MeidyaCoM-policy: Approach for modelling and checking repair policies for self-healing systems

Mohamed Hadj Kacem | Imen Tounsi | Ahmed Hadj Kacem

ReDCAD laboratory, University of Sfax, Sfax, Tunisia

Correspondence
Imen Tounsi, ReDCAD laboratory, University of Sfax, Sfax 3000, Tunisia. Email: Imen.Tounsi@redcad.org

Abstract
The architecture of distributed systems is subject to certain failures: component failure, downed connections etc. These failures come from the dynamicity and complexity of these systems. As a solution to cure this weakness, adaptation plans can be added. However, the main difficulty of self-adaptation emerges while considering the soundness of the adaptation and preservation of stylistic constraints of the system. The MeidyaCoM-Policy, an architecture-centric approach for modelling and checking repair policies to guarantee the success of the adaptation, is presented. This approach is a solution for applying self-healing. A new unified modelling language (UML) profile that provides a visual notation for modelling repair policies is proposed to automatically transform UML models to Z notation using transformation rules coded in eXtensible styles language transformation (XSLT) language. These specifications are implemented under the Z/EVES theorem prover to prove specification soundness and consistency to guarantee the success of the adaptation. MeidyaCoM-Policy applies to all architectural styles. It is instantiated for the publish/subscribe style and illustrated with a case study. An outline of the developed software environment is provided.

1 | INTRODUCTION

The architectures of current software systems have become increasingly large, dynamic, and complex. The proper functioning of these systems requires properties related to adaptability, reusability, and mobility. Self-adaptability has been proposed as an efficient approach to reduce the complexity associated with managing the architecture of these systems. It endows a software system with the ability to adapt in runtime to deal with unexpected errors (connections going down, component failure etc.) or surrounding changes. Because adapting the architecture of these systems is difficult, it must not be ad hoc; adaptation plans must guide it. Our research work comes within this context and consists of proposing MeidyaCoM-Policy, an architecture-centric approach for modelling and checking repair policies for self-healing systems.

MeidyaCoM-Policy is an extension of the MeidyaCoM project [1, 2]. This project supports the modelling of correct by design publish/subscribe architectural style and the definition of the basic reconfiguration operations. MeidyaCoM project does not allow the self-adaptation of software architecture. It serves as the basis for the adaptation. Repair policies use reconfiguration operations for adaptation to the architectural style. A recent branch of work has demonstrated an interest in using architectural style as a basis for runtime monitoring, error detection, and repair [3, 4] where errors correspond to architectural constraint violation that triggers the appropriate repair policy. Architectural style was usually defined as a set of component types and connections with a definition of invariant properties and a description of how elements can be combined. One of the key issues in making this adaptation successful is ensuring the conformance of the new system state (obtained after reconfiguration) to the defined style. By conformance, we mean a system that satisfies all of the constraints defined by the style. MeidyaCoM-Policy applies to all architectural styles, and it is instantiated for the publish/subscribe style.

The MeidyaCoM-Policy approach is based principally on three contributions. The first contribution consists of modelling repair policies allowing the architectural adaptation. For this modelling, we propose a unified modelling language (UML)-based approach by introducing a new UML profile (a first version of the profile was presented in [5]). The profile assists

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2021 The Authors. IET Software published by John Wiley & Sons Ltd on behalf of The Institution of Engineering and Technology.
design activities for describing repair policies using a visual notation. This part of our approach provides several advantages. First, the modelling process is defined in a high level of abstraction, providing a generic and reusable model. Second, our approach seeks to take advantage of the expressive power of standard visual notations provided by the semi-formal UML 2.0 language that makes the understanding of policy models easy. Third, we implemented a software environment that supports the different features of this approach, and we integrated it as a plug-in in the open-source Eclipse framework.

The second contribution consists of automatically generating policy specifications. We implemented a rule-based generator that automatically translates the repair policy model specified using our UML profile into Z specifications, so we generate the Z specification of the repair policies. The automatic transformation to Z is the main advantage of this contribution. Thanks to its set theory and predicate logic, Z notation [6] is well suited to specify repair policies. For example, Z notation allows us to add pre- and post-conditions in the schema. In our work, we use preconditions to guard the policy. They prohibit the execution of the policy if they are not satisfied. In addition, Z notation allows the use of conditional expressions (if Predicate then Expression, else Expression); this cooperates very well with the form of policy (if condition then action1, else action2). Additionally, MeidyaCoM-Policy is a part of the MeidyaCoM project that uses the Z notation as a basic language. So logically, we work with this notation because it allows us to reach our objectives.

Finally, the third contribution is based on formal methods. Using theorem-proving tools supporting Z, such as Z/EVES [7], we can prove the soundness (i.e. every execution of the policy will place the system in a correct state w.r.t. the original style) and consistency (i.e. absence of any contradiction between architectural invariants) of the generated repair policies [8].

