysis and Modeling*. Springer, 144–162.

[14] Scott D Fleming, Betty HC Cheng, RE Kurt Stirewalt, and Philip K McKinley. 2005. An approach to implementing dynamic adaptation in c++. In *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*. 1–7.

[15] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Educ. India.

[16] George Kamiya. 2020. Data Centres and Data Transmission Networks. https://www.iea.org/reports/data-centres-and-data-transmission-networks.

[17] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice*. Addison-Wesley Professional.

[18] Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. 2008. Context-oriented programming. *Journal of Object technology* 7, 3 (2008), 125–151.

[19] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2015. A domain-specific language for building self-optimizing AST interpreters. *ACM SIGPLAN Notices* 50, 3 (2015).

[20] Jean-Marc Jézéquel, Benoît Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. 2015. Mashup of metalanguages and its implementation in the Kermeta language workbench. *Software and System Modeling* 14, 2 (2015), 905–920.

[21] Gwendal Jouneaux, Olivier Barais, Benoit Combemale, and Gunter Mussbacher. 2021. Towards Self-Adaptable Languages. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*.

[22] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. 2008. Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*. 1–10.

[23] Stephen Kell. 2008. A Survey of Practical Software Adaptation Techniques. *Journal of Universal Computer Science* 14, 13 (2008), 2110–2157.

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

[25] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European conference on object-oriented programming*. Springer, 220–242.

[26] Jörg Kienzle, Gunter Mussbacher, Benoit Combemale, Lucy Bastin, Nelly Bencomo, Jean-Michel Bruel, Christoph Becker, Stefanie Betz, Ruzanna Chitchyan, Betty H. C. Cheng, Sonja Klingert, Richard F. Paige, Birgit Penzenstadler, Norbert Seyff, Eugene Syriani, and Colin C. Venters. 2020. Toward Model-Driven Sustainability Evaluation. *Commun. ACM* 63, 3 (Feb. 2020), 80–91.

[27] Wojtek Kozaczynski and Grady Booch. 1998. Component-based software engineering. *IEEE software* 15, 5 (1998), 34.

[28] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. 2015. A Survey on Engineering Approaches for Self-Adaptive Systems. *Pervasive Mob. Comput.* 17, PB (Feb. 2015), 184–206.

[29] Manuel Leduc, Thomas Degueule, Benoit Combemale, Tijs Van Der Storm, and Olivier Barais. 2017. Revisiting visitors for modular extension of executable DSMLs. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 112–122.

[30] Manuel Leduc, Thomas Degueule, Eric Van Wyk, and Benoit Combemale. 2019. The Software Language Extension Problem. *Softw. Syst. Model.* (Dec 2019), 1–5.

[31] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 105–114.

[32] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause. 2016. An Empirical Study of Practitioners' Perspectives on Green Software Engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 237–248.

[33] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4, Article 62 (March 2016), 33 pages.

[34] Peter D Mosses. 2004. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming* 60-61 (2004), 195–228.

[35] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty H. C. Cheng, Philippe Collet, Benoit Combemale, Robert B. France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. 2014. The Relevance of Model-Driven Engineering Thirty Years from Now. In *Model-Driven Engineering Languages and Systems*, Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran (Eds.). Springer International Publishing, Cham, 183–200.

[36] Bruno C d S Oliveira and William R Cook. 2012. Extensibility for the Masses. In *ECOOP*. Springer, 2–27.

[37] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. 2019. Approximate loop unrolling. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. ACM.

[38] Stuart Russell and Peter Norvig. 2002. Artificial intelligence: a modern approach. (2002).

[39] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[40] International Telecommunication Union. 2018. Recommendation Z.151 (10/18): User Requirements Notation (URN) - Language Definition.

[41] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, flexible, polyglot instrumentation support for debuggers and other tools. *arXiv:1803.10201* (2018).

[42] Axel van Lamsweerde. 2009. *Requirements Engineering: From System Goals to UML Models to Software Specifications* (1 ed.). Wiley.

[43] Wholegrain Digital. 2020. Website Carbon. https://www.websitecarbon.com/.

[44] Anthony Williams. 2019. *C++ Concurrency in Action*. Manning.

[45] Niklaus Wirth. 1985. *Programming in MODULA-2 (3rd Corrected Ed.)*. Springer-Verlag, Berlin, Heidelberg.

[46] Q. Xu, T. Mytkowicz, and N. S. Kim. 2016. Approximate Computing: A Survey. *IEEE Design Test* 33, 1 (Feb 2016), 8–22.

[47] Zhenxiao Yang, Betty HC Cheng, RE Kurt Stirewalt, J Sowell, Seyed Masoud Sadjadi, and Philip K McKinley. 2002. An aspect-oriented approach to dynamic adaptation. In *Proceedings of the first workshop on Self-healing systems*. 85–92.

[48] Eric Yu. 1995. *Modelling Strategic Relationships for Process Reengineering*. Ph.D. Dissertation. University of Toronto.

[49] Ji Zhang, Betty HC Cheng, Zhenxiao Yang, and Philip K McKinley. 2005. Enabling safe dynamic component-based software adaptation. In *Architecting Dependable Systems III*. Springer, 194–211.