A Domain-Specific Language to Specify Planned Architectures of Adaptive Systems

Daniel S.M. Santibanez
Exact Sciences Department, ULA
Osorno, Chile
Computing Department, UFSCar
São Carlos, Brazil
daniel.sanmartin@ulagos.cl

Valter Vieira de Camargo
Computing Department, UFSCar
São Carlos, Brazil
valtervcamargo@ufscar.br

ABSTRACT

Adaptive Systems (ASs) are able to monitor their own behavior and adapt it when quality goals are not accomplished. MAPE-K is a reference model that provides abstractions to design ASs. Structuring such systems with these abstractions brings many benefits, some related to maintenance and some related to comprehension of the architecture. However, many existing ASs are not designed according to MAPE-K, making those abstractions hidden in their architecture. Architectural Conformance Checking (ACC) is a technique for checking if the current architecture (CA) of a system is obeying the rules prescribed by a planned architecture (PA) or even by a reference model, like MAPE-K. In this paper we present DSL-Remedy, a language for specifying PAs in the context of Adaptive Systems. Our language provides adaptive-systems specific abstractions so that engineers can be more precise when describing the planned architecture. Besides, as our DSL is ASs-specific, it incorporates some rules already known in MAPE-K. We have evaluated our DSL comparing it with a general-purpose DSL and the results show improvements in the productivity.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

adaptive systems, architectural conformance checking, architectural drift, controlled experiment

ACM Reference Format:
Daniel S.M. Santibanez and Valter Vieira de Camargo. 2021. A Domain-Specific Language to Specify Planned Architectures of Adaptive Systems. In *15th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3483899.3483903

1 INTRODUCTION

After years of maintenance, the architecture of software systems tend to deviate from the architecture that was initially planned. A possible way to check if the current implementation is becoming different from the planned architecture (PA) is by employing Architectural Conformance Checking (ACC) approaches, whose goal is to detect architectural drifts in existing systems. The motivation is obvious, if the system is deviating from its planned architecture, its quality attributes may not be met anymore. ACC approaches normally involve the following steps: *i*) specifying the PA making evident the hierarchical compositions and the communication rules among the architectural elements; *ii*) Map source-code elements of the system to the architectural elements prescribed in the PA and *iii*) Perform the checking/comparison between both [26].

Most of the existing ACC approaches are domain-independent, i.e., they deliver canonical and domain-independent architectural abstractions such as components, layers and modules [23, 25, 28]. So, software architects must work with these abstractions along the whole ACC process by mapping all the source code elements to them. However, in more specialized domains (like Adaptive Systems), domain-specific abstractions become very important and strongly influence how systems are structured. In these cases, systems have components/abstractions with very specific responsibilities, that guide how the how the architecture must be designed.

Adaptive Systems (ASs) are able to autonomously cope with disturbances that can show up in the environment, within themselves and in their quality goals [5]. MAPE-K is a well known reference model for guiding the design of ASs [2, 11]. It prescribes the main abstractions that must be used for architecting the adaptive parts as well as the expected structural and communication rules between those abstractions [11]. Although MAPE-K is well known and provides a suitable guide for architecting ASs, it is possible to find ASs that do not follow the basic principles of MAPE-K [19, 29].

In this paper we present a language called DSL-REMEDY for specifying Planned Architectures of Adaptive Systems. Our language can be considered domain-specific, as it provides abstractions that are specific of adaptive systems, such as monitors, planners, knowledge and sensors. Our premise is that engineers can be more precise when specifying ASs-specific elements and the whole ACC process can be done in a more accurate way.

Besides, as our language is AS-specific, some communication rules prescribed by MAPE-K is already incorporated in the language, saving engineers from the tasks to write down all the rules. We have compared our language with another one that is not domain-specific called DCL-KDM. The goal was to evaluate productivity of software architects at time to specify the adaptive part of an AS.

We claim that our approach have the following advantages: *i*) It enables software architects to use the domain vocabulary to specify the PA. This allows them to focus just on the important points of the architecture, writing more concise specifications leading to better productivity. *ii*) It allows domain rules to be incorporated in the specification and can be generated automatically, leaving architects out of having to worry about the canonical rules.

This paper is structured as follows: Section 2 explains ASs and ACC; Section 3 shows a robotic system to aid in the description of the DSL; Section 4 describes DSL-REMEDY; Section 5 reports the controlled experiment; Section 6 outlines related work and Section 7 makes concluding remarks and suggests future work.

2 BACKGROUND

2.1 Adaptive Systems and MAPE-K

ASs are systems with the capability to adapt their behavior at runtime to changes in its execution conditions and user requirements [13]. Nowadays they are extremely demanded in pervasive, mobile and embedded computing environments because their execution context requires a high degree of unpredictability and dynamism.

MAPE-K is a reference model that uses the concept of control loops for designing ASs. MAPE-K is an acronymn for Monitor, Analyzer, Planner, Executor and Knowledge [22]. Figure 1 shows a schematic view of MAPE-K that we consider as a base reference model [11]. Normally, a system that is considered adaptive is composed of the Managed Subsytem, which is the bigger base part, and one or more Managing Subsystems, which are modules responsible for performing the adaptations. Notice that MAPE-K is much more devoted to design the adaptation parts than the base system itself.

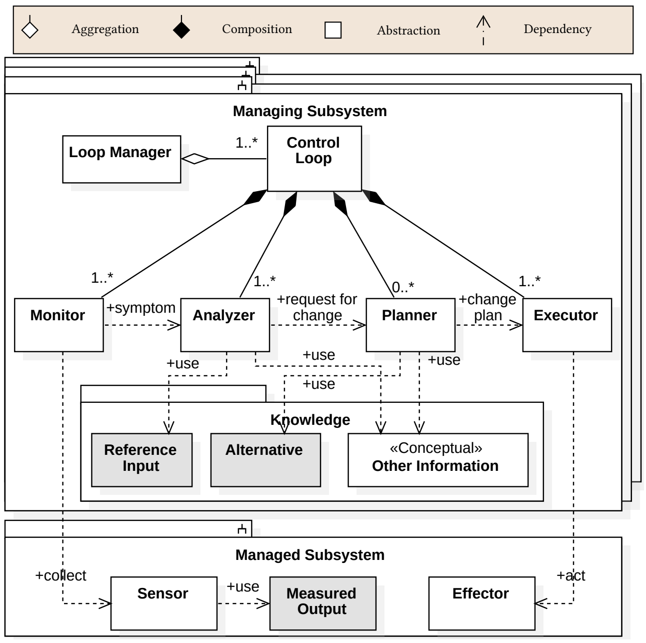

Figure 1: MAPE-K enriched with lower-level abstractions

MAPE-K of Figure 1 contains the conventional known abstractions and also some lower level abstractions presented by other works (Measured Outputs, Alternatives and Reference Inputs) [1, 33, 34]. Therefore the abstractions that appear in this figure, as well as the relations among them, are those we consider important to be evident in the architecture of most ASs. However, it is important to emphasize that different combinations of these abstractions are possible, providing flexibility for architects to decide the best combination. For example, it is perfectly possible to have an architecture with two or more control loops or with two or more monitors.