Z and UML are powerful modelling languages for describing and modelling complex systems. UML is an informal graphical language, whereas Z is a formal specification language. Specifications in Z are abstract and are based on the mathematical definition. UML notations are icons and graphics. Specifications in UML are easy to understand but have the potential to be ambiguous. In addition to the theoretical comparison, the practical experience revealed that UML provided a good approach to modelling the main concepts using activity diagrams, and component diagrams, while Z provided a precise model for the specification of the problem. Z and UML appear to be complementary. Combining Z and UML in different phases and aspects is the recommended approach. Thanks to this combination, our approach MeidyaCoM-Policy allows early modelling of the architectural software reparation during application design, which improves the quality of the application by enabling early analysis and fault detection. This quality improvement is confirmed by the evaluation results obtained from our approach and the associated tool. In this evaluation, we focussed on two evaluation approaches—the first is based on case study analysis, and the second is based on logical reasoning. More precisely, we used three evaluation criteria: generality, simplicity, and usability.

The remainder of the paper is organized as follows. In Section 2, we provide some background information on the MeidyaCoM project, we present the case study we will use in the rest of the paper to illustrate the different steps of our approach, and we provide some background information on the Z notation. In Section 3, we present our approach MeidyaCoM-Policy to define a repair policy profile and the translation into Z language, and then we show how we can prove the soundness property. In Section 4, we present the Eclipse plug-in implementing MeidyaCoM-Policy. In Section 5, we provide an evaluation of our approach. In Section 6, we discuss the related work. We examine several studies done on the description of self-healing capabilities of software architectures. We also examine works that use the UML modelling and works that combine UML notation with a formal language like Z. Ultimately, in Section 7, we present conclusions and future works.

2 | BACKGROUND

2.1 | MeidyaCoM

MeidyaCoM-Policy1, the main contribution of this paper, extends the MeidyaCoM project. MeidyaCoM is a style-based approach for modelling and formal specification of software architectures. The principal steps of the MeidyaCoM project are described as follows:

- Style design: This step allows the correct modelling of a publish/subscribe architectural style following the P/S-CoM approach [2]. P/S-CoM is an MDA-based modelling process comprising two steps. The first step is the UML modelling of the publish/subscribe architectural style. The second step is a formal specification of the architectural style with the Z notation.
- Reconfiguration operations: This step allows model reconfiguration operations following the MEIDYA approach. MEIDYA [1, 9] is an ‘end-to-end’ approach that supports dynamic reconfiguration of software architectures taking advantage of graphical modelling, formal methods, and aspect-oriented programming. MEIDYA is an MDA-based modelling process. The main objective is to model the architectural adaptation by expressing the conformance to a given architectural style proposed by the style design step. MEIDYA proposes a UML-based approach for describing dynamic software architectures using a new UML profile.
- Correctness and consistency proof: A proof process is elaborated to ensure that any constructed specification corresponding to a designed style following P/S-CoM is consistent. A verification process is also elaborated to verify that a given reconfiguration operation makes the system evolve correctly (w.r.t. its architectural style).

1The home page of our approach is http://www.redcad.org/members/hadjkacemm/tools/MeidyaComPolicy/index.htm
2.2 Case study

'Follow me' [10] is a concrete example that we use to illustrate the Z notation, MeidyaCoM approach, and our approach. It is an audio application that is aware of the user’s location and any speakers around the room. Once detected in a room (lobby, reception, restaurant,…), the client’s PDA sends a signal to its access point (manager). Following this signal, a sink will be informed that it must produce the music. This will be done at an access point (manager) that will cooperate with all other managers in turn to spread the music to the client as with progression into space.

The first step of the MeidyaCoM approach (style design) allows modeling this application at a high level of abstraction using the event-based publish/subscribe architecture style. This style emerges for distributed applications on a large scale. It is based on three types of components: a producer that generates the event, a consumer that consumes it, and an event service that ensures the exchange of the event between the producer and the consumer. This style provides a full decoupling of entities in terms of time and space [11]. It is provided by the event service that can be centralized and implemented as a single entity (dispatcher) or distributed and implemented as a network of dispatchers. Dispatchers cooperate to route information from producers to subscribed entities. The publish/subscribe style is marked by a variety of interconnection topologies governing the dispatchers. It can be in hierarchical, acyclic P2P, and general P2P form [12].

In the case study, the architectural style designed using UML 2.0-based notation is illustrated in Figure 1 and represents a publish/subscribe style architecture. The propagation of events between participants is mediated through a network of event dispatchers called Managers. In the case study, the interconnection topology used is the general P2P form. Both Client and Sink are associated with producer-consumer components. Communication links are modelled by links. For a link, events are propagated from the producer component to the consumer component, so we use the syntax PushSourceDestination. For instance, events in the PushMC link are propagated from the Manager to the Client.

2.3 Z notation

Z is a formal specification language that is widely used for providing formal semantics and verifications in various application domains. Z notation [6], based on set theory and first-order predicate logic, can be split in two: a notation for discrete mathematics and a notation for describing and combining schemas, called the schema calculus. Appendix 7 summarizes the complete syntax of Z modelling language grammar used in the paper. The mathematical language is used to describe objects and the relationships between them. The schema calculus is an important aspect of the Z notation because it makes Z suitable for describing large software systems and their properties. The schema can be used to describe the state of a system and how that state may change. A Z schema comprises two parts: a declarative part for the declaration of variables that characterize the schema and a predicative part for constraining their values.

Two forms can be used to introduce a new Z Schema: the vertical and horizontal forms. In the vertical form, we use a box, and in the horizontal form, we use the writing operator ≡. The vertical form is equivalent to the horizontal form.

\[
\text{Z Schema} \equiv [\text{DeclarativePart}] \land [\text{PredicativePart}]
\]

We use the horizontal form to introduce composed schemas, for example, ActionPlan ≡ Action_1 \& Action_2.

State Schema: A state schema is used to declare various system variables and constraints that concern these variables. We used the state schema in the style design. As an example, we present the state schema follow_ me. It allows specifying the architectural style of the application. In its declarative part, we specify the involved component types as well as their relations. Cl represent a finite set (F) of Clients, S represent a finite set (F) of Sinks and M represent a finite set (F) of Managers. Links are specified using relations. For example, PushCM is a relation between a client and a manager (Client ↔ Manager). In the predicative part, the first predicate [P1] states that the communication between a client and a manager must be bidirectional. The predicate [P2] states that the communication between a manager and a sink must be also bidirectional. With the predicates [P3], [P4], and [P5], we specify the domain (source) and the range (destination) of the links (PushCM, PushMS, and PushMM, respectively). In our work, we enhance the architectural style with QoS properties. The predicate [P6] states that the latency between any two managers should not exceed a predefined threshold (maxLatency):
Operation Schema: An operation schema is used to describe changes in the system state or to extract data from the system. To describe an operation upon a state, we use two copies of the state: the first represents the state before the operation, and the second represents the state afterwards:

\[
\Delta \text{Schema}
\]

\[
in_1?, \ldots, in_n? : \text{INPUT}
\]

\[
out_1!, \ldots, out_m! : \text{OUTPUT}
\]

We used the operation schema in the specification of the reconfiguration operations and the repair policies.

Schematic decorations: The presence of decorations in a schema indicates its nature.

- **Postfixed decorations**: Let \( n \) be a member of a schema. We note

- Prefixixed decorations: Let \( \text{Schema} \) be a given schema that describes the state of a system. In the declarative part \( \Delta \text{Schema} \) (delta Schema), we note a schema that includes two schemas: \( \text{Schema} \) and \( \text{Schema}^* \) (the state before and after the reconfiguration, respectively).

Schemas can be considered units we can manipulate and combine by means of schema operations like sequential composition operators represented by \( \circ \) or the piping operator represented by \( \Rightarrow \). Schema composition is designed for use with operation schemas. For each \( x \), a component of \( \text{Op1} \), and \( x \) itself, is a component of \( \text{Op2} \); the types of these two components must be the same. We call \( x \) a matching state variable. In addition, the types of any other components they share (including inputs, outputs, and state variables that do not match) must be the same. The schema \( \text{Op1} \circ \text{Op2} \) has all the components of \( \text{Op1} \) and \( \text{Op2} \) except for the components \( x \) of \( \text{Op1} \) and \( x \) of \( \text{Op2} \). Schema piping is designed for use with operation schemas that have an almost independent effect on two disjoint sets of state variables. In \( \text{Op1} \Rightarrow \text{Op2} \), the outputs of \( \text{Op1} \) (i.e., the components decorated with \( ! \)) are matched with the inputs of \( \text{Op2} \) (decorated with \( ? \)) and hidden, but the other components are merged as they would be in \( \text{Op1} \land \text{Op2} \).

An example is presented in Section 3.2.5.

Relations: A relation is used to describe ways in which elements of two distinct sets are related.

- **Ordered pair**: If \( A \) and \( B \) are two distinct elements, we call ordered pair \( a \) to \( b \), the pair having the first element \( a \) (start element) and the last element \( b \) (arrival element). We note that \( a \mapsto b \) or \( (a, b) \).

- **Relation**: A relation is defined as a set of ordered pairs, a subset of a cartesian product. If \( A \) and \( B \) are two sets, then \( R: A \leftrightarrow B \) denotes the relation between \( A \) and \( B \). The domain of \( R \) is the set of elements in \( A \) related to something in \( B \): \( \text{dom } R \). The range of \( R \) is the set of elements of \( B \) to which some element of \( A \) is related: \( \text{ran } R \). We also say that \( A \) and \( B \) are the source and target sets of \( R \), respectively.

- **Domain restriction**: The domain restriction \( (S \triangleleft R) \) of a relation \( R \) to a set \( S \) relates \( x \) to \( y \) only if \( R \) relates \( x \) to \( y \) and \( x \) is a member of \( S \).

- **Number of members**: To count the number of members of a set \( S \), we use the symbol \#. If \( n \) is the number of members of \( S \), then \#(\( S \)) = \( n \).

3 | MEIDYACOM-POLICY APPROACH

The main challenge of MeidyaCoM is building an adaptation mechanism to endow software systems with adaptation ability. It is an important stage in the development of adaptable systems allowing the modelling of a correctly designed architectural style and basic reconfiguration operations. The main contribution of MeidyaCoM-Policy, compared with MeidyaCoM, is to make adaptive systems that can adjust themselves when the adaptation needs to be performed so as to obtain self-adaptive systems. This contribution is achieved by proposing repair policies. It allows the application of reconfiguration operations with respect to the architectural style. The reason behind using the architectural style is to verify that the evolution of the architecture preserves the architectural invariants. This is very important for self-adaptive systems to maintain a correct software architecture and to guarantee that the repairs ensure the success of the adaptation. In our work, we improved the architectural style by \( QoS \) properties because many reasons behind the failures of software architecture come from performance properties. Here, we mainly focus on three properties: the dispatcher load, the latency, and the bandwidth between two dispatchers. These properties are mentioned as examples.

MeidyaCoM-Policy supports the modelling of repair policies, the automatic transformation of policy models to \( Z \) specifications, and the formal verification of their soundness and consistency. The modelling solution that we provide guides and assists design activities for describing repair policies using a visual notation based on the graphical UML notation.

3.1 | Repair policy profile

We propose a UML profile to describe repair policies. Three main reasons lead to the use of UML for modelling repair policies. UML profiles are mechanisms defined by OMG to structure UML extensions. Thanks to the annotations (tagged-values, stereotypes, constraints), UML profiles allow the definition of a vocabulary for extending UML to a specific area. In addition, the notations used in UML 2.0, and especially in the activity diagram, allow integration of the repair policy concepts and description of the E-C-A (event, condition, action) form explicitly. It is also important to mention that the profiles proposed by the MeidyaCoM project are based on UML 2.0.

The proposed profile, as illustrated in Figure 2, defines a set of concepts that can be used for modelling the structure of a repair policy following our approach. It extends the activity diagram of UML 2.0 [13]. It is composed of a set of meta-
classes and stereotypes. Table 1 describes the different proposed concepts in the policy profile.

To instantiate the policy profile, we propose a general repair policy model. As described in Figure 3, the RepairPolicy has a name (Policy) and a SkewedStyle (Sk_Style). After the detection of a triggering event, some conditions are tested to execute the appropriate Solution. In Solution compartments, we specify the ActionPlan, and in ActionPlan compartments, we specify Actions and their connections. All solutions converge with the join connector to form a new architecture (NewArchitecture).

3.1.1 | Case study: policy modelling

Let us apply our approach to the case study and give an example of a repair policy. The policy model is illustrated in Figure 4 and follows the UML profile in Figure 2. The name of the Policy is Policy, and the name of the SkewedStyle is Sk_follow_me. From an Initial state and with a trigger event, the repair policy is triggered. It is triggered when an event of high latency between two Managers (mg1? and mg2?) is detected, that is, when the constraint (1) (latency(mg1?, mg2?) ≤ maxLatency) is represented in Figure 4 as a trigger (predicate [P6] in the style specification) is violated. The system is then said to be in a state of incorrect configuration, and consequently, it no longer conforms to the follow_me style. In other words, it satisfies all stylistic constraints except [P6]. If we remove the concerned constraint from the architectural style schema, we obtain the style (SkewedStyle) to which this incorrect configuration belongs. The policy that we propose offers three possible solutions depending on the cause of the failure. So after testing a number of conditions, the appropriate solution is triggered. In Solution compartment, we specify the ActionPlan. In the ActionPlan compartment, we specify actions (Replicate_M (Replicate Manager), Connect_Replica, Find_M (Find Manager) etc.) interconnected with a piping connection. These solutions are just examples of plausible solutions.

- If the policy detects a problem of high latency, as we said, and one of the two managers is overloaded (mg1? load ≥ maxLoad), the proposed solution (Solution1) is to replicate the overloaded manager with its existing links to lighten its load and connect the replica to its neighbours. So if mg1? is overloaded (mg1? load ≥ maxLoad), we apply Solution1 with the parameters [m1? := mg1?, m2? := mg2?] allowing the replication of mg1? with its existing links and connecting the replica to its neighbours. Else, that is, mg2? is overloaded (mg2? load ≥ maxLoad), we apply Solution1 with the parameters [m1? := mg2?, m2? := mg1?] allowing the replication of mg2? with its existing links and connecting the replica to its neighbours.

- If the policy detects degradation of the bandwidth of the connection and the configuration contains only two managers (in the [else] branch), the proposed solution (Solution2) is to create a third manager and connect it to the already existing ones (mg1? and mg2?). This solution not only resolves the problem but also provides the opportunity for better scalability to the system because increasing the number of managers reinforces the parallel operation of managers.

The repair policy uses a strategy in which it executes only one of the solutions: if the first condition succeeds, it applies the corresponding solution. Otherwise, it tests the following condition. The execution of the suitable solution brings the system back to a resulting configuration called NewArchitecture.

3.2 | Formal semantics of repair policies

We propose describing the semantics of repair policies in the Z notation to carry out analysis and verification.
We define 10 transformation rules allowing the generation of Z specifications from the graphical policy model. The advantage of our solution is that these transformation rules are general and independent from case studies.

3.2.1 | Action specification

Actions are reconfiguration operations. The transformation of an Action is performed using seven basic rules to automatically generate Z operation schema [1]. We do not
present these transformation rules because we obtain the Z specifications of *Actions* from the second step of the MeidyaCoM project. The following schema describes the template of an *Action* specification (reconfiguration operation) that defines the input parameters \( (c_i^?) \) as well as the pre- and post-conditions. These conditions are essential to verify that the evolution of the architecture preserves the architectural invariants. Actions are executed only if their pre-conditions are satisfied. In the Action schema below, \( \Delta_{SkewedStyle} \) indicates that the action *Action_\( i \)* changes the state of an architecture belonging to the schema *SkewedStyle*. \( PreCond_\( i \) \) and \( PostCond_\( i \) \) denote, respectively, a pre- and a post-condition of the action *Action_\( i \)*.

### 3.2.2 Action Plan mapping

An *ActionPlan* describes the change in state that results when *Actions* are executed sequentially. It is transformed as presented formally in Figure 5.

The link relying actions can be of ‘piping’ type or ‘sequential composition’ type, depending on the action type. If it is a sequential composition, it will be represented by the composition operator ‘\( \circ \)’, and if it is piping, it will be represented by the piping operator ‘\( > > \)’. The details of this transformation rule are listed in the following translation algorithm (/fcmp, /zpipe, /sdef are a latex exported format of the Z notation that is, respectively, the composition operator ‘\( \circ \)’, the piping operator ‘\( > > \)’, and the writing

---

**Figure 3** General repair policy model

**Figure 4** ‘Follow_me’ repair policy model
operator ‘\( \rightleftharpoons \)’. We used the latex format because the plug-in automatically maps UML models towards XML language describing the policy. We found that the only way to import this description under the Z/EVES theorem prover for the specification is by generating the corresponding LaTeX description through eXtensible styles language transformation (XSLT) transformation rules. Algorithm 1 details this transformation.

Algorithm 1 Transformation rule of an Action Plan

1: for each Solution do
2: if FindActionPlan() then
3: write(/begin{zed})
4: write(Select(ActionPlan[current].Name))
5: write(/sdef(ActionPlan/Action current.Name))
6: …
7: nbAction ← CountNbAction()
8: for i = 2 to nbAction do
9: TypeLink ← FindTypeLink()
10: if TypeLink = seqComp then
11: write(/fcmp(ActionPlan/Action[i].Name))
12: else
13: if TypeLink = piping then
14: write(/zpipe(ActionPlan/Action[i].Name))
15: end if
16: end if
17: end for
18: write(/end {zed})
19: else
20: exit
21: end if
22: end for

3.2.3 | Solution mapping

A Solution is formally specified as presented in Figure 5. As we have explained, a Solution describes the necessary adaptation ActionPlan provided that asserted conditions (if they exist) are satisfied. Algorithm 2 details the transformation rule of a Solution.

Algorithm 2 Transformation rule of a Solution

1: for each Solution do
2: S ← FindSolution()
3: if FindSolution() then
4: write(/begin{schema}{S.Name})
5: write(Delta Select(SkewedStyle.Name))
6: write(S parameters)
7: …
8: write(Where)
9: write(Select (Solution[current]/ActionPlan.Name))
10: write(/end {schema})
11: else
12: exit
13: end if
14: end for

3.2.4 | Policy mapping

Finally, a repair policy, as presented in Figure 3, is formally translated as follows:

where

- \( \Delta\text{SkewedStyle} \) indicates that the policy will change the system state.
- \( \text{in}_i : \text{Type} \) is an input parameter of the policy.
- \( \text{out}_i : \text{Type} \) is an output parameter of the policy. Policies do not necessarily have outputs.
- \( \text{Pre} \) conditions are specified to guard the policy. They will prohibit the policy execution if they are not satisfied.
- \( \text{Triggering Event} \) describes the detected error that triggers the policy.
- \( \text{Solution}_i \) defines a repair solution.
The corresponding translation algorithm is the following:

**Algorithm 3 Transformation rule of a Policy**

1. write(/begin{schema}{Policy.Name})
2. write(Delta Select(SkewedStyle.Name))
3. write(Select (Solution parameters))
4. ...
5. write(Where)
6. write(Select (Transition1.condition))
7. for each Transition do
8. write(Transition.condition)
9. write(Select(Solution.Name))
10. ...
11. end for
12. write(/end{schema})

3.2.5 | Case study: policy specification

The result of the transformation of the repair policy of the case study is shown in the following Z schema:

\[
\begin{align*}
\text{Solution}_2 & \equiv \text{Find}_M >> \text{Redirect}.
\end{align*}
\]

Actions Find_M and Redirect are respectively represented in the following Z schemas. The role of the Find_M action is to find the nearest and least loaded manager (ml) among the neighbouring managers (M) of (m1?). Because actions Find_M and Redirect are related with the piping link (>>), the output of Find_M (ml) is matched with the input of Redirect (m2?). The role of the Redirect action is then to redirect the links from the old manager (m1?) to the new manager (m2?) found by the action Find_M. So the set of the links between managers will be enhanced with two links (m2?, m2?) and (m2?, m2?) and the rest of relations (PushCM, PushMC, PushSM, and PushMS) and sets (M, Cl, and S) remains unchanged. The specification of these two actions is obtained from the second step of the MeidyaCoM approach:

Defining repair policies is a solution, but it is not enough. We need to ensure that defined policies generate a correct state w.r.t. the architectural style. To do that, we should prove the soundness and the consistency properties of specified policies.

3.3 | Formal verification

The transformation to Z is an intermediate phase to carry out formal reasoning by proving the soundness and consistency properties.
3.3.1  Policy soundness checking

A repair policy is represented by an operation schema used to describe changes in the system state. By adding $\Delta$SkewedStyle in the schema of the policy, the theorem prover generates two copies of the state: SkewedStyle represents the state before applying the policy, and SkewedStyle' represents the state afterwards. So once executed, the Policy will place the system in a new architecture that is represented by SkewedStyle'. If the Policy is sound, the new architecture SkewedStyle' should be an instance of the original style Style'. To verify that every execution of the policy will place the system in a correct state w.r.t. the original style, we propose to prove the following theorem:

**Theorem Soundness.**

\[ \forall \text{Policy} \cdot \text{SkewedStyle}' \Rightarrow \text{Style}' \]

The theorem states that if the policy is executed, it will generate a correct configuration w.r.t. the defined style. However, we have no guarantee that the policy will be executed, which may be due to

- a policy specification that does not preserve the constraints defined by the style (it proposes changes that violate the defined constraints)
- an initial configuration (SkewedStyle) that does not belong to the collection of states for which the proposed changes are allowed (i.e. the input configuration was not appropriate)

3.3.2  Policy consistency checking

If we prove that every Solution proposed by the Policy can be executed at least once, we can say that the policy will always be executed and generate a correct configuration. To do that, we propose to test each Solution on an incorrect configuration and evaluate the test result. The defined architecture is therefore an instance of a 'SkewedStyle'. Formally, it is defined by the schema TestArchitecture. In Z notation, an initial state is usually characterized by a decorated schema representing the state after initialization. In the predicative part, we have to initialize all variables that are mentioned in the declaration part, that is, those declared in the 'SkewedStyle':

| TestArchitecture |
|------------------|
| SkewedStyle'     |
| Variable Initialization |

Once this has been done, we can specify the execution result of the defined policy. PolicyExec describes the change in the state that occurs when the policy is executed on TestArchitecture:

\[ \text{PolicyExec} \triangleq \text{TestArchitecture} \circ \text{Policy} \]

If the execution is performed, a new architecture, which must be correct if the applied solution is sound, will result. We specify this new architecture with a state schema NewArchitecture. This schema stipulates that this architecture is an instance of the original style. In the predicative part, PolicyExec expresses the new variable values obtained following the reparation:

\[ \begin{align*}
\text{NewArchitecture} \\
\text{Style}'
\end{align*} \]

\[ \text{PolicyExec} \]

As we specified in the declaration part, NewArchitecture is an instance of Style. However, we have no proof of that yet. Therefore, the following initialization theorem has to be proved:

**Theorem Consistency**

\[ \exists \text{Style}' \cdot \text{NewArchitecture} \]

Proving this theorem ensures that the Policy (and more precisely, the tested solution) was executed and (that this policy) generated NewArchitecture, which is definitely proved to be correct w.r.t. the original style.

To sum up the checking process, we propose to verify the theorem Soundness. Once proved, we must test every solution on a corresponding incorrect configuration to check its consistency. If proving the two theorems succeeds, we can affirm that the defined policy is both sound and consistent.

3.3.3  Case study: formal verification

QoS properties are used in triggering the policy and selecting the best solution. In the checking process, we use them as parameters to prove the consistency of the solution.

To verify that every execution of the policy will place the system in a correct state w.r.t. Structural_Style (the structural style is the architectural style containing only structural properties), we have proved the soundness theorem (Figure 6).

To prove that every solution proposed by the policy can be executed at least once, we propose to test each solution on an incorrect configuration and evaluate the test results. For example, to test Solution1, three steps are necessary. The first consists of specifying an initial invalid state where the manager m2 is overloaded, knowing that the configuration contains only two managers, and the latency between these two managers is above the threshold:

\[ \begin{align*}
\text{Init}_\text{Sk}_\text{follow_me} \\
\text{Sk}_\text{follow_me}'
\end{align*} \]

\[ \begin{align*}
M & = \{m1, m2\} \\
C & = \{c1, c2\} \\
S & = \{s1, s2\} \\
\text{PushMM} & = \{\{m1, m2\}, \{m2, m1\}\} \\
\text{PushCM} & = \{\{c1, m1\}\} \\
\text{PushMC} & = \{\{m1, c1\}\} \\
\text{PushSM} & = \{\{s1, m2\}\} \\
\text{PushMS} & = \{\{m2, s1\}\}
\end{align*} \]

\[ m1, m2 : \text{Manager} \]
\[ c1, c2 : \text{Client} \]
\[ s1, s2 : \text{Sink} \]

\[ \begin{align*}
\text{maxLatency} & = 5 \\
\text{maxLoad} & = 5 \\
\text{latency}(m1, m2) & = 7 \\
\text{m1.load} & = 2 \\
\text{m2.load} & = 8
\end{align*} \]
In the second step, we specify the execution result of the Policy. It is described as follows:

\[\text{Policy}_\text{Exec} \triangleq \text{Init}_\text{Sk}_\text{follow}_\text{me}\text{'}\_\text{Policy}(mg1:=m1, mg2:=m2)\text{'}\_\text{NewArchitecture}_\text{Structural}_\text{Style}’\_\text{Policy}_\text{Exec}.\]

The third step consists of proving a second theorem called Consistent_Policy to ensure that NewArchitecture is an instance of Structural_Style (Figure 6).

We developed the proof of this theorem under the Z/EVES theorem prover. As shown by Figure 6, the value Y in the Proof column means that the theorem was successfully proved. This result is obtained after 24 proof steps. The Z proof is semi-automatic. Consequently, this means that the tested solution was an executed and generated New-Architecture that respects all the structural properties of the stylistic constraints defined by the style follow_me([P1]...[P5]).

By proving these theorems for all solutions, we ensure that the specified policy is sound and that it will place the system in a correct state w.r.t. its structural properties of the architectural style follow_me.

4 | TOOL SUPPORT

We present in Figure 7 right side a detailed representation of the MeidyaCoM project.

The style design step is supported by an Eclipse plug-in that provides a reusable set of proved communication schemas and the corresponding composition rules. Schemas are represented by UML graphs and come with their formal semantics expressed in Z notation. To design a publish/subscribe architectural style, the architect simply applies a set of composition rules on the necessary communication schemas. The plug-in generates an XML file describing the designed style then generates the corresponding LATEX description through XSLT transformation rules. The architect can import this description under the Z/EVES theorem prover for specification refinement according to the application requirements.

The reconfiguration operation step is also supported by an Eclipse plug-in (it was Fujaba, and it is now supported by an Eclipse plug-in) by which the designer specifies the dynamic software architecture according to the UML profile. First, the architect defines the reconfiguration operations associated with the pre- and post-conditions. Second, the plug-in automatically maps UML models towards XML language describing the designed reconfiguration operation then generates the corresponding LATEX description through XSLT transformation rules. The architect can import this description under the Z/EVES theorem prover for specification refinement according to the application requirements.

- **Policy design:** The architect models repair policies using the diagram editor. It is a tool where diagrams, as well as graphical elements, can be created to model policies. First, the designer defines different solutions to the defined problems (i.e. by defining the set of action plans and actions) using an Eclipse plug-in that we propose. Second, the plug-in automatically maps UML models towards XML language. It ensures that each generated XML description is valid w.r.t. the meta-model. We have implemented and integrated a set of controls. For example, there is the possibility of creating a single SkewedStyle. An ActionPlan can only be created inside a Solution, and even an Action can be created only within an ActionPlan. The diagram editor of the policy is represented with an illustration of the case study ‘Follow me’ on the home page of our approach.

- **Policy transformation:** After modelling the repair policy, the plug-in transforms the XML file, according to transformation rules expressed with the XSLT language, into Z specifications saved in a LATEX (.tex) file. This LATEX file will then be imported under the Z/EVES theorem prover to visualize Z schemas, validate their syntax, and verify the theorems. These steps are represented in Figure.

- **Correctness and consistency proof:** The designer should interact with the Z/EVES theorem prover to check the syntax of the proposed repair policies as well as

---

Proof steps used in this proof, along with their results, are available on the home page of our approach: [http://www.redcad.org/members/hadjkacemm/tools/MeidyaComPolicy/index.htm](http://www.redcad.org/members/hadjkacemm/tools/MeidyaComPolicy/index.htm) #CaseStudies
their soundness and consistency. To prove the proposed theorems, we use a set of proof commands executed sequentially in the Z/EVES tool.

Using our tool, the modelling, the specification, and the proof become easier. For example, the modelling of the 'Follow me' case study is done in a few minutes, the
transformation step takes just a few seconds, and the proof steps used in this proof take only a few minutes. In conclusion, applying our approach using our tool is very reasonable for practical applications.

5 | EVALUATION

In this section, we present an evaluation of *MeidyaCom-Policy* approach and the associated tool. We will focus on two ways of evaluation: the first is the evaluation based on case studies analysis, and the second is based on logical reasoning. Three evaluation criteria are defined to evaluate our approach: *generality, simplicity, and usability* [14, 15].

5.1 | Evaluation based on case studies analysis

We applied *MeidyaCom-Policy* on two case studies dealing with different problems and contexts. The first case study, ‘Follow me’, presented here, describes an audio application that is aware of the user’s location and the location of speakers around the room. The second case presents an *Emergency Operation System* (EOS) [8]. This case study has been introduced in the NetQos European project and is used to describe software architectures [16, 17]. The application involves structured groups of robots or military personnel who cooperate for the realization of a common mission during a crisis. Each robot or person has a different function; we find a controller, a coordinator, or an investigator. The architecture of this system follows the *Publish/Subscribe* architectural style, and the propagation of events between these participants is mediated through a network of event dispatchers called managers.

Both cases are designed as decentralized distributed applications because the system contains several distributed event dispatchers connected with the general P2P topology. For this adaptation in the EOS case study, we treat the case of a broken connection between a dispatcher and client. Concerning the ‘Follow me’ case study, where we introduce some performance properties, we handle the case of high latency between two dispatchers.

These case studies have shown that for various contexts and resource descriptions, our approach can be applied while also considering the quality of results and performance. We are planning to test other case studies with different topologies of the dispatchers (hierarchical, cyclic, or acyclic) and a different application context to reach the *generality* criterion, which states that the proposed method can cover the most important aspects of the application domain.

Another advantage is that through these case studies, we prove that the defined graphical and visual notations of our UML profile simplify the modelling of repair policies. With this advantage, we reach the *simplicity* criterion, which states that the desired application and its different properties can be easily defined based on the proposed approach.

For the two case studies, the screenshots of the modelling phase, the generated XML files of the validation phase, and the generated Z files of the verification phase are available on our website.\(^1\)

5.2 | Evaluation based on logical reasoning

For the second evaluation approach, we rely on logical reasoning to prove the *usability* criterion. This criterion states that the application should be easy to use by users for whom it was intended. One of the goals of our *MeidyaCom-Policy* approach is to simplify, to the maximum extent possible, the work of the architecture designer. We propose a GUI for our defined UML profile to graphically model repair policies with a palette of tools.

Our solution saves time thanks to the automatic transition between the development process phases (GUI to XML, XML to Z). In addition, it saves the effort of the designers because the tool supports the validation between the meta-model and the model and then ensures the correct by construction and the automatic transition between the development process phases. In the validation phase, Z/EVES helps us to the proof. It can be used either to explore or to prove properties. But having an informal specification or plan being conducted is crucial for its effective use. Z/EVES is effective for guided proofs that usually rely on a high level of automation, leaving the internal proof steps as hidden as possible. Yet a deep understanding of each tactic behaviour is fundamental for fine-grained control of the prover.

Figure 9 presents the benefit in terms of effort and time spent by a designer using our approach compared with the manual approach. The unit of the y-axis is a ratio of the time spent in each phase averaged over the two case studies. More than 50% (10/18) is provided automatically by the integrated tools. The design phase takes the same effort and time using our approach or the manual approach. Compared with the

\(^1\)http://www.redcad.org/members/hadjkacem/m/tools/MeidyaComPolicy/index.htm
manual approach, the time and effort taken in the transformation phase using our approach is zero. The verification phase using our approach takes half the effort and time of the manual approach.

The time taken by a user to model a policy depends on the size of the policy itself. For example, the modelling and verification of the policy presented here can be done in less than 30 min. This is calculated by relying on the demonstration video showing the modelling and the transformation of the policy with the tool according to our approach.

In conclusion, we have tested our tool in two case studies, and it has been proved that the tool allows modelling these case studies appropriately with ease and considerable time saving. Using our tool, the modelling, specification, and proof become easier thanks to the tool’s graphical interface and automation of the transformation. If we go to scale now and deal with real cases, we suppose that our tool is capable of handling such applications. Therefore, applying our approach is very reasonable for practical applications.

6 | RELATED WORK

Our research contributes to the domain of software adaptation. We are interested in UML modelling and formal verifications. There are three general branches of research relevant to our work: self-healing systems, UML modelling, and the transformation from UML to Z. We provide an overview of the most relevant works and outline the differences between them and our approach.

6.1 | Software adaptation

Many researchers have contributed a great deal to communities on architecture-based self-adaptive/self-healing systems. For example, Jeff Krammer et al. [18] have described their vision of self-management at the architectural level and have proposed an approach to adaptive software engineering [19].

Garlan et al. have proposed an adaptation framework for self-healing systems that consists of monitoring, analysis, and adaptation [20]. In their work, the architectural style is used as the basis for deciding when to apply modifications that are design-time artefacts [3]. When a violation of a stylistic constraint is detected, the most appropriate adaptation strategy will be triggered. Each stylistic property is associated with an adaptation strategy that is formed by a set of tactics. To validate their approach, they examine the results of the adaptation; if the problem is solved, the adaptation is likely to be correct. However, no tests have been applied to check the conformity of the new configuration according to the architectural style [21]. This makes the difference between their work and our work.

Taylor et al. focus on using dynamically maintained policies, called knowledge-based policies, and system observations that allow the evolution of adaptation policies at runtime [4] using architectural style. However, they do not check the reliability of adaptation w.r.t. the stylistic properties.

Perry et al. have examined the basic functional requirements for self-healing systems. They have explored a number of issues related to architectural designs for incorporating runtime reflection and adaptation into software systems [22]. They have proposed some enhancements in architectural description languages (ADLs) and system design environments to incorporate explicit support for self-healing architectural frameworks. For example, they add to the ADLs new syntactic constructs to specify, define, and combine architectural strategies for self-healing systems. Like previous works, they do not address the correctness of adaptation w.r.t. the architectural style.

Yang et al. propose a self-healing approach for dynamic software architecture [23]. They use architectural reