Towards Self-Adaptable Languages

Gwendal Jouneaux, Olivier Barais, Benoit Combemale, Gunter Mussbacher

Towards Self-Adaptable Languages

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 concern for many software systems that have to operate in complex and changing environments. At the core of self-adaptation, there is a feedback loop and associated trade-off reasoning to decide on the best course of action. However, existing software languages do not abstract the development and execution of such feedback loops for self-adaptable systems. Developers have to fall back to ad-hoc solutions to implement self-adaptable systems, often with wide-ranging design implications (*e.g.,* explicit MAPE-K loop). Furthermore, existing software languages do not capitalize on monitored usage data of a language and its modeling environment. This hinders the continuous and automatic evolution of a software language based on feedback loops from the modeling environment and runtime software system. To address the aforementioned issues, this paper introduces the concept of *Self-Adaptable Language* (SAL) to abstract the feedback loops at both system and language levels. We propose *L-MODA (Language, Models, and Data)* as a conceptual reference framework that characterizes the possible feedback loops abstracted into a SAL. To demonstrate SALs, we present emerging results on the abstraction of the system feedback loop into the language semantics. We report on the concept of *Self-Adaptable Virtual Machines* as an example of semantic adaptation in a language interpreter and present a roadmap for SALs.

Keywords: self-adaptation, feedback loop, trade-off analysis, software language, L-MODA framework

ACM Reference Format:
Gwendal Jouneaux, Olivier Barais, Benoit Combemale, and Gunter Mussbacher. 2021. Towards Self-Adaptable Languages. In *Onward! '21: , October 17–22, 2021, Chicago, IL.* ACM, New York, NY, USA, 16 pages. https://doi.org/???

Onward! '21, October 17–22, 2021, Chicago, IL

1 Introduction

Software systems are more and more evolving in complex environments on which they eventually are highly dependent and consequently require dynamic self-adaptation to best deliver the expected service [9]. Self-adaptation necessitates a feedback loop to react to changes in the environment and trade-off analysis to determine the best course of action given the current context [27]. While such a feedback loop is considered as the primary concern of systems like autonomous cars (*e.g.,* Waymo[1]) and large-scale video streaming platforms (*e.g.,* Netflix[2]), 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. For example, in the context of green IT [31], many systems like e-commerce applications can leverage on trade-offs related to sustainability and balance the provided service 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 [28, 34]. Developers of such systems can benefit from enhanced support for this emerging concern, *i.e.,* the implementation of feedback loops and trade-off reasoning.

The modeling (or programming) ecosystem for the development of such systems is a similarly complex system, which requires the use of various software languages for a varied set of activities (incl., all software engineering activities), performed by very different stakeholders. Software languages are evolving just like natural languages in response to emerging concepts and relationships, possibly for a particular application domain. We observe that a currently informal feedback loop exists in this ecosystem. For example, a language user who is using a domain-specific language or a software language engineer may discover new patterns and their trade-offs, which may then be reified in the language.

Language users and software language engineers alike can benefit from a more formal approach to the specification of feedback loops and trade-off analyses to adapt and

[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.

evolve a software language, and better support the implementation of complex self-adaptable software systems, and the organic evolution of the software languages at hand. Over the last decades, the software engineering community proposed an important body of knowledge about the design and implementation of self-adaptable systems [9]. 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 [27]. Trade-off analysis is supported with dedicated modeling infrastructure (*e.g.,* models@runtime [6], goal modeling [48, 50, 57]).

In the same way software languages abstracted concerns like concurrency and parallelism [19, 53] into high level language constructs for language users who do not have the expertise or do not need (want) to explicitly deal with those concerns, there is a need to do the same for self-adaption of the system to be developed as well as the languages themselves. This paper argues for the software engineering community to increasingly investigate how to incorporate self-adaptation into modern software languages. The main objective is to abstract from the feedback loop of the self-adaptive system as much as possible, so as to free the developers from the detailed specification or implementation of the feedback loop. After elaborating four motivating examples in Section 2, we introduce the concept of *Self-Adaptable Language* (SAL) in Section 3. To demonstrate the benefit of the overall vision, we explore the integration of a feedback loop in language semantics, leading to the concept of *Self-Adaptable Virtual Machines*. We discuss the design of the *Self-Adaptable Virtual Machines* in Section 4 and report on emerging results in Section 5. Section 6 present a roadmap of the main expected features covering the lifetime use of SALs. Finally, we discuss related work in Section 7 and conclude the paper in Section 8.

2 Motivating examples

In this section, we provide motivating examples for the introduction of feedback loops and trade-off reasoning at the language level. The section is structured around the three following languages: *MiniJava*, *RobLANG*, and *HTML*.

- **MiniJava** is an imperative and object-oriented language, subset of Java.
- **RobLANG** is a domain-specific language (DSL) to specify the actions of a robot (*e.g.,* use sensors, movement).
- **HTML** allows to describe the structure of a web page and its content.

First, we look at the self-adaptation of the language semantics and depict for each of the three languages the trade-off, the monitored environment, proposed adaptations, and the roles of the stakeholders (Sections 2.1-2.3). The three language examples describe a broad range of application domains for *Self-Adaptable Languages* and are complementary in terms of language characteristics (from general-purpose to DSL, imperative to declarative) as well as the discussed feedback loop. For each language, a different stakeholder decides on the desired constraints for trade-offs in the feedback loop. A summary of this information is presented in Table 1. Second, we look at the self-adaptation of the language concepts through the adaptation of its abstract syntax (Section 2.4). We take the RobLANG DSL as an example of languages that could benefit from this adaptation to evolve according to its use by language users.

2.1 Saving computations in MiniJava

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. For the case of MiniJava, we choose a trade-off between accuracy and execution time. This trade-off is addressed by approximate computing techniques [33, 55].

To illustrate the possible adaptations for this trade-off, we propose to apply the approximate loop unrolling technique [39] with a perforation rate depending on the CPU load of the computer. This self-adaptable MiniJava will capitalize its internal feedback loop to perform the trade-off analysis considering the monitored environment (*i.e.,* CPU load) and the new execution paths provided by the approximate loop unrolling adaptation (*i.e.,* different perforation rates).

In this example, the configuration of the feedback loop and adaptation are managed by the language engineer. This implies that the language engineer configures trade-off analysis, the set of possible execution paths (*i.e.,* possible perforation rates), and the relaxation of constraints, like accuracy.

2.2 Optimizing battery usage in RobLANG

RobLANG is 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.

Among the possible adaptations to best satisfy the trade-off, we choose a speed regulation adaptation. Robots are used to perform tasks and most of their consumption is caused by the actuators (*e.g.* motors). The goal of this adaptation is to reduce the speed of the robot motors when moving or turning to save energy. This adaptation is interesting because the consumption of a motor is deeply impacted by its speed : $P_i = P_{max}(\frac{Speed_i}{Speed_{max}})^3$ [1]. Hence, we should be able to save a lot of energy without impacting too much the robot speed.

For this example, the language engineer will configure the feedback loop and trade-off analysis, but will let the language user configure the set of possible execution paths by giving more or less freedom to the adaptations.

VMs	Type of Language	Trade-off	Configuring stakeholder
MiniJava	Imperative GPL	Accuracy / Execution time	Language engineer
RobLang	Imperative DSL	Time / Energy consumption	Language user
HTML	Declarative DSL	Rendering quality / Transfer size	End user

Table 1. Summary of the languages with its type, trade-offs, and stakeholder in charge of configuring the adaptations.

2.3 Reducing data transfer in 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 [2]. Comparatively, the web electricity demand was representing around 2% of 2015 world electricity demand [18].

For this reason, we choose a trade-off between energy consumed to display the web page and the quality of the page rendering. The energy consumed by a website is difficult to assess due to the networking part. However, Website Carbon Calculator [52] provides an algorithm that estimates this consumption, and this consumption is proportional to the size of the transferred data.

We propose to study three adaptations, 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. The last one is the degradation of the image to reduce its size. The idea is to request a degraded version of the image if its size exceeds a certain threshold.

In this case, we empowered the end user with the definition of the expected trade-off due to the subjective character of the rendering quality. The language user does not configure anything, allowing compatibility with any HTML code. On the other hand, the language engineer remains in charge of the trade-off analysis configuration.