Besides the main known abstractions, there are others that are not represented in the canonical MAPE-K model [33, 34]. These abstractions are in a lower level of abstraction and they are important when architects are working with detailed design of the system [33]. Notice that grey boxes indicate abstractions that are not present in the standard MAPE-K. In the next paragraph we give a brief description of each of these lower level abstractions.

Alternative represents a set of available options that an AS uses for changing the system behavior [1]. *Reference Inputs* consists of the concrete and specific set of values that are used to specify the state to be achieved and maintained in the managed system [33]. *Measured Outputs* consists of the set of values that are measured in the managed system. Naturally, as these measurements must be compared to the Reference Inputs to evaluate whether the desired state has been achieved [33].

In this work we consider MAPE-K as a base reference model for adaptive systems. That means some rules of this reference model must be preserved in planned architectures for adaptive systems. Observing Figure 1, we highlight two kinds of rules: *i*) structural rules and *ii*) communication rules. Structural rules are those related to composition of the abstractions. For example: Loop Managers must be inside Managing Subsystems; Monitors must be declared within Control Loops, Sensors must be declared inside Managed Systems and Alternatives must be declared inside Knowledge. There is an exception related to Control Loops. They can be declared as inside Managing Subsystems or inside Loops Managers. The latter occurs when two or more Control Loops (not necessarily deployed in the same location) need some kind of interaction for achieving the adaptation goal. Every MAPE-K abstraction must be declared within an existing Control Loop. Reference Inputs as Alternatives must be declared within an existing Knowledge. Sensors, Measured Outputs and Effectors must be part of a Managed Subsystem so these abstractions just can be declared on this type of subsystems.

Regarding the communication rules, they are related on how the abstractions are expected to communicate. They are present in Figure 1 in two ways. The first is as UML associations tagged with representative names. These communication rules describes the expected communications among abstractions. They can be considered the "must use" rules. Therefore, it is expected that Monitors collect information from Sensors; Executors act over Effectors, Analyzers and Planners perform queries in the abstractions of the Knowledge, etc. The second form of communication rules occurs when there is no relation among abstractions. In this case, this means that the abstraction "must not use" others. Therefore, we need a mechanism to specify when an abstraction is not able to communicate to others in a explicit way and also assume that when there is no communication rule the relationship is forbidden.

Although the rules we have described are the canonical ones in MAPE-K, the implementation of a DSL should be flexible enough for software architects have the control over the specifications.

2.2 Planned Architectures in ACC Approaches

Architectural-Conformance Checking (ACC) is one of the main activities in software quality control. The goal is to reveal relations, constraints and other architectural rules foreseen in the Planned Architecture (PA) that were violated by the system's implementation [15]. Usually these violations are called drifts and occur due to unconscious erosion of the architecture. A drift occurs when there is a reference model (or planned architecture) to be followed, but the current implementation of the system presents points which diverge from the rules prescribed [17]. Architectural drifts not necessarily result in dramatic problems; they may be there consciously. However, it is advisable to be aware of them, since maintenance activities can be compromised [24].

One of the main step of ACC approaches is the specification of the PA which is an artifact where architectural elements and rules must be declared. Typically this is performed by using two different techniques: *i*) reflexion models [21] and *iii*) Constraint languages [30]. In the first one software engineers define a high-level model of the system based on their experience. This model is composed of entities that will be mapped with entities of a source model that previously was generated from the source code. In the second one software engineers use constraint languages to declare architectural elements and the constraints between these elements. The language adopts a vocabulary/terminology, usually a generic one like components, modules, subsystems, etc.

3 ADAPTIVE ROBOTIC SYSTEM

The Adaptive Robotic System aims at monitoring indoor environments following walls. It uses a light sensor for following the walls, trying to keep a constant distance of them. The effort of the robot to keep a constant distance from walls is controlled by a first control loop which makes adjustments in the percentage of turning the wheels and in the velocity. Besides, there is a slower control loop over the first one which analyze the adjustment parameters and change them to improve the performance of the robot. The goal is to make the robot to move as straight as possible.

Figure 2 shows schematically the Planned Architecture for the system. All blocks/rectangles/packages (as the bigger as the smaller ones) represent instances of architectural abstractions available in our DSL. By the stereotypes it is possible to see the names of the abstractions available in the DSL. In the upper part of the figure there is the Managing Subsystem, called *adaptationManager.* In the lower part there is the Managed Subsystem, called *Environment Guard Robot.* For simplicity reasons, the Managed part is not detailed, but usually this is much bigger than the Managing part. As can be seen, the Managing aggregates just one Loop Manger *loopManager.* The abstractions *ManagingSubSystem* and *Managed-Subsystem* are "default" abstractions, as they must be presented in any specification.

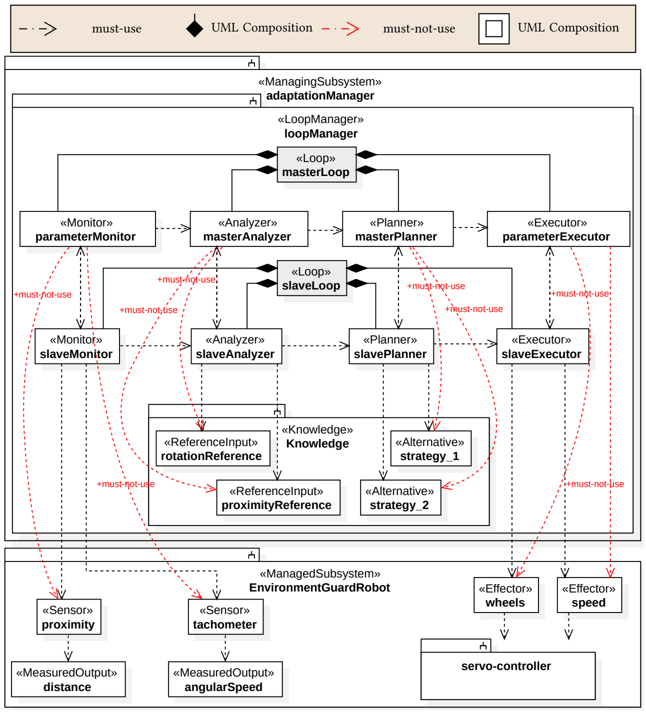

Figure 2: Adapted UML to represent the Robotic System

In this case, the *LoopManager* aggregates two Control Loops, *masterLoop* and *slaveLoop*. Each one of the Control Loops was designed to have four abstractions; Monitor (*parameterMonitor, slaveMonitor*), Analyzer (*masterAnalyzer, slaveAnalyzer*) , Planner (*masterPlanner, slavePlanner*) and Executor (*parameterExecutor, slaveExecutor*). In addition, *slaveLoop* also contains a Knowledge abstraction called *knowledge* that is composed of two ReferenceInputs (*proximityReference, rotationReference* and two Alternatives *(strategy_1, strategy_2*), which represent different strategies of adaptation.

The Managed is composed of two Sensors (*proximity, tachometer*), two effectors; *wheels* and *speed*, two MeasuredOutput; (*distance, angularSpeed*) and a generic component called servo-controller which is not detailed in the specification of the DSL.

There are two types of relations that can be identified in this figure. One type is represented by the arrows among the elements - *communication rules.* The another type is represented by the hierarchical compositions among them - *structural relations.* The last one occurs when element/abstraction is within another one. Figure 2 shows 26 rules of type *must-use*, 8 rules of type *must-not-use* and when a communication rule is not present it means that the relationship is forbidden.

4 DSL-REMEDY

DSL-REMEDY is our language for specifying planned architectures in adaptive systems. This language is part of a more complete ACC approach as can be seen in Figure 3. The step A is when software architects create/specify a PA using the DSL-REMEDY and this normally happens (but not always) at the early stages of life cycle.

Notice that steps B and C were intentionally draw in lightgrey indicating they are not covered in this work.

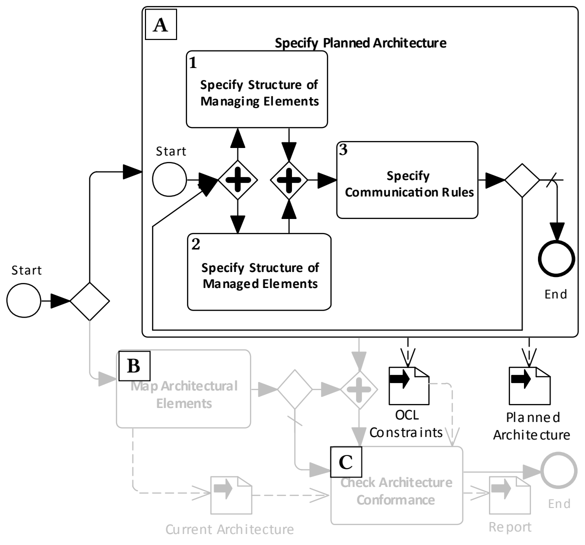
A
Specify Planned Architecture
1
Specify Structure of Managing Elements
Start
3
Specify Communication Rules
2
Specify Structure of Managed Elements
End
Start
B
Map Architectural Elements
OCL Constraints
Planned Architecture
C
Check Architecture Conformance
End
Current Architecture
Report

Figure 3: REMEDY approach

The specification process is divided in three interdependent substeps: *i*) Specify Structure of Managing Elements - When software architects specify (Step A.1) the adaptive elements that must exist in the system (monitor, analyzer, planner, executor, etc); *ii*) Specify Structure of Managed Elements - when the core elements are specified (Step A.2) employing common architectural abstractions (layers, components, modules, etc) and *iii*) Specify Communication Rules (Step A.3).

Step B aims at obtaining a representation of the Current Architecture (CA) of the system. Here software architects must map the architectural elements declared in the specification (monitors, analyzers, layers, components, modules, etc.) to the source code elements (variables, methods, etc.). This is the moment in which architects inform how the abstract elements are materialized in the source code. Finally, in the step C the goal is to identify the drifts by comparing the PA with the CA. Internally, both the PA and CA are represented as KDM model instances [31]. REMEDY was implemented as an eclipse plugin which provides all user interfaces to support the mentioned steps. Although, we explain the PA in three different listings for better understanding in practice the specification of the PA must be done in one file.

4.1 Specify the Planned Architecture

4.1.1 Specify Structure of Managing Elements. In this substep software architects must specify the elements of the Managing Subsystem. Listing 1 shows the specification that corresponds to the Managing Subsystem of Figure 2. Bold words indicate the keywords of our DSL. The way the abstractions are composed follow a Java syntax/style which defines the structural rules.

An abstraction can be composed of others or solely, then the nomenclature to specify it follows:

`[AS_Abstraction] [ID] [{..}];` | `[AS_Abstraction] [ID];`

Where `[AS_Abstraction]` is a MAPE-K abstraction and `[ID]` is a unique string that identifies the abstraction in the whole specification. Inside the brackets architects must specify the abstractions that compound the abstraction of higher hierarchy. We intentionally implemented the unique abstraction ID strategy to simplify the access to them at time of specifying communication rules.

In Line 1 one must inform a name that identifies this PA. In this example, there is just one Managing subsystem (Line 2) and just one Loop Manager (line 3). A Loop Manager can manage one or more Loops and in this case there are two of them specified in the PA (lines 5 and 12). Loops can holds several MAPE-K abstractions and the *masterLoop*, which is the Loop that coordinates the *slaveLoop*, has four; a *parameterMonitor*, a *masterAnalyzer*, a *masterPlanner* and a *parameterExecutor* (lines 6-9).

```
1  Architecture EnvironmentGuardRobot-PlannedArchitecture {
2    Managing adaptationManager {
3      LoopManager loopManager {
4
5        Loop masterLoop withDomainRules{
6          Monitor parameterMonitor;
7          Analyzer masterAnalyzer;
8          Planner masterPlanner;
9          Executor parameterExecutor;
10       }
11
12       Loop slaveLoop withDomainRules{
13         Monitor slaveMonitor;
14         Analyzer slaveAnalyzer;
15         Planner slavePlanner;
16         Executor slaveExecutor;
17         Knowledge knowledge {
18            ReferenceInput proximityReference;
19            ReferenceInput rotationReference;
20            Alternative strategy_1;
21            Alternative strategy_2;
22 ...
23 }
```

Listing 1: Managing subsystem of the PA

As the robot has constrained resources, this specification enables the decentralization of Loops to a better scalability with respect to communication and computation. For instance, the *masterLoop* could be deployed in a remote server and the *slaveLoop* in the robot. Thus, the communication overhead is limited and the computational burden is spread over the two nodes [34]. The *slaveLoop*, is composed of 5 MAPE-K abstractions. It adds a Knowledge called *knowledge* in line 17. This abstraction is composed of four abstractions (lines 18-21); *proximityReference*, *rotationReference*, *strategy_1* and *strategy_2*. The first two are of type *ReferenceInput* and the second two are of type *Alternative*. The *proximityReference* holds a value that indicates the distance between the robot and the wall. The *rotationReference* holds a value that indicates the angular speed of wheels. Notice that Loops are specified with the keyword **withDomainRules** (lines 5 and 12) to activate domain rules of ASs which is explained in subsection 4.1.3.

4.1.2 Specify Structure of Managed Elements. In this substep software architects must specify elements of the Managed subsystem that deals with the domain functionality. Listing 2 shows the specification of the Managed Subsystem of Figure 2.

```
23 Architecture EnvironmentGuardRobot-PlannedArchitecture {
24 ..
25   Managed environmentGuardRobot {
26      Sensor proximity;
27      Sensor tachometer;
28      Effector wheels;
29      Effector speed;
30      MeasuredOutput distance;
31      MeasuredOutput angularSpeed;
32      Component servo-controller;
33   }
34 }
35 ..
```

Listing 2: Managed subsystem of the PA

Managing subsystems have communication with the Managed subsystem by means of touchpoints which commonly are implemented by sensors and effectors [11]. Line 25 declares a Managed subsystem which in this case is composed of two Sensors (lines 26 − 27), two Effectors (lines 28 − 29), two MeasuredOutput (lines 30-31) and one generic component (line 32). As we can see, our DSL provides these abstractions to be specified in the Managed subsystem section. In literature, there are several approaches to specify systems in a generic way by using abstractions such as layers, components, modules and interfaces [12, 30]. Thus to our purpose, we have started the integration of the approach developed by Landi et al. [6] with our DSL. This also relies in KDM models for checking the architectural conformance so it fits very well to our intentions.

4.1.3 Specify Communication Rules. In our DSL, we opt for having just two types of communication rules: *must-use* and *must-not-use*. The first one means that an abstraction *A* must access an abstraction *B*. The access type can be a callable method, objects creation, implementation of interfaces among others. The second one means that an abstraction *A* must not access an abstraction *B*.

Listing 3 shows the communication rules of Figure 2. Lines 41 and 42 enable the communication between *masterLoop*, *slaveLoop* and vice-versa. The DSL automatically checks whether there are rules that connect both abstractions or not. If it detects the absence of rules connecting them then one or more errors will raise at design time and PA will not generate its output. On the other hand, if software architects specify that a Loop must not use another one then all rules that connects both abstractions will not take into account in the generated output of the PA. Also, it is possible to interconnect LoopManagers which follow the same rules as Loops.

```
40 Rules{
41   loop masterLoop must-use loop slaveLoop;
42   loop slaveLoop must-use loop masterLoop;
43   monitor parameterMonitor must-use monitor slaveMonitor;
44   monitor slaveMonitor must-use monitor parameterMonitor;
45   monitor slaveMonitor must-use sensor proximity;
46   monitor parameterMonitor must-not-use sensor proximity;
47   monitor slaveMonitor must-use sensor tachometer;
48   analyzer masterAnalyzer must-use analyzer slaveAnalyzer;
49   analyzer slaveAnalyzer must-use analyzer masterAnalyzer;
50   analyzer slaveAnalyzer must-use reference-input proximityReference;
51   analyzer slaveAnalyzer must-use reference-input rotationReference;
52   analyzer masterAnalyzer must-not-use reference-input proximityReference;
53   analyzer masterAnalyzer must-not-use reference-input rotationReference;
54   planner masterPlanner must-use planner slavePlanner;
55   planner slavePlanner must-use planner masterPlanner;
56   planner slavePlanner must-use alternative strategy_1;
57   planner slavePlanner must-use alternative strategy_2;
58   planner masterPlanner must-not-use alternative strategy_1;
59   planner masterPlanner must-not-use alternative strategy_2;
60   executor parameterExecutor must-use executor slaveExecutor;
61   executor slaveExecutor must-use executor parameterExecutor;
62   executor slaveExecutor must-use effector wheels;
63   executor slaveExecutor must-use effector speed;
64   executor parameterExecutor must-not-use effector wheels;
65   executor parameterExecutor must-not-use effector speed;
66   sensor tachometer must-use measured-output distance;
67   sensor orientation must-use measured-output angularSpeed;
68   effector wheels must-use Servo-Controller;
69   effector speed must-use Servo-Controller;
70 }
```

Listing 3: Communication rules of the PA

Lines 43 − 47 specify all rules related with Monitors and their accesses. Lines 48 − 53 specify all rules related with Analyzers and their accesses. Lines 54 − 59 specify all rules related with Planners and their accesses. Lines 60 − 65 specify all rules related with Executors and their accesses and finally lines 66 − 69 specify all rules related with Sensors, Effectors and their accesses.

Table 1 presents all the rules allowed by the DSL that can be written by software architects. Abstractions of each column *must* or *must not* use abstractions of each row. Notice that they can be connected if they belong to the same level of abstraction. The only exceptions are Monitor to Sensor, Executor to Effector, Analyzer to ReferenceInput, Analyzer to Alternative, Planner to Alternative and Sensor to MeasuredOutput.

LM^1	L^2	M^3	A^4	P^5	E^6	K^7	S^8	
✓								**LM** (1. Loop Manager)
	✓							**L** (2. Loop)
		✓	✓	✓	✓	✓		**M** (3. Monitor)
		✓	✓	✓	✓	✓		**A** (4. Analyzer)
		✓	✓	✓	✓	✓		**P** (5. Planner)
		✓	✓	✓	✓	✓		**E** (6. Executor)
		✓	✓	✓	✓			**K** (7. Knowledge)
		✓						**S** (8. Sensor)
					✓			**EF** (9. Effector)
			✓					**RI** (10. Reference Input)
			✓	✓				**A** (11. Alternative)
							✓	**MO** (12. Measured Output)
✓: It means that an abstraction *must-use* or *must-not-use* other.								

Table 1: The rules allowed by the DSL

DSL-REMEDY implements twenty domain rules that can be activated or deactivated by software architects through a user interface provided by our plugin. These rules were obtained by means of the analysis of the MAPE-K reference model. For instance, planners must not use monitors and vice-versa. Table 2 presents the complete set of domain rules where $\longrightarrow$ represents the *must-use* accesses and $\not\longrightarrow$ represents the *must-not-use* accesses. When an AS does not conform its domain rules we will refer to this concept as domain drift. Notice that in Listing 3 it was not necessary specify domain rules in the DSL because they already were activated.

It is important to highlight that these rules are conceptual rules, that means they take into account direct and indirect dependencies. Direct dependencies occur when an abstraction uses another one without the intervention of a third one. In this case, the relationship is explicit in the CA. On the other hand, indirect dependencies occur when an abstraction use another one but with the intervention of a third one. Thus there is no explicit dependency in the CA, nevertheless REMEDY create the implicit dependencies in order to perform correctly the ACC.

Monitor	Analyzer	Planner	Executor	Knowledge	
	$\not\longrightarrow$	$\not\longrightarrow$	$\not\longrightarrow$	$\not\longrightarrow$	Monitor
$\longrightarrow$		$\not\longrightarrow$	$\not\longrightarrow$	$\not\longrightarrow$	Analyzer
$\not\longrightarrow$	$\longrightarrow$		$\not\longrightarrow$	$\not\longrightarrow$	Planner
$\not\longrightarrow$	$\not\longrightarrow$	$\longrightarrow$		$\not\longrightarrow$	Executor
$\longrightarrow$	$\longrightarrow$	$\longrightarrow$	$\longrightarrow$		Knowledge

Table 2: Domain rules of ASs

As we state before, the keyword *withDomainRules* must be declared in Loops to enable domain rules. Thus, the rules will affect all abstractions that match with a specific rule. For instance, if a Loop has two monitors and the rule *Monitor* $\not\longrightarrow$ *Planner* is activated then a constraint for each monitor will be generated to be checked in the CA. Despite the DSL can load domain rules they are not mandatory if a software architect writes a custom rule that violates a specific domain rule. Thus it is important the implementation of custom validators that implement additional constraint checks of a DSL, which cannot be done at parsing time.

DSL-REMEDY implements three types of validators. The first one checks that abstractions do not access themselves. For instance, the rule *monitor parameterMonitor must-use monitor parameterMonitor* is forbidden. Listing 4 shows an example of this validator where line 1 checks the monitor is not null, line 2 checks if a monitor is accessing itself and line 3 raise the error in the corresponding line where the rule was specified in the DSL. The same implementation is valid for the other abstractions so we do not include them in the document.

```
1  if (dslRuleMonitor.monitor2 !== null)
2    if (dslRuleMonitor.monitor == dslRuleMonitor.monitor2)
3      error("Check a monitor does not have dependency with itself",
          SasDslPackage.eINSTANCE.DSLRuleMonitor_Monitor2,
          DUPLICATE_MONITOR_ACCESS)
```

Listing 4: Checking if an abstraction depend on itself

The second one checks that rules can not be duplicated. In order to implement this validator a *Hash* data structure is created for each abstraction to disallow duplicate communication rules. In Listing 5, line 1 implements a for loop that iterates over the elements of the *Hash* structure. In line 3, if there are duplicates then line 5 raise errors in the corresponding lines where of duplicated rules.

```
1  for (entry:multiMapRuleMonitor2Monitor.asMap.entrySet) {
2      val duplicates = entry.value
3    if (duplicates.size > 1){
4      for (d:duplicates)
5        error("Duplicated rule",d, SasDslPackage.eINSTANCE.
          DSLRuleMonitor_Monitor2, DUPLICATE_RULES)
6    }
7  }
```

Listing 5: Checking if a communication rule is duplicated

The third one checks if communicating rules are violating domain constraints. Listing 6 shows an example of this kind of validator that checks if a monitor must use a planner. Line 1 get the domain rule written in the DSL and line 5 verifies if the domain rule is deactivated through the computational support. If the rule is presented but deactivated then a warning is raised.

```
1    var dslDomain = dslRuleMonitor.monitor.eContainer.eContents.filter(
          DSLDomainRule).toList
2    if (!dslDomain.isEmpty) {
3      val queryClass = new QueryClass(MainView.getDatabaseUrl())
4      val rule = queryClass.ruleIsActive("Monitor","Planner");
5      if (Boolean.valueOf(rule.get(1)))
6        if (dslRuleMonitor.planner !== null && dslRuleMonitor.access.equals("
          must-use"))
7          warning("The rule is violating the domain rule number  " + rule.get
          (0), SasDslPackage.eINSTANCE.DSLRuleMonitor_Planner)
8    }
```

Listing 6: Checking if a rule is violating a domain rule

4.1.4 *Technical Details.*

In this subsection we detailed the two outputs of DSL-REMEDY: The AS planned architecture and an OCL file. The Xtend/Xtext framework provides the mechanisms for the construction of templates that takes into account the grammar of the DSL. Once these templates are implemented, the code generation is straightforward. The PA of the AS is an instance of KDM metamodel which can represent several viewpoints of a system ranging from lower-abstractions, such as source code up to higher-abstractions such as architecture.

Listing 7 shows part of the PA as a KDM instance where line 2 declares the architectural model, line 3 declares the Managing *adaptationManager*, line 4 the Loop *loopManager* and line 6 the Monitor *parameterMonitor*.

The relationships among the architectural elements are represented by means of the *AggregatedRelationship* metaclass and they are addressed by the relationships among source code elements. Line 7 shows an example of it that enables the relationship of *parameterMonitor* with *slaveMonitor* and *masterAnalyzer*. This PA is used for a graphical visualization of the architecture, and to do so several *qvt-o* rules are applied on it to be transformed in a UML component diagram.

```
1    <kdm:Segment name="Planned Architecture">
2    <model xsi:type="structure:StructureModel" name="ArchitecturalView_">
3      <structureElement xsi:type="structure:Subsystem" name="
          adaptationManager" stereotype="/0/@extension.0/@stereotype.11">
4         <structureElement xsi:type="structure:Component" name="loopManager"
        stereotype="/0/@extension.0/@stereotype.7">
5             [...]
6             <structureElement xsi:type="structure:Component" name="
          parameterMonitor" stereotype="/0/@extension.0/@stereotype.0"
          outAggregated='//@model.1/@structureElement.0/@structureElement.0/
          @structureElement.0/@structureElement.0/@aggregated.0 [...] ' >
7               <aggregated from='//@model.1/@structureElement.0/
          @structureElement.0/@structureElement.0/@structureElement.0' to='//
          @model.1/@structureElement.0/@structureElement.0/@structureElement
          .0/@structureElement.4' relation='//@model.0/@codeElement.0/
          @codeElement.1/@actionRelation.0 [...]' density='6'/>
8      </structureElement>
9      ..
10   </model>
11   </kdm:Segment>
```

Listing 7: A PA Serialized as a Structure Package Instance

Listing 8 shows a brief snippet of the OCL file generated from the PA of Listings 1, 2 and 3 but with domain rules activated for *loop_2*. Due to space restrictions, it shows one rule per section because the whole file has more than 400 LoC. Notice that the OCL file is composed of four sections. The first one verifies the existence of the declared abstractions. The second one verifies the structural rules. The third one verifies the communication rules and the fourth one verifies domain rules that were activated by software architects.

```
1  package structure
2  -- Check the existence of AS abstractions --
3    context StructureModel
4    inv exist_parameterMonitor: Component.allInstances()->exists(c| c.name='
          parameterMonitor' and c.stereotype->asSequence()->first().name = '
          Monitor')
5  -- Check structural rules of AS --
6    context StructureModel
7    inv composite_parameterMonitor: Component.allInstances()->select(c| c.
          name='parameterMonitor' and c.stereotype->asSequence()->first().name
           = 'Monitor')->
8                    exists(d|d.oclContainer().oclAsType(Component).name='
          masterLoop' and d.oclContainer().oclAsType(Component).stereotype->
          asSequence()->first().name = 'Loop')
9  -- Check communication rules of AS --
10   context StructureModel
11   inv not_access_parameterMonitor_proximity: not AggregatedRelationship.
          allInstances()->exists(c| c.from.name='parameterMonitor' and c.to.
          name='proximity')
12 -- Domain rules --
13   context StructureModel
14   inv domain_not_access_slaveMonitor_slavePlanner: not
          AggregatedRelationship.allInstances()->exists(c| c.from.name='
          slaveMonitor' and c.to.name='slavePlanner')
15 endpackage
```

Listing 8: Snippet of OCL constraints file

The first rule (lines 3 − 4), verifies the existence of the Monitor *parameterMonitor*. Checks if element with name *parameterMonitor* exists and if the stereotype corresponds to a Monitor. The second rule (lines 6 − 8), verifies the composition of *parameterMonitor* in *masterLoop*. It checks if *parameterMonitor* is contained in *masterLoop*. The third rule (lines 10 − 11), verifies the forbidden access from *parameterMonitor* to *proximity*. It will check that there is no *AggregatedRelationship* between *parameterMonitor* and *proximity*. The fourth rule (lines 13 − 14), verifies the domain rule $Monitor \not\rightarrow Planner$.

5 EVALUATION

5.1 Scoping

One of the claimed benefits of using domain-specific ACC approaches is the improvement of productivity because some structural and communication rules from the domain are already known and do not need to be specified by the architects [32]. Nevertheless, at best of our knowledge, no controlled experiments have been conducted that provide evidence of improved productivity when software architects use domain-specific ACC approaches, particularly in the research area of ASs.

Hence, the contribution of this section is to present a controlled experiment that compares two different approaches/tools for ACC. The first is DCL-KDM [6] a generic ACC approach and the second is DSL-REMEDY, a domain-specific ACC approach for ASs. The goal of our experiment is put in the following statement:

Analyze the architecture specification of ASs with ACC tools †
for the purpose of evaluating two different tools
with respect to their productivity
from the point of view of researchers
in the context of final-year undergraduate students in computer engineering.
† We are referring to the adaptive part of an AS.

In this context, *productivity* relates to cost in time, quantity of errors and work effort required to specify the architecture of the adaptive part of an AS. The experiment was carried out in the context of a PhD study, involving students of the Computer Engineering Bachelor program in a prestigious university of Chile.

5.2 Setting

The experiment was performed in one week at the end of the second semester of 2020. The subjects of the experiment were 24 final-year undergraduate students that had been taken software engineering, domain-specific language and software architecture courses. The experiment consisted of three activities with four hours of work each day:
1. **A training session** was given covering the theoretical topics of AS, software architecture and DSLs and a more practical topic teaching REMEDY and DCL-KDM. Also, they filled a form that was used to profile the students for dividing them in two groups of 12 students each one. Some of the questions asked to them were about programming skills, industry experience and if they attended some courses of the bachelor's program related with the experiment;
2. **A pilot experiment** was conducted to familiarize with the artifacts used in the real experiment. Students signed a consent letter and the two groups perform two architectural specification of ASs by using REMEDY and DCL-KDM. With the provided information by the pilot, we analyzed if the given time to complete the activity was optimal, if students understood experiment instructions and if the tools were used correctly.
3. **The experiment** was then conducted with the same format as the pilot but different AS architecture specifications.

REMEDY can be downloaded from this repository https://tinyurl.com/y34jyeut while DCL-KDM from this one 10.5281/zenodo.5136838.

5.3 Planning

5.3.1 Experimental Design. Table 3 presents the experiment design. Students were separated in two balanced groups according to their profile and each group performed two different assignments for specifying ASs. Both assignments are hypothetical, specified in UML notation, but they had the same level of difficulty and were designed by taking into account well known patterns of AS [34]. Thus it involved the creation of AS abstractions and their communication rules.

Group	First Task	Second Task
G1	S-I (DCL-KDM)	S-II (REMEDY)
G2	S-II (REMEDY)	S-I (DCL-KDM)

Table 3: Experiment design

In the first task, Group 1 specified S-I by using DCL-KDM and Group 2 specified S-II by using REMEDY. In the second task, Group 1 specified S-II by using REMEDY and Group 1 specified S-I by using DCL-KDM.

Due to space restrictions we do not include the specification diagrams, dataset and R-statistics operations for statistical analysis but they are available in the following url: https://doi.org/10.5281/zenodo.4626751.

5.3.2 Hypotheses Formulation. The research goal of this experiment is to compare the use of DCL-KDM and REMEDY regarding productivity in terms of time to complete an AS specification and the number of errors made by subjects. Also, we want to know if there are differences of perception of effort when subjects perform architectural specifications with both tools. Therefore, the research goal can be refined in 3 sub-goals that map to a set of hypotheses. In particular, each sub-goal maps to a null hypothesis to be tested, and an alternative hypothesis in favor and to be accepted if the null hypothesis is rejected. We formulate 3 null hypotheses (H_0) and three alternative hypotheses (H_α):

- H_{01}: There is no difference in time to complete an architectural specification of AS by using DCL-KDM or REMEDY.

$$H_{01} : \mu_{time_{DCL-KDM}} = \mu_{time_{REMEDY}} \quad (1)$$
$$H_{\alpha 1} : \mu_{time_{DCL-KDM}} > \mu_{time_{REMEDY}} \quad (2)$$

- H_{02}: There is no difference on errors when specifying the architecture of an AS with DCL-KDM or REMEDY.

$$H_{02} : \mu_{errors_{DCL-KDM}} = \mu_{errors_{REMEDY}} \quad (3)$$
$$H_{\alpha 2} : \mu_{errors_{DCL-KDM}} > \mu_{errors_{REMEDY}} \quad (4)$$

- H_{03}: There is no difference on effort when specifying the architecture of an AS with DCL-KDM or REMEDY.

$$H_{03} : \mu_{effort_{DCL-KDM}} = \mu_{effort_{REMEDY}} \quad (5)$$
$$H_{\alpha 3} : \mu_{effort_{DCL-KDM}} > \mu_{effort_{REMEDY}} \quad (6)$$

5.3.3 Independent and Dependent Variables. Each hypothesis requires the definition of a set of independent and dependent variables, and a selection of proper metrics to measure the dependent variables.
• *Independent Variables:* Independent variables are variables in the experiment that can be manipulated and controlled. In our experiment, there are two independent variables:

- Techniques: The treatment used by a subject to solve an assignment. This variable is the factor of the experiment that is changed to observe the effect on the dependent variables. The two possible values of this factor are DCL-KDM and REMEDY;
- Specifications: The problem to be solved by the subject (specifications S-I and S-II). Since the specifications have the same level of difficulty, the specification is not considered as a factor but as a fixed variable.

• *Dependent Variables:* Dependent variables are variables that we want to study to see the effect of different treatments. For each hypothesis, we defined the corresponding dependent variables:
- Time: The time in minutes to complete an architectural specification of AS;
- Errors: Number of errors found after finish the architectural specification;
- Effort: Likert-type scale from 1 to 4 that denotes the perception of effort of subjects, where 1 means easy to use and 4 very difficult.

5.4 Analysis & Discussion

5.4.1 Analysis. In total, 24 subjects provided usable data for paired comparison of time, errors and effort. Table 4 depicts the mean and standard deviation for the factor Tool on Time on Error and on Effort. The values show that there is a difference between the means but to know if they are sufficiently different we applied an analysis of variance test.

	Time		Error		Effort	
Tool	**mean**	**sd**	**mean**	**sd**	**mean**	**sd**
DCL-KDM	64.75	18.43024	4.791	3.106	2.83	0.637
REMEDY	45.75	16.67920	1.291	1.122	1.70	0.624

Table 4: Mean and Sd of Time, Error and Effort

As we have 1 factor with 2 levels within-subjects (repeated measures), we applied the paired sample t-test. Also, in order to mitigate carryover effects which introduces biases in the results we used a full counterbalancing strategy, as is depicted in Table 3. After applying a t-test on Orders, we got a p-value of 0.9174 that suggests our results do not have an order effect where order itself could cause differences of performance. The Shapiro-Wilk normality test of residuals for subjects and subjects tools were 0.1119 and 0.06314 respectively indicating normality. The t-test paired samples on Tools gave us a p-value of 0.00088 that means there is a significant difference between both approaches.

Errors are a count response and often do not satisfy the assumption of normality for anovas. In our case the error data for DCL-KDM fits on a Poisson distribution with a p-value of $6.890398e-06$ by using the goodness–of–fit tests. On the other hand, errors of REMEDY did not fit so we chose to apply a non-parametric analysis, the wilcoxon signed-rank test. Thus the p-value after applying wilcoxon signed-rank test was $5.722e^{-6}$ that means there is significant differences on Errors by using DCL-KDM or REMEDY.

Effort is an ordinal likert-scale response from 1 to 4 and this psychometric scale rarely satisfies the conditions for anova so we used again the non-parametric analysis wilcoxon signed-rank test. The p-value was $9.537e-7$ which indicates that there are differences on Effort when subjects use DCL-KDM or REMEDY. Table 4 shows the mean and standard deviation for the factor Tool on Effort. On average, the perception of subjects is that DCL-KDM is harder to use than REMEDY for specifying the architecture of ASs.

Therefore, based on the statistical analysis every null hypothesis (H_0) is rejected with a significance level (α) of 0.05.

5.4.2 Discussion. The descriptive analysis shows that there is a clear improvement for the dependent variables when subjects use REMEDY compared to DCL-KDM. This is confirmed by the statistical tests. On average, time is about 30% lower with REMEDY compared to DCL-KDM. Closer examination of the specifications performed with REMEDY reveals that subjects made use of predefined domain-specific rules support when needed so this could explain the saving time. Also, subjects that performed the specifications with DCL-KDM wrote some abstractions with different types. The solutions with both tools can be obtained in the following 10.5281/zenodo.5136838.

For instance, DCL-KDM allows three types of abstractions; subsystem, layer and component, where subsystem and layer can be composed of subsystems, layers and components. Thus the choice decision among component, layer and subsystem for an AS abstraction could affect the productivity at time to do the specification.

With regard to errors, Figure 4 shows a bar chart with the number of errors made by subjects. When subjects used DCL-KDM the total number of errors was 115, where 46 correspond to structural rules (SR) and 69 to communication rules (CR). On the other hand, when subjects used REMEDY the total number of errors was 31 and all of them correspond to communication rules.

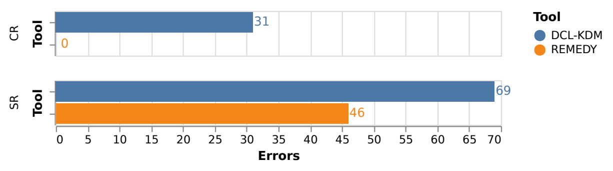

Figure 4: Errors found with both tools

The main characteristic of DCL-KDM is the capability of specifying compound abstractions deeply. For instance, a subsystem can hold layers that in turn can hold other subsystems and layers that in turn can hold components. On the other hand, the composition of abstractions in REMEDY is well-known due to AS domain restrictions. Therefore in copy paste operations or when the specification contains several lines of code that makes difficult to read and understand it, subjects are more likely to make mistakes when specifying structural rules. Moreover, we found different types of error of communication rules in DCL-KDM such as duplicate rules and circular rules due to lack of validation.

Regarding errors made with REMEDY all of them are related to communication rules of low-level abstractions. A possible answer of why this happens is because subjects made copy and paste of the communication rules where were involved reference input abstractions of different control loops. Indeed, this was a bug because when the rule that allows access between two control loops was absent, REMEDY did not validate accesses of abstractions of one control loop to another with low-level abstractions. The bug was corrected in the next version of REMEDY.

Finally, concerning effort the overall score was average for REMEDY and difficult for DCL-KDM. We believe the results are strongly influenced by the type of DSL. REMEDY uses an appropriate language for the AS domain and the way it constructs the structural rules is identical to the MAPE-K reference model which contrast with DCL-KDM. Although, the results could be predictable we were able to demonstrate it scientifically.

5.5 Threats of Validity

5.5.1 Threats to Internal Validity. Internal validity is the extent to which independent variables are responsible for the effects seen to the dependent variables. Due to global pandemic of SARS-CoV-2, the controlled experiment was carried out online and as a consequence the instructor could not verify in situ if there was any kind of interaction among subjects about how to specify the architectures.

To reduce this threat, before experiment activities begin we explained to them that it was not a competition so there was not any kind of rewards for finishing in less time or having an optimal architecture. After having analyzed their specifications (pilot and experiment) we determine that exists heterogeneity in architecture specifications, so if there was any type of interaction among subjects it did not affect the experiment.

5.5.2 Threats to Construct Validity. Construct validity is the degree to which the operationalization of measures in the study represent the constructs in the real world. We have seen one type of such threats; Inadequate preoperational explication of constructs. Although researchers delivered all concepts involved in the experiment to subjects, maybe the depth of contents have not been optimal for the understanding of some subjects, this was due to some restrictions of the number of online sessions and the time duration of each online session.

To reduce this threat, a subject profile was performed and contrasted with the academic history of each participant. Moreover, a researcher of this study was in charge of at least one course that subjects needed to attend to participate in this experiment so in a certain way we already known subjects competencies.

5.5.3 Threats to External Validity and Conclusion Validity. External validity is the degree to which findings of a study can be generalized to other subject populations and settings. Conclusion validity concerns generalizing the result of the experiment to the concept or theory behind the experiment. Due to practical restrictions, we deal with students of an undergraduate program in Computer Engineering as subjects for our study. Although students do not represent expert software engineers, they are the next generation of software professionals [14]. Also, it is possible the specifications does not exists in real world applications. To mitigate this threat, the architectural specifications were designed considering patterns based on influential papers of the area.

Finally, there is a threat concerning the reliability of time measure. We have asked the subjects to set the starting and ending time. In this sense we could have had a problem because a subject could forget to mark the time. To mitigate this, we use an online program to chronometer the time. Each subject shared the chronometer so the instructor was aware of the time spent.

6 RELATED WORKS

In literature we found two ways to specify PAs. The first one is by using non-extensible/generic approaches and the second one extensible/domain-specific approaches. The main characteristic of generic approaches is the use of a generic vocabulary to describe architectural abstractions such as entity, layer, module and subsystem. Moreover, some techniques use vocabulary that relies on source-code concepts such as packages and classes to denote components or modules for specifying the intended or planned architecture [3, 8]. One of the first approaches developed for detecting architectural violations is reflexion models [9]. This technique compare two models; a high-level model which contains entities and relations between these entities and a low-level model commonly created from source code and represented as a call graph.

On the other hand, there are solutions that uses a textual notation that are capable of checking rules specified in a dedicated DSL. *DCL 2.0* [27], *DCL-KDM* [6], *TamDera* [8] and InCode.Rules [18] are designed to define constraints on code dependencies (e.g., accesses, declarations, extensions). Textual notation tools are characterized by high usability and a well defined strict specification language. The authors of DCL 2.0 claim that their language is more usable than other logic inspired alternatives. Those are supposedly based on a more complex and heavyweight notation and offer poor performance. Other researchers recognize the difficulty that typical users encounter when approaching solutions that require a basic understanding of logic programming [16]. Besides the mentioned approaches, architectural rules can be translated into a language based on first order logic. The main advantage of this type of approaches is that they are inherently extensible within the boundaries set by the underlying language model. Users can define new concepts by declaring and combining facts and predicates. This form of extensibility allows developers to adapt the notation to the specific vocabulary required to describe their architecture. Nevertheless, according to [4] the usability is compromised because these kind of languages involves programming capabilities which typically go beyond the skills possessed by average software engineers.

Languages like *SOUL* [20], *LogEn* [7] and SCL [10] are examples of solutions that can be used for conformance checking. *SOUL* is a Prolog-inspired internal DSL implemented in Smalltalk. A set of predefined high-level predicates can be used to create architectural rules or define new predicates. Pre-defined predicates are evaluated using dedicated analyzers. The representation of the target architecture can be enriched by adding new facts to the fact base. Although we do not find in literature approaches/techniques of ACC used in AS domain, it is clear that non-extensible and extensible approaches can be used for that purpose. Regarding to non-extensible approaches they use generic terms to specify the system architecture where some of them use concepts near to source-code (low-level of abstractions) and others use concepts such as subsystems, components and layers (high-level of abstraction). Also, the majority of them perform the conformance checking in systems developed in Java.

7 CONCLUSION

We have presented DSL-REMEDY, a language to specify Planned Architectures of Adaptive Systems. There are two moments in which our DSL can be employed. The first in in early stages of development for creating the adaptive part of the architecture of a new system. The second is in later states, for existing systems. This happens when architects need to verify whether the adaptive modules were designed conforming to some expected architectural rules. In this case, an architecture specification may even not exist.

As our approach is domain-specific, software architects do not need to create rules for checking dependencies stated by MAPE-K, just inform which of them need to be checked. Although generic ACC approaches have the advantage of being applied in a vast set of systems, there some points in favor of domain-dependent ACC approaches: *i*) ACC domain-independent used for specifying the PA is not able to incorporate domain-specific rules, forcing architects to write rules which are very common and canonical of the domain. This waste of time could be avoided; *ii*) Considering that the architects know the ASs domain, they can be much more precise using a language which delivers the canonical abstractions of the domain, as they already know the behavior of them.

A specific point regarding our work, that differs from other works, is the use of an existing reference model as base for the specification of planned architectures. Usually, ACC approaches start the process by creating a planned architecture from scratch, i.e., the architects are totally free for specifying the planned architecture. Although our DSL also allows this freedom, it has the ability for turning on and off the rules of the adaptive domain. Therefore, the reference model of the domain embedded in the DSL. Yet another point is that the validators enhance the usability of our DSL. They reduces the possibility of software architects make mistakes in the architecture specification. Another positive point is that our DSL is capable of generating two separate artifacts; one for check the constraints and another one to visualize the PA architecture.

Our controlled experiment demonstrated that REMEDY improves the productivity when specifying PA because it saves time to architects and decreases the number of errors. Moreover, REMEDY is capable of identifying drifts that involve low-level abstractions that are not evident in MAPE-K. We are currently working on extending our DSL so that it can specify not only adaptive parts, but also the conventional and canonical parts of the whole system.

ACKNOWLEDGMENTS

This work was funded by the National Agency for Research and Development (ANID) PFCHA/DOCTORADO BECAS CHILE/2016 – 72170024. We also would like to thank FAPESP (p.n. 2016/03104-0).

REFERENCES

[1] Imen Abdennadher, et al. 2017. A Design Guideline for Adaptation Decisions in the Autonomic Loop. *Procedia Computer Science* 112, C (Sept. 2017), 270–277.
[2] Yuriy Brun, et al. 2009. *Engineering Self-Adaptive Systems through Feedback Loops.* Springer, Berlin, Heidelberg, 48–70.
[3] João Brunet, et al. 2012. On the Evolutionary Nature of Architectural Violations. In *2012 19th Working Conference on Reverse Engineering*. 257–266.
[4] Andrea Caracciolo, et al. 2015. A Unified Approach to Architecture Conformance Checking. In *2015 12th Working IEEE/IFIP Conference on Software Architecture.* 41–50.
[5] Betty H. C. Cheng, et al. 2009. *Software Engineering for Self-Adaptive Systems: A Research Roadmap.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1–26.
[6] A. d. S. Landi, et al. 2017. Supporting the Specification and Serialization of Planned Architectures in Architecture-Driven Modernization Context. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 327–336.
[7] M. Eichberg, et al. 2008. Defining and continuous checking of structural program dependencies. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 391–400.
[8] Alessandro Gurgel, et al. 2014. Blending and Reusing Rules for Architectural Degradation Prevention. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14).* Association for Computing Machinery, New York, NY, USA, 61–72.
[9] R. C. Holt, et al. 2000. Architectural Repair of Open Source Software. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension.* IEEE Computer Society, Los Alamitos, CA, USA, 48.
[10] D. Hou et al. 2006. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering* 32, 6 (2006), 404–423. https://doi.org/10.1109/TSE.2006.60
[11] IBM. 2005. *An Architectural Blueprint for Autonomic Computing.* Technical Report. IBM.
[12] P. Jamshidi, et al. 2013. A Framework for Classifying and Comparing Architecture-centric Software Evolution Research. In *2013 17th European Conference on Software Maintenance and Reengineering.* IEEE, 305–314.
[13] J. O. Kephart et al. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan. 2003), 41–50.
[14] B. A. Kitchenham, et al. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28, 8 (2002), 721–734.
[15] Jens Knodel et al. 2007. A Comparison of Static Architecture Compliance Checking Approaches. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07).* IEEE, 12.
[16] Angela Lozano, et al. 2015. Usage contracts: Offering immediate feedback on violations of structural source-code regularities. *Science of Computer Programming* 105 (2015), 73–91.
[17] C. Maffort, et al. 2013. Heuristics for discovering architectural violations. In *2013 20th Working Conference on Reverse Engineering (WCRE).* IEEE Computer Society, 222–231.
[18] R. Marinescu et al. 2010. inCode.Rules: An agile approach for defining and checking architectural constraints. In *Proceedings of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing.* IEEE, 305–312.
[19] D. S. Martín, et al. 2020. Characterizing Architectural Drifts of Adaptive Systems. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 389–399.
[20] K. Mens, et al. 1999. Declaratively codifying software architectures using virtual software classifications. In *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275).* 33–45.
[21] Gail C. Murphy, et al. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-level Models. *ACM SIGSOFT Software Engineering Notes. Software Engineering Notes* 20, 4 (Oct. 1995), 18–28.
[22] Hausi Müller et al. 2014. *Runtime Evolution of Highly Dynamic Software.* Springer Berlin Heidelberg, 229–264.
[23] Leonardo Passos, et al. 2010. Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software* 27, 5 (2010), 82–89.
[24] Dewayne E. Perry et al. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes. Software Engineering Notes* 17, 4 (Oct. 1992), 40–52.
[25] Leo Pruijt, et al. 2013. Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support. In *2013 IEEE International Conference on Software Maintenance.* IEEE, 220–229.
[26] Leo Pruijt, et al. 2017. The accuracy of dependency analysis in static architecture compliance checking. *Software: Practice and Experience* 47, 2 (2017), 273–309.
[27] Henrique Rocha, et al. 2017. DCL 2.0: modular and reusable specification of architectural constraints. *Journal of the Brazilian Computer Society* 23, 1 (16 08 2017), 12.
[28] Jacek Rosik, et al. 2011. Assessing Architectural Drift in Commercial Software Development: A Case Study. *Softw. Pract. Exper.* 41, 1 (Jan. 2011), 63–86.
[29] M. A. Serikawa, et al. 2016. Towards the Characterization of Monitor Smells in Adaptive Systems. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS).* IEEE, 51–60.
[30] Ricardo Terra et al. 2009. A Dependency Constraint Language to Manage Object-oriented Software Architectures. *Softw. Pract. Exper.* 39, 12 (Aug. 2009), 24.
[31] William M. Ulrich et al. 2010. *Information Systems Transformation: Architecture-Driven Modernization Case Studies.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[32] Perla Velasco-Elizondo, et al. 2018. Towards Detecting MVC Architectural Smells. In *Trends and Applications in Software Engineering*, Jezreel Mejia, et al. (Eds.). Springer International Publishing, 251–260.
[33] Norha M. Villegas, et al. 2011. A Framework for Evaluating Quality-driven Self-adaptive Software Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '11).* ACM, 80–89.
[34] Danny Weyns, et al. 2013. *On Patterns for Decentralized Control in Self-Adaptive Systems.* Springer Berlin Heidelberg, 76–107.