2.4 RobLANG evolution through self-adaptation

Software languages evolve to keep up with the demands of the changing needs of their stakeholders. This applies to general purpose languages as well as to domain specific languages. Consider, e.g., the evolution of the for loop in the Java general-purpose programming language, from counter-based or iterator-based syntax to the more streamlined syntax of the for-each concept. Similarly, DSLs are tools for humans to better understand and manipulate domain-specific concepts. In this context, they also tend to evolve over time to provide better abstractions because of, e.g., a better understanding of the application domain.

In the case of RobLANG, we manipulate domain-specific concepts like "move forward", "turn left", or "sense position" that can be used to perform any movement. However, one of the first functions implemented by a majority of language users may be the `goTo(x,y)` function. Upon monitoring the modeling environment of these language users and detecting the repeating implementation of the `goTo(x,y)` function, an adaptation proposed by a self-adaptable abstract syntax would be the reification of the instructions that compose the `goTo` function into a new domain-specific concept called *goTo* provided directly by RobLANG. This new function could then be used for new systems, but existing systems could also be updated (semi-)automatically to take advantage of the new language concept. The co-evolution of existing models/programs with the adapted language ensures that the language users are aware of the change by highlighting applicable portions of the implementation that can be adapted and what the adaptation entails.

3 Introducing Self-Adaptable Language

A *Self-Adaptable Language* (SAL) is a software language which provides capabilities to abstract the design and execution of feedback loops while performing trade-off analyses. First, such a language frees language users from the explicit (typically from scratch) implementation of the feedback loop and associated reasoning, which often requires the application architecture to be primarily based on the overpowering concern of self-adaptation. Hence, a SAL offers the ability to primarily focus on the expected domain-specific services to be delivered by the software system, and possibly customize the underlying feedback loop to control the trade-off analysis performed at the language level. Second, such a language allows for continuous and automatic evolution of itself based on feedback from the modeling environment[4]. In essence, such a self-adaptable system, its users and their environments, the software modeling environment including the *self-adaptable language*, and the language engineering environment form a *socio-technical modeling system*, *i.e.,* one interconnected, highly dynamic, heterogeneous, and adaptive system with several feedback loops and trade-off analyses.

As envisioned, the activities of the language engineer have a great and profound impact on the system. From the language engineer's point of view, the concept of SAL introduces this new concern that brings the runtime consideration

[4]We refer to the environment used by a language user to build systems generally as modeling environment and include the development environment including tool support in that definition.

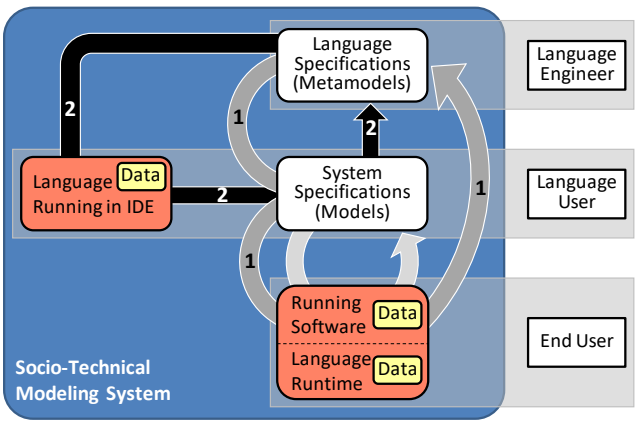


Figure 1. L-MODA Reference Framework for Self-Adaptable Languages

of application-specific context and the design-time consideration of development-specific context into the realm of language definition. At the heart of a specification of a *Self-Adaptable Language* (SAL) are *feedback loops* that encompass the application domain including the *runtime software system*, the *modeling environment* used to build the application, and the *language engineering environment* used to provide the modeling environment. A key criteria of these feedback loops is that they abstract the self-adaptation that may occur during the design or execution of a software system to adapt the *language syntax, semantics, or pragmatics.*

To discuss the fundamental concepts of SALs we propose L-MODA (*Languages, Models, and Data*) as a conceptual reference framework that characterizes the dependencies of *(i)* the software system at runtime including its data, *(ii)* the modeling environment with its data, models, and the self-adaptable language, and *(iii)* the language definition environment.

Figure 1 introduces the proposed L-MODA conceptual reference framework with its two distinct feedback loops. The figure depicts the running software/language in orange, its data in yellow, and system and language specifications in white rounded rectangles. Furthermore, the roles of the three major stakeholders are highlighted with gray shaded background and white rectangles: *language engineer*, *language user*, and *end user*.

The first feedback loop involves the running software system as well as its language and system specifications, feeding system data from the execution environment to the language engineering environment. This feedback loop will be referred as the *runtime feedback loop* (cf. 1 in Fig. 1). Examples of the runtime feedback loop are the MiniJava, RobLANG, and HTML adaptations discussed in Section 2.1, 2.2, and 2.3. The second feedback loop involves the use of the language in its environment (*Language Running in IDE*), feeding data and models from the modeling environment to the language engineering environment. This feedback loop will be referred as the *design feedback loop* (cf. 2 in Fig. 1).

An example of the design feedback loop is the evolution of the RobLANG language discussed in Section 2.4. System data is any data from the runtime use of the software system (*e.g.,* performance measures of the application - CPU load in the MiniJava example), whereas modeling data is any data from the use of the language (*e.g.,* number of times a construct is used or an error occurs - use of `goTo(x,y)` function in RobLANG). The outcome of these two feedback loops are changes to the *language syntax, semantics, or pragmatics.*

The L-MODA framework supports various uses of its feedback loops. First, a language engineer may be in complete control of the feedback loop, i.e., a language user is not involved as is the case for the MiniJava example. Based on system data (e.g., CPU load) or modeling data and models, the language is adapted. Since the language user is not involved, this typically involves the adaptation of the language's execution semantics using the *runtime feedback loop*. This may either involve choosing the best adaptation option from a set of existing ones or the discovery of a new one to be considered for the language.

Second, a language engineer may provide the language user the ability to customize what/how something is adapted or monitored. The language is then adapted accordingly, taking system data or modeling data and models into account. Since the language user is involved, this may involve the adaptation of language syntax and pragmatics through the *design feedback loop* in addition to language semantics by the *runtime feedback loop*. For example, *(i)* new language patterns may be detected through monitoring, the patterns then reified as new adaptation options including their trade-offs, hence changing the language syntax; or *(ii)* new guidelines for the use of the language may be developed based on observations of projects, leading to updated language pragmatics or more intelligent modeling assistants [35]. This is the case in the RobLANG examples, where semantics (see Section 2.2) and abstract syntax (see Section 2.4) are adapted.

Third, the language user may defer some customizations of the *runtime feedback loop* to the end user. For example, an end user may indicate preferences for trade-offs such as energy over quality, which are then taken into account during the adaptation, as is the case in the HTML example.

4 Design of Self-Adaptable VMs

This section reports on the incorporation of the runtime feedback loop in languages operational semantics. In particular, we explore the concept of *Self-Adaptable Virtual Machines* (VMs) as language interpreters with adaptive operational semantics that free language users of domain-specific dynamic adaptations during software development. The adaptation of the abstract syntax or pragmatics of a language is left for future work (see roadmap in Section 6).

In our approach, we implement *Self-Adaptable VMs* through a pluggable architecture with a generic MAPE-K loop orchestrator that manages the different plugins (adaptations)

allowing a wide audience to build adaptations for the virtual machine. Thus, the creation of a *Self-Adaptable VM* requires, in addition to the virtual machine itself, the development of three additional components: the concrete functions of the feedback loop, the adaptation context, and adaptation rules. The feedback loop functions contain the logic to update the runtime models, perform trade-off analysis, make a decision, and extract the result during the ***M***onitor, ***A***nalyse, ***P***lan, and ***E***xecute functions of the MAPE-K loop.

Adaptation rules are built in the form of modules that expose *(i) a predictive model of impacts*, *(ii) the part of the semantics affected*, and *(iii) constraints that can be relaxed*. A module exposes a predictive model of its impacts on the properties of interest allowing the feedback loop to evaluate the relevance of its application in the current context. The part of the affected semantics is exposed to let the virtual machine know when to call the module. Finally, the constraints that can be relaxed are exposed to allow non-designers of the module to configure it (*e.g.,* RobLANG in Section 2.2). The part of the affected semantics is dynamically verified by calling a method of the modules that verifies the need to call the adaptation for this node semantics. This method allows verification of structural properties on the underlying abstract syntax tree (AST) as well as non-structural like time related properties. An exposed constraint represents what will be degraded by the adaptation (*e.g.,* accuracy, time, quality) in favour of an important property of interest according to the trade-off. Hence, the relaxation of this constraint to a certain extent allows the decision process to compute different execution paths for the concerned part of the semantics and choose the best path according to the expected trade-off. For instance, the speed regulation module of RobLANG exposes its trade-off between time and energy, affects the nodes of the AST that make use of the motors (*e.g.,* turn and move) , and the time constraint that can be relaxed to let the module reduce the speed. The adaptation context acts as the common ***K***nowledge of the MAPE-K loop [12] and contains the properties of interest, the monitored environment, and the registry of adaptation modules.

We identify three stakeholders in the implementation of *Self-Adaptable Virtual Machines*: the language engineer, the adaptation developer, and the language user. The language engineer is in charge of the overall MAPE-K loop implementation, including the architecture for the trade-off analysis and the definition of the *adaptation context*. The adaptation developer uses this common knowledge to define the adaptation modules to be included in the MAPE-K loop. We differentiate the role of adaptation developer, because language engineers, language users or any developer could also plays the role of the adaptation developer. Finally, the language user configures the MAPE-K loop by setting the trade-offs among the properties of interest (see Section 2.2), if allowed by the language engineer (see Section 2.1). The language engineer or language user may also delay the setting of trade-offs to the end user (see Section 2.3).

4.1 Studied *Self-Adaptable Virtual Machines* (VMs)

We built three VMs, one for each of the previously mentioned examples (Section 2.1, 2.2, and 2.3).

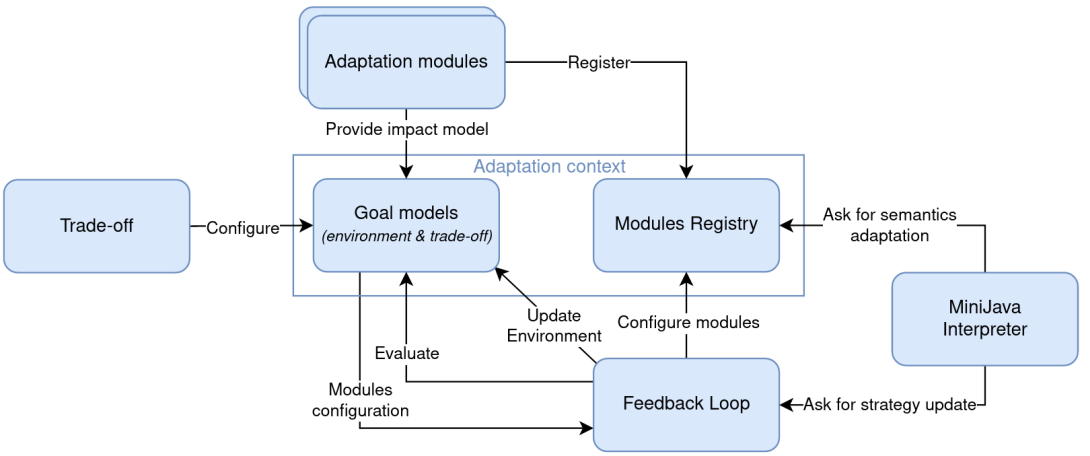
Among the multiple trade-off analysis techniques available, we choose to use goal models for the feedback loop of all these VMs. These goal models are defined using GRL [48] to model the trade-off, as well as the contribution of the adaptations to the satisfaction of the underlying properties of interests. However, any trade-off analysis process could replace the goal models.

Three types of goal models are involved: a trade-off model, an environment model, and one or more impact models. The first two are part of the adaptation context and are designed by the language engineer, while the last one is defined within the modules by the adaptation developer. The trade-off model defines the properties of interest and the expected trade-off between them. This trade-off can be delegated to the language user (RobLANG) or to the end user (HTML). The environment model defines the monitored resources that will be updated during the ***M***onitoring phase. Finally, the impact model is the predictive model exposed by the module. This model is connected at runtime to the previous ones, 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. At runtime, ***M***onitoring consists of observing the system and updating the environment model, ***A***nalysis and ***P***lanning consists of solving the global model (or the derived constraint model) and choosing the adaptation rules to be applied, and ***E***xecution activates or deactivates the modules. In addition, when the expected trade-off is delegated (RobLANG and HTML) and subject to change at run-time, the ***M***onitoring phase must also update the trade-off in the relevant model. Resolving the goal model is done in two phase. First, we evaluate the values of the adaptation parameters by looking at their impact on the global trade-off and set to the predetermined value for positive or negative impact. Second, we compute the impact of the module on the global trade-off according to the monitored environment and activate/deactivate the modules with a positive/negative impact.

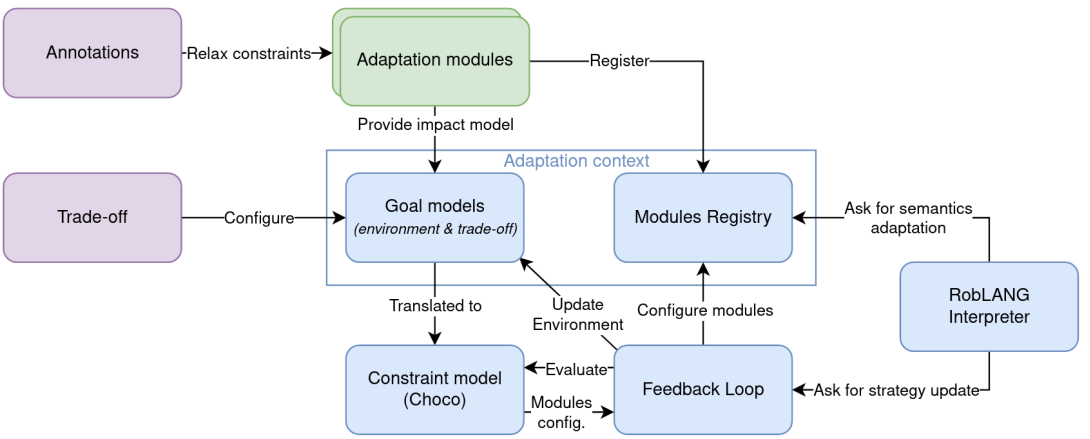
Each of these VMs covers different possible uses of *Self-Adaptable Virtual Machines*, including: different types of languages, different trade-offs, and different levels of configuration for adaptations. The VMs, languages, trade-offs, and stakeholder in charge of the configuration are summarised in Table 1 and will be detailed for each virtual machines later.

MiniJava. MiniJava is a subset of Java, thus it is a general purpose imperative and object oriented language (GPL). For a general purpose language, the domain concepts are the computations themselves. Often, the developer wants his program to be computed as quickly as possible.

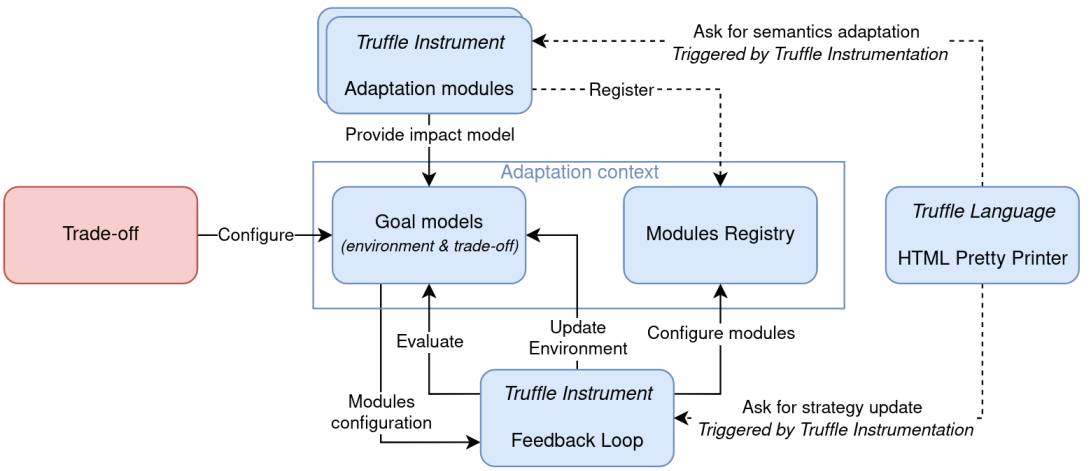
To address this concern, we choose to study the well known trade-off between execution time and accuracy. This trade-off can be addressed, among others, by using approximate



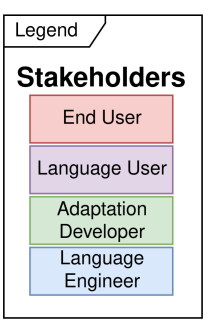
(a) MiniJava VM



(b) RobLANG VM



(c) HTML VM



(d) Legend

Figure 2. Overview of the VMs core components, their interactions, and the stakeholders involved in their design

computing techniques [33, 55]. For this language, we have chosen to let the language engineer (represented in blue in Figure 2a) configure the whole VM, and therefore also create the adaptation modules. In this case, all the execution paths can be set at design time and the best path is chosen at runtime according to the current environment.

Figure 2a presents the core components involved in our implementation of the MiniJava VM, their interactions, and the stakeholders involved in their design. The MiniJava interpreter was designed on top of EMF [45] technologies. We created an adaptation context with the module registry, a trade-off model with two properties of interest (accuracy and time) and the environment model containing the CPU load. We implemented the feedback loop that is triggered when the time spent since the last iteration is superior to a certain amount in order to regularly monitor the CPU load.

At each iteration, the feedback loop updates the CPU load in the environment model and evaluates the model. Based on this evaluation, the loop selects the best configuration for the modules and updates the state of the modules in the registry. In addition, the semantics of the AST nodes are wrapped with calls (i) to the feedback loop, to trigger it if needed, and ii) to the module registry to initiate the necessary adaptations. Finally, we build an adaptation module that applies the approximate loop unrolling technique [39] that interpolates the next n values stored in an array traversed by the loop. In our case, this n depend on the CPU load, below 25% $n = 0$, between 25% and 50% $n = 1$, between 50% and 75% $n = 3$, above 75% $n = 7$. This behavior is reflected in the predictive model (impact model) that exposes an increase in the time property satisfaction by $n/n + 1$ and a decrease in the accuracy by the same amount.

RobLANG. The RobLANG DSL is a domain-specific language for robotics created to specify behavior for robots in the Webots simulator [32]. Robots, like any autonomous battery-powered device, must perform their primary tasks while avoiding running out of energy. For this reason we choose to do a trade-off between energy consumption and time spent to perform actions, where, in this case, the time spent is tightly related to the motors speed. For this VM, we postpone the constraints relaxation to the language user (represented in purple in Figure 2b). Indeed, the actions of a robot are not of equal importance and need the expertise of the language user to best select the possible execution paths.

Figure 2b presents the core components involved in our implementation of the RobLANG VM, their interactions, and the stakeholders involved in their design. The RobLANG interpreter was implemented manually on top of the abstract syntax generated from an Xtext [7] project. The adaptation context contains the module registry, the trade-off model between time and energy, and the environment model containing the battery level. The feedback loop is triggered when the constraints imposed by the developer change or when a certain period of time has elapsed. At each iteration, the battery level is updated in the environment model. Then, we use the arithmetic semantics of goal models [16] to convert the goal model to a constraint model with the relaxation of modules' constraints specified by the language user. In our implementation, we use the Choco Solver [25] to solve this multi-constraint optimization problem. Using the solution of the optimization problem, we configure the adaptation modules in the registry. As in the MiniJava VM, the AST nodes semantics are wrapped with calls to the feedback loop and module registry. Finally, we implemented a speed control adaptation module. This module reduces the speed of the robot in order to reduce the consumption of the motors, while maintaining the speed above a percentage of the robot's nominal speed. This percentage depends on the relaxation of the time constraint specified by the language user.

HTML. Given the surging electricity demands of ICT, we decided to work on the trade-off between the quality of web browsing and its energy consumption. The energy consumption for displaying a web page being proportional to the size of data transferred [52], we worked on HTML which is the first data loaded and source of all the subsequent data transfers. For this VM, the configuration is deferred to the end user. The quality of a web page being context dependent, the trade-off is configured by the end user (represented in red in Figure 2c) of the system that can freely change the trade-off depending on his/her expectations for the web page. This means that all the constraints are relaxed by default and the configuration of the modules is chosen based only on the satisfaction value of the trade-off specified by the end user.

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 two artifacts: an extension for the end user browser and an HTTP Proxy. Figure 3 presents the interaction of these artifacts with the HTML VM and the messages going through the network. The gray box denotes the artifacts created (or partially for the browser with extension) for this approach.

The browser extension is the interface provided to the end user to configure the HTML VM. This interface contains a button to activate or not the redirection of the request to the proxy, and a slider to express the expected trade-off between quality and energy consumption. When used, the proxy receives the URL of the requested page and requests it (1 & 2 in Figure 3). Then, the received HTML code (3) is fed to the HTML VM (4) that adapts its content according to the selected trade-off (5). Finally, the proxy delivers the adapted page to the browser that will display it (6).

As we send the adapted version of the page to the browser engine, this adapted page must be in HTML format. Therefore, the HTML VM has been built as an HTML pretty printer.

As for the previous VMs, Figure 2c details the implementation of our HTML VM, and the stakeholders involved in its design. The HTML pretty printer is built using the Truffle DSL [22], benefiting from its instrumentation framework [49]

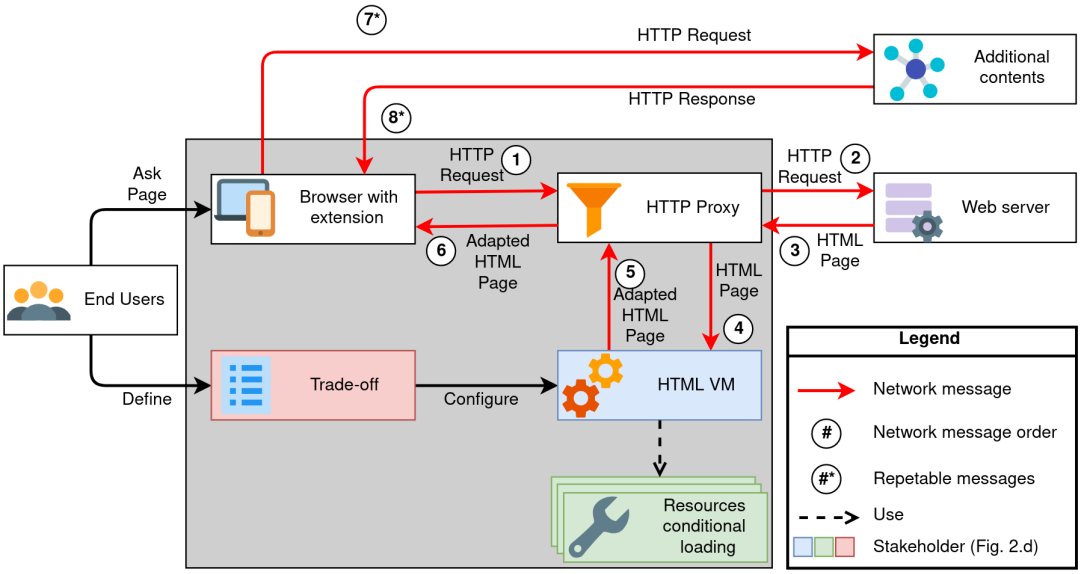


Figure 3. Overall architecture of the Self-Adaptable HTML VM

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 interactions already managed by the Truffle instrumentation API are represented using dashed arrows in Figure 2c. The registration is automatically done just after the Truffle registration to ensure the module instantiation by the instrumentation system. 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. 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 framework before each processed page. As the end user trade-off is taken into account for the whole page and only affects the result, the feedback loop is triggered once per page. For this VM, the monitoring consist of updating the trade-off model according to the end user trade-off. Then, the VM evaluates the overall goal model and selects the best configuration for the modules. Finally, the modules are activated or deactivated in the registry. Thanks to the Truffle instrumentation framework, we do not need to wrap the semantics of the nodes with appropriate calls, but only redirect the calls made to the instrument to the adaptation function.

The HTML VM works with three adaptation modules: *conditional loading*, *HTML lists perforation*, and *image degradation*. The *conditional loading* module deletes content loaded after the HTML depending on its URL. For our implementation, we delete content coming from a different domain name. This heuristic allows us to remove external content that is less likely to contain information relevant for the end user. The *HTML lists perforation* module removes list items from the page depending on the size of the list. Our implementation focuses on unordered lists where all elements of the list have the same importance. We apply loop perforation [43] to the loop in the unordered list semantics with a perforation rate inferred from the expected trade-off and the size of the list. While these modules result in a loss of information, the *image degradation* module keeps the information in a degraded version. For the latter module, we scale down the requested images to reduce their size. We rely on the structure of our approach (proxy) to manage a cache of images and degraded versions for several end users. This reduces the cost of the original image download by spreading it over several end users. In addition, this cache also gives access to the original image size which was unavailable *a priori*. The choice between the original and degraded version is made according to the original size and the end user trade-off.

5 Evaluating Self-Adaptable VMs

5.1 Experimentation

All the experimentations were run on a computer with 15Gb of RAM and an Intel(R) Xeon(R) W-2104 CPU (4 Core 3.20GHz). The VMs are run on GraalVM CE version 20.2.0.

MiniJava adaptive Sobel filter. To evaluate the correct adaptation of the MiniJava VM on the accuracy/execution time trade-off, we use an image processing algorithm implementing a Sobel filter. To perform this trade-off, we created an approximate loop unrolling [39] module that affects a subset of MiniJava for-loops. This module interpolates a part of the values (0 to 7 loop iterations) depending on the monitoring of the CPU percentage.

The metric we choose to evaluate is the Sobel filter's execution time. Benchmarks are executed on Krun [5] and designed using JMH [42] to perform 10 warmup iterations (manually

inspected beforehand to ensure a steady state), then execute the program 30 times. These measures are repeated on 3 VM invocations to prevent the impact of the initial state [26]. We use the `stress` command (tool to impose load on and stress test systems) to generate CPU load during the benchmarks. With no CPU load we measure the cost of our approach when no adaptation is done, at 25% the adaptive VM starts to apply approximation and interpolates 1/2 of the values, then 3/4 and 7/8 at 50% and 75%, respectively. Finally, we look at how the VM behaves when the CPU starts to be saturated. We compare our approach to a normal implementation of the VM that uses the interpreter design pattern.

RobLANG battery optimization. The goal of the adaptation in the RobLANG language is to preserve battery while assuring the completion of the task in a certain proportion of the nominal time. We evaluate the correct adaptation of the robot behavior on three programs. First, we evaluate the impact on the *Turn* action using a program that makes the robot do complete rotations (360 degree) until battery depletion. Second, we evaluate the impact on the *Move* action using a program that makes the robot go forward meter by meter until battery depletion. Finally, we evaluate the impact on a combination of these actions using a program that makes the robot move in a square pattern until battery depletion.

The metric we choose to evaluate is the number of actions performed by the robot, *i.e.,* the number of rotations, meters, and squares, respectively performed compared to the case using the original interpreter. Each program is evaluated with three time constraint relaxation values (25%, 50%, and 75% of the nominal time). The trade-off is set to the most energy saving profile, hence the speed chosen by the solver is the lowest (respectively 75%, 50%, and 25% of the nominal speed). The simulator is configured to be deterministic on the experimentation computer according to cyberbotics guidelines[5]. Although small difference due to computation precision can still appear when different machines are used, the actions performed by the robot are long enough to mitigate these errors. We count the number of actions (rotations, meters, and squares) performed by the robot once per program-configuration combination.

HTML adaptive rendering. For the HTML adaptive rendering, we evaluate the benefits of our approach on the 100 most visited websites[6]. We apply two types of adaptations, with and without loss of information. For adaptation with loss, we use conditional loading of external content and lists perforation. For adaptation without loss, we choose to reduce image quality. We leverage on the structure of the proxy to create a cache and avoid the cost of downloading images multiple times. Out of the 100 websites, 45 deliver a degraded version of the content, 31 do not deliver the content due to security or aggressive adaptation, 21 cause errors, and 3 were removed for their adult content. The analysis of the results are performed on the 45 websites that deliver a degraded version of the web page.

The metric we choose to study is the energy consumption of a website access. We evaluate it using Website Carbon Calculator [52] algorithm, while evaluating the additional cost of our approach using the Intel RAPL [13] (Running Average Power Limit) tool to get energy consumption information from the hardware. Out of the 45 websites working with our approach, we selected 12 websites with different number of HTML tags to compute the average consumption of our approach independently of the website size. For each of those 12 websites, we measure the proxy consumption for handling a batch of 10 requests and the processsing of a single file for the HTML VM. We do not warm up the HTML VM because the steady state is not representative of the fresh VM invocations done by the proxy for each pages. Each of those experiments are executed 10 times with new VM invocations.

Feedback loop & trade-off reasoning abstraction. To evaluate the benefits of our approach in terms of abstraction, we evaluate the size of additional code needed in programs written for the normal language interpreters. This additional code contains, among other things, the feedback loop, the adaptations code, and supplementary code needed due to specificities for introducing a feedback loop.

For the MiniJava and RobLANG VMs the code is added to programs. On the other hand, HTML does not possess the expressivity to code these adaptations. To overcome this problem we evaluate the size of an additional JavaScript code that performs the adaptations on the HTML page. Since the adaptation needs to be done before going to the browser's HTML engine, this JavaScript code is written in the form of a browser extension intercepting the page. The metric we choose to study is the number of lines of code of those programs. All programs use the same coding convention.

5.2 Results and Discussion

Based on our experimentation, we are evaluating 1) the gain provided by the adaptation rules (according to the metrics aforementioned), 2) the relevance of their dynamic application according to the current context, and 3) the benefits of our approach in terms of abstracting the feedback loop and associated trade-off reasoning.

MiniJava adaptive Sobel filter. When applying the Sobel filter to an image, we measure the execution speed with different CPU load. When comparing the self-adaptable VM to the original VM, we report a range of slowdown and speedup from 0.493 when no optimization is done to 1.510

[5] https://cyberbotics.com/doc/guide/modeling#how-to-make-replicabledeterministic-simulations

[6] List available here: https://anonymous.4open.science/r/e8e73a84-3a28-451a-bb82-83643886bb8e/Evaluated_Websites.png

when interpolating 7/8 pixels. We report our results[7] in Table 2. The first column indicates the CPU load considered in the context model to infer the required interpolation rate.

%CPU (Interpol.)	Original time (s)	Adapted time (s)	Slowdown / Speedup
0% (0)	12.606 ± 0.012	25.574 ± 0.034	x0.493
25% (1/2)	12.599 ± 0.007	16.298 ± 0.008	x0.773
50% (3/4)	12.668 ± 0.007	10.883 ± 0.007	x1.164
75% (7/8)	12.674 ± 0.011	8.454 ± 0.072	x1.499
100% (7/8)	13.222 ± 0.044	8.759 ± 0.067	x1.510

Table 2. Performances of the Sobel filter on MiniJava's original and self-adaptable VMs depending on CPU load (and the proportion of interpolated pixels).

We observe that our approach introduces an overhead when no adaptation can be done. We assume this is due to a naive and inefficient implementation of the MAPE-K loop. However, the performance is improving when we adapt the VM to apply the loop perforation, and the VM correctly adapts itself according to the CPU load. We control manually the accuracy by checking the acceptable quality of the resulting images.

When trying to implement the adaptation manually, the code implemented to perform the trade-off reasoning is rather small (around 40 LoC). However, the general purpose nature of the language hampers the application of adaptations like approximate loop unrolling due to the lack of facilities to act on the domain concepts (here the for-loop). For instance, the loop for filling an array with values from computation (*function* in Listing 1 and 2) takes only one line in the for loop when using the Self-Adaptable VM.

```
for (int i = init; i < array.length; i += 1) {
    array[i] = function(i);
}
```

Listing 1. Example of adaptive for-loop when using MiniJava Self-Adaptable VM

On the other hand, when implementing the adaptations directly in MiniJava, each loop needs to be changed to perform the trade-off analysis before, adapting the progression of the loop, and perform the interpolation of the skipped iterations.

```
int step = loopPerforationConfig();
for (int i = init; i < array.length; i += step) {
    Type val = function(i);
    for (int j = 0; j < step-1; j++) {
        array[i+j] = val;
    }
    step = loopPerforationConfig();
}
```

Listing 2. Example of adaptive for-loop when using MiniJava interpreter

[7] https://anonymous.4open.science/r/5f92b1d8-0be1-4e19-8211-7b7e22cd85d1/

RobLANG battery optimization. We compare our approach to the baseline (original interpreter) on three programs (Turn, Move, and Square) measuring the number of actions performed for 0% (baseline), 25%, 50%, and 75% of time constraint relaxation. When using the *speed regulation* module, we report a range of enhancements in the number of actions performed while completing these actions in the time range allowed. We report the number of actions performed and the enhancements relative to the baseline in Table 3. Each row represents a different program while the columns represent the number of actions performed given a time constraint relaxation (0% (baseline), 25%, 50%, and 75%). Non-baseline cells also represent the relative enhancement between parentheses. Finally, the last line indicates the geometrical mean of the relative enhancement for each configuration of the module.

Program	Relaxation of the time constraint (in %)			
	0% *	25%	50%	75%
Turn	288	524 (182%)	1164 (404%)	3018 (1048%)
Move	87	154 (177%)	330 (379%)	860 (989%)
Square	74	142 (192%)	294 (397%)	794 (1073%)
Mean	100%	183.56%	393.19%	1036.06%

(*) We use the original interpreter as baseline.

Table 3. Number of actions performed by robot (and relative enhancements compared to the baseline) depending on the program and the percentage of time constraint relaxation.

We observe that our approach allows us to greatly improve the number of actions that the robot can perform before running out of battery while respecting the constraints expressed by the language user.

The manual implementation of the adaptation in RobLANG benefits from its DSL nature. As a DSL, RobLANG provides all the facilities to affect the domain concepts (*e.g.,* robot movement), making it easy to wrap the calls to the movement function with calls to the feedback loop and the adaptations. Yet, the lack of existing code for RobLANG makes impracticable the implementation of the trade-off analysis done in the Self-Adaptable VM. Indeed, the multi-constraint optimization problem solved by the Choco solver in our approach cannot be used in RobLANG. RobLANG does not include facilities to call external programs, the realization of the trade-off analysis necessitates the implementation of a complete solver in RobLANG. In RobLANG, as well as many other DSLs, the development cost of this solver would be unaffordable.

HTML adaptive rendering. We evaluated the average consumption of our approach on 12 websites and added it to the consumption of the websites when using our HTML VM. For the 45 websites used to evaluate the impact of our approach, we measure an average decrease of energy consumption of 63.8% ranging from -8.7% to 97.2% and with a 95% confidence interval of [54.2%, 73.4%] compared to normal

web browsing when the trade-off is set to the most energy saving profile[8].

These results demonstrate the non-negligible impact on energy consumption, and highlight the need of self-adaptation in rendering engines (*e.g.,* in HTML, CSS, and JS VMs).

The manual adaptations for HTML were implemented in the form of a browser extension in JavaScript to perform the adaptation before the processing of the HTML code by the engine. In this case, we bridge the last two scenarios, using a GPL (JavaScript) that benefits from a large available codebase and acting on a tightly coupled DSL (HTML) using the facilities provided to act on the DSLs concepts. However, this implementation of the adaptations requires the creation of a server to handle the image degradation module. We choose to take our Java implementation as reference for the size of this server. Taking all into account, the implementation is ~300 lines of code (150 in JavaScript and 150 for the server).

Discussion. Based on the three examples of *Self-Adaptable VMs*, we note the correct adaptation of the languages semantics according to the context. The example of the MiniJava VM highlights the necessity (and the difficulty) to implement the overall feedback loop including the trade-off analysis. This is especially true when the execution time of the program is one of the property of interest. On the other hand, the HTML VM shows that the adaptation modules need to deal with the diversity of programs running on the VM. For instance, the *conditional loading* module is efficient on handcrafted websites, while Single Page Application (SPA), such as facebook, cannot be displayed at all if the *conditional loading* module is applied. As the language user is oblivious to the self-adaptive nature of the virtual machine, the VM needs to deal with this diversity.

When looking at the development cost of the adaptations and trade-off analysis of the three implemented VM, the results show that the impact of our approach depends on the target language expressivity and community. In the case of a GPL like MiniJava, the lack of possibility to act on the concepts of the language hinders the automation of the adaptations application. On the other hand, DSLs like RobLANG can benefit from the abstraction of the domain in the language to wrap easily the semantics with adaptations. However, the smaller community and code base of DSLs can make unaffordable the development of a trade-off reasoning engine for a system, while GPLs benefit from their large community and numerous maintained libraries. While the approach for HTML reduces less the efforts for the language user than in the other cases due to the good synergy between JavaScript and HTML, our approach allows language users to avoid the hindrance of approaches similar to MiniJava (GPL) and makes practical complex trade-off analysis in cases like RobLANG (DSL). In all cases, our approach also frees language users from the design implications of the feedback loop, the feature interactions, and the monitoring of the environment. The latter may not even be supported in some languages.

To summarize, *Self-Adaptable Virtual Machines* provide an abstraction to address the complex trade-off between various properties and to dynamically adapt the VM accordingly. Our experimentation demonstrates the benefits according to the use cases (*e.g.,* performance, energy), but also raises the need for adaptation designers to implement the adaptation rules efficiently, as well as for language engineers to implement the overall MAPE-K loop. We also show that the cost to implement manually the feedback loops and trade-off reasoning is dependent of the language used and its community. On the other hand, our approach allows the abstraction of the adaptation system, while also avoiding system-wide design implications and feature interaction.

5.3 Threats to Validity

The main internal threat to validity is the complexity of reliable and precise energy and time measurement, and their correct analysis. To mitigate this risk, our companion web pages[7,8] provide all the data and algorithms used to validate our work. The main external threat to validity is the chosen use cases for our experimentation. For the HTML VM, we selected the most visited websites, but other websites could have been selected, *e.g.,* according to their internal website architectures. For MiniJava, the Sobel filter with approximate loop unrolling may be too specific to generalize to other data processing or iterative programs. Another threat comes from the simulator used to run the experiment on RobLANG. We mitigated this risk by following the instructions[5] to make the simulation reproducible. Finally, the number of lines of code may not be the best metric to evaluate the development cost of the adaptation system.

6 Research roadmap

When considering a SAL, we now review in this section the main expected features to support the design and use of SALs, and the underlying challenges. This support could take various forms such as independent tools, but can also take the form of a set of plugins for integration in a code-editor or IDE (*e.g.,* VS Code, Eclipse). For this reason, we discuss in this section the support independently of the way it is integrated into the development workflow. This section is structured around the two feedback loops presented in Section 3. Section 6.1 covers the support to setup, configure, and analyze the *runtime feedback loop*, while Section 6.2 discusses the support of the *design feedback loop*.

6.1 On the Support of the Runtime Feedback Loop

This section does not cover the actual implementation of the SAL by a language engineer, as these features are heavily influenced by the well-researched MAPE-K loop, i.e., continuous handling of online monitoring, trade-off reasoning,

[8] https://anonymous.4open.science/r/e8e73a84-3a28-451a-bb82-83643886bb8e/

decision making, and change propagation. A language engineer provides these capabilities with first-class constructs in a meta-language with support from a dedicated language workbench, which is outside the scope of this paper. The features discussed in this section are from the point of view of the language user and serve as use cases that need to be supported by such a language workbench for SAL.

A language user requires *(i)* features for setup and configuration of a feedback loop, *(ii)* more advanced analysis features that need to be provided by development tools , and *(iii)* more advanced analysis features that need to be provided by development tools to reason about the broader impact on the socio-technical modeling system. All features require the definition of protocols and interfaces that support setup, customization, and advanced analysis.

Feedback loop setup and configuration. In a SAL, feedback loops analyze trade-offs among various quality and functional objectives considering language use as well as the runtime environment of the running software system. In some cases, a language engineer may be in a position to completely define and configure a feedback loop at the time of the definition of a SAL. In many cases, however, a language user will *setup feedback loops* at the start, *i.e.,* decide which ones are needed for an application. Then, a language user will configure these feedback loops, by specifying *(i) preferences*, *i.e.,* how important is an objective compared to others, and *(ii)* the concrete *data sources* that need to be monitored.

Beyond such settings, a language user may also want to *customize the structure of a feedback loop*, *i.e.,* add/remove considered objectives or solutions, define the impact of a solution on an objective, define the types of data sources that need to be monitored for a solution and how a data source is monitored, and define how a solution is applied to the software system or language. All of these configurations may be done at design-time or dynamically at runtime. The configuration may go even further and possibly include the choice or customization of the reasoning mechanism of the feedback loop (e.g., goal models, statistical regression, or neural networks).

Tool support for software analysis. The feedback loops apply dynamic adaptations on the language according to its use and the runtime environment. When a language user designs a system with a SAL to be self-adaptable, it is crucial to support him/her in the understanding of the resulting software systems with advanced *analysis features*. More specifically, one can think about *visualization tools* to characterize the execution space in which the software system will operate according to possible adaptation of the language runtime. Such tools go beyond existing support for visualization. They need to make the implicit feedback loop explicit to the language user, showing its impact on the execution space which emerges only during system execution, and providing a better understanding of the correctness envelope in which the system will operate. Besides, dedicated *validation & verification tools* need to ensure that liveness and safety properties hold within this execution space. For instance, test cases may be automatically amplified to make sure the emerging execution space is actually covered.

Tool support for broader impact analysis. The decisions made during a feedback loop are often based on trade-offs among system qualities. Support is hence needed in development tools to ensure that the qualities covered by a SAL match with the quality expectations of the stakeholders in the socio-technical (modeling) system, which may shift over time. This requires an understanding of the broader impact on the entire system, not limited to the software system with its execution space. Additional *visualization and analysis* features are needed that target the problem space in addition to the execution space. For example, a SAL may adapt the availability of language constructs based on the language user's skill level, hence requiring customizable models of these stakeholders.

Reference framework for common implementation. The implementations of the runtime feedback loop of SALs are often similar. Abstracting the definition of this loop in a reference framework would help the language engineer to focus on the specificities of this loop for its language. More specifically, the language engineer will focus on the definition of the monitored environment, select the appropriate analysis and plannification of adaptation, and define the correctness envelope of the adaptations with, for instance, explicit specification of variation points in the semantics. In addition, a common implementation will facilitate the creation of language-agnostic tools, such as debuggers, using a tooling API, as Van De Vanter et al. have done for the Truffle framework. [49].

6.2 On the Support of the Design Feedback Loop

Complementary to the *runtime feedback loop*, the *design feedback loop* encompasses the adaptation of the use of the language (*e.g.,* metamodel, pragmatics) with respect to the development context acquired from monitoring of the data and models from the modeling environment. Hence, we identified new research challenges in the realization of this second feedback loop.

A model for the development context. The design feedback loop adapts the language according to a monitored development context. To correctly adapt, the model of the development context resulting from the monitoring needs to reflect as precisely as possible the changes in the use of the language. For instance, this model could include created models as well as atomic changes made to the models and who made these changes.

Detection of language evolution opportunities. To adapt and evolve a language, there is a need for an analysis and planning system that reasons about the concepts manipulated by this language. This new analysis and planning system will

leverage on the previously mentioned development context model to detect and reify language evolution opportunities (*e.g,* new patterns, constructs that need to be deprecated).

Closed and open-world adaptations. We envision two scenarios for the adaptation performed by the *design feedback loop*. First, we close the world by defining the set of possible language variants in the form of a software language product line [37]. In this scenario, the adaptation of the language is performed through a migration from one language variant to another. On the other hand, with an open-world hypothesis, language variants do not need to be known beforehand and new patterns unforeseen at language design time may emerge. In both scenarios, the adaptations would need to provide a consistent way for co-evolving the models with respect to the change in the language.

Historical zoom over adapted program evolution. Using a language that evolves over time can hamper the understanding of the new abstractions for its users. In order to help the language users to keep up with language evolution, we propose to provide tooling to the language user to zoom in (to visualize the patterns that were reified as language concepts and keep a consistent understanding of the language) and zoom out (to manipulate the abstractions proposed by those patterns).

Predictive model for language evolution. As a long term goal, we envision a feedback loop with analysis and planning phases able to detect trends in the use of the language by the developers and predict their future needs. Hence, the feedback loop would use this information to provide new constructs to the developers when they need them rather than reifying patterns after they have been used.

Communication in the socio-technical system. Adaptation may lead to changes that require enactment at the model level and changes to the language pragmatics at the language level. Both require human intervention of varying degrees and there is a need to be able to reason about these changes on the whole socio-technical modeling system. For example, a feedback loop based on language use may lead to new modeling guidelines, which need to be communicated to language users through improved training. Human intervention is needed for the creation of new training material and scheduling of training sessions, and one needs to reason about the impact of these measures on the language users' skills.

7 Related work

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. All these language concerns are specified using one or several meta-languages provided by a language workbench that will compile or interpret these specifications to drive the development of all the services. The long standing history on language theory provided well-defined meta-languages (*e.g.,* EBNF [17] or MOF [20] for the syntax, or SOS [38] for the operational semantics). All these meta-languages provide an unambiguous way of defining the language concerns.

However, with the advent of Digital Twins [24] and DevOps [29], all these meta-languages fall short in considering continuous improvement based on the data collected from the use of the language, either at design time or at runtime. In the modeling community, the use of *models@runtime* [6] has been explored to offer an abstract representation of a running system, but no relationships have been established with the initial language used for defining the various system models. Recent work explored specific approaches to feed back the usage or production data directly to the language environment [10, 54], such as the language user can leverage on this for trade-off analysis and decision making. These approaches are currently apart from, and loosely coupled with, the language specifications, which prevents any generalization to domain-specific languages and the ability to take them into account for the various language services.

Design of Self-Adaptive Systems. The definition of a feedback loop has been investigated in the context of Self-Adaptive Systems [9], with a field mature enough to provide time-honored patterns such as the MAPE-K loop [27]. 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 [11].

In addition to patterns, the community built reference architectures (*e.g,* three layer architecture [30], MORPH [8], PLASMA [46]) and frameworks (*e.g,* Executable Runtime Megamodels [51], DCL [36], ActivFORMS [23], Ponder2 [47])

that provide abstractions helping the design of self-adaptive systems. Among other things, some frameworks provide abstractions for the implementation of the feedback loop and the decision process. For instance, the Ponder2 framework [47] provides abstractions to define policy-based self-adaptive systems and allows to configure and control managed elements using the PonderTalk language. While PonderTalk eases the management of the policies applied in Ponder2 systems, it does not abstract the policies definition and adaptation implementation from its users.

On the other hand, executable runtime megamodels [51] offer to abstract feedback loops implementation by defining them explicitly at a higher level of abstraction. By using *megamodels* (*i.e.,* models of models-relations) as runtime models for self-adaptive systems, the feedback loop functions are defined through model operations and their control flow in the megamodel. Similarly, Dynamic update of Control Loops (DCL) [36] abstracts the implementation of the feedback loop by associating elements of the system goal-model to feedback loop functions. In addition, the approach leverages on the KAOS goal-models operationalization concept to link the conceptual functions to the actual implementation. However, in both cases, the users of those frameworks will remain in charge of the implementation of the loop functions, including the trade-off reasoning.

Although it is desirable for these frameworks to let their users implement these parts when the adaptation is the primary system concern, the complexity and the design implications nevertheless become unbearable when this is not the case.

Trade-off reasoning at language level. The trade-off between quality of service and non-functional goals can be desirable in wide-ranging types of systems, from streaming platforms to Internet of Things. These trade-offs are already well addressed by approximate computing techniques [33, 55]. Approaches like ACCEPT [41] and Green [4] were developed to address these trade-offs at the language level using approximate computing. These approaches provide annotations to the language user and calibrate the approximations to infer the configuration that best deliver the expected service. On the other hand, PowerDial [21] reduces the power consumption by adapting the value "dynamic knobs" that affect the Power/QoS trade-off. However, none of these approaches can address complex trade-offs that involve the execution environment and multiple properties of interest.

Algorithm selection for QoS. Changing the semantics is not the only way to improve QoS properties at the language level. Several approaches studied the selection of the best algorithm to perform a task in the given context [3, 14, 44]. The PetaBricks language was extended with the concept of variable-accuracy to perform its algorithm selection depending on the accuracy of the result [3]. In the same way Diniz and Rinard [14] proposed to compile multiple versions of a program and periodically select the best given the sampled current context. On the other hand, Soman et al. [44] focused on the virtual machine and not the program with the dynamic selection of the garbage collection strategy. Yet, our approach focuses on the domain-specific adaptation concern, while existing work focuses more on 'system' concerns.

Injection of adaptations in the semantics. To offer better separation of cross-cutting concerns (*e.g.,* security, logging), techniques such as Aspect Oriented Programming (AOP) propose to implement those concerns separately and inject them statically at compile time or dynamically at runtime. The advice (code of the cross-cutting concern) is injected at different join points (places where one can inject code) specified by a pointcut expression (expression specifying affected join points). While the use of AOP is a good way to centralize the concern of adaptation [56], it has limitations. First, this approach [56] 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 system engineer (named language user for SALs) that knows which ones to target. Finally, the system engineer will have to deal manually with the feature interaction of the different aspects used.

On the other hand, the use of AOP is relevant for the language engineer to inject the calls to the self-adaptation mechanism. This approach was explored in the construction of the MiniJava Self-Adaptive Virtual Machine to wrap the node's semantics with the appropriate calls.

Use Meta-programming to abstract adaptation. The use of metaprogramming has proven to be a viable solution to add new paradigms to a host language [15]. However, our framework aims to be applicable independently of any technological stacks and languages, while providing specific first-class concepts to implement the adaptation loop (in comparison to a more general metaprogramming framework that does not specifically support the language engineer/user in the implementation of the adaptation loop).

SALs as Utility-based agents. The adaptive nature of SALs allows them to be classified as "intelligent agents". More precisely, SALs map to the special case of Utility-based agents [40]. A utility-based agent, as well as SALs, is an agent that tries to maximize the satisfaction of its goals (i.e., its utility). A model of possible programs of such agents is presented by Russell and Norvig [40], and consists of 5 steps: 1) update our representation of the world, 2) prediction of the consequence of actions, 3) evaluation of the utility in this new state, 4) choose the best action to perform, and 5) perform the action. In our case (1) is done in the monitoring phase, (2) and (3) are the analysis phase where we evaluate the goal model (*i.e.,* utility) using impact models of the modules (*i.e.,* prediction of consequence), (4) is the planning phase where we choose the modules to be activated, and (5) is the execution phase with concrete activation of the modules.

8 Conclusion

We introduce the concept of *Self-Adaptable Language* (SAL) that incorporates the definition and execution of feedback loops and trade-off reasoning at the language level and abstracts it for the language user. The proposed L-MODA (*Language, Models, and Data*) conceptual reference framework details two feedback loops in the socio-technical modeling system, comprised of the running software system, its modeling environment, and its language engineering environment. We report on the concept of Self-Adaptable Virtual Machines as an implementation of the *runtime feedback loop* in languages operational semantics and provide promising experimental results. Furthermore, we present a roadmap of the key features expected by language users across the lifetime use of a SAL including setup and configuration of its feedback loops, analysis and consistency management features provided by its IDE, and the understanding of the broader impact on the overall socio-technical (modeling) system. We call upon the SE community to increase its efforts to realize self-adaptable languages.

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

[3] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 85–96.

[4] Woongki Baek and Trishul M Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 198–209.

[5] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 52.

[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] 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.

[9] 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. https://doi.org/10.1007/978-3-642-02161-9_1

[10] Jürgen Cito, Philipp Leitner, Martin Rinard, and Harald C. Gall. 2019. Interactive production performance feedback in the IDE. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 971–981. https://doi.org/10.1109/ICSE.2019.00102

[11] 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.

[12] Autonomic Computing et al. 2006. An architectural blueprint for autonomic computing. *IBM White Paper* 31, 2006 (2006), 1–6.

[13] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. IEEE, 189–194.

[14] Pedro C Diniz and Martin C Rinard. 1997. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*. 71–84.

[15] Dragan Djuric and Vladan Devedzic. 2010. Magic potion: Incorporating new development paradigms through metaprogramming. *IEEE software* 27, 5 (2010), 38–44.

[16] 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.

[17] International Organization for Standardization. 1996. *ISO/IEC 14977:1996: Information technology – Syntactic metalanguage – Extended BNF*.

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

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

[20] Object Managament Group. 2016. *Meta Object Facility, version 2.5.1*.

[21] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic knobs for responsive power-aware computing. *ACM SIGARCH computer architecture news* 39, 1 (2011), 199–212.

[22] 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).

[23] 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.

[24] Klementina Josifovska, Enes Yigitbas, and Gregor Engels. 2019. A Digital Twin-Based Multi-modal UI Adaptation Framework for Assistance Systems in Industry 4.0. In *HCI (3) (Lecture Notes in Computer Science, Vol. 11568)*. Springer, 398–409.

[25] 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.

[26] 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.

[27] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan 2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[28] 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. https://doi.org/10.1145/3371906

[29] Gene Kim, Patrick Debois, John Willis, and Jez Humble. 2016. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press.

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

[31] 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.

[32] Olivier Michel. 2004. Cyberbotics Ltd. Webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems* 1, 1 (2004), 5.

[33] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4, Article 62 (March 2016), 33 pages. https://doi.org/10.1145/2893356

[34] 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.

[35] Gunter Mussbacher, Benoit Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien Mosser, Houari Sahraoui, Eugene Syriani, Dániel Varró, and Martin Weyssow. 2020. Expert Voice: Opportunities in Intelligent Modeling Assistance. *Software and Systems Modeling* (2020).

[36] 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.

[37] Gilles Perrouin, Moussa Amrani, Mathieu Acher, Benoît Combemale, Axel Legay, and Pierre-Yves Schobbens. 2016. Featured model types: towards systematic reuse in modelling language engineering. In *Proceedings of the 8th International Workshop on Modeling in Software Engineering*. 1–7.

[38] Gordon D Plotkin. 2004. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming* 60-61 (2004), 3 – 15. https://doi.org/10.1016/j.jlap.2004.03.009 Structural Operational Semantics.

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

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

[41] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01* 1, 2 (2015).

[42] A Shipilev et al. 2018. JMH: Java microbenchmark harness, Oracle Corporation.

[43] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 124–134.

[44] Sunil Soman, Chandra Krintz, and David F Bacon. 2004. Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th international symposium on Memory management*. 49–60.

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

[46] 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.

[47] 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.

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

[49] 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 preprint arXiv:1803.10201* (2018).

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

[51] 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.

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

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

[54] Jos Winter, Maurício Finavaro Aniche, Jürgen Cito, and Arie van Deursen. 2019. Monitoring-aware IDEs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 420–431. https://doi.org/10.1145/3338906.3338926

[55] Q. Xu, T. Mytkowicz, and N. S. Kim. 2016. Approximate Computing: A Survey. *IEEE Design Test* 33, 1 (Feb 2016), 8–22. https://doi.org/10.1109/MDAT.2015.2505723

[56] 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.

[57] Eric Yu. 1995. *Modelling Strategic Relationships for Process Reengineering*. Ph.D. Dissertation. Department of Computer Science, University of Toronto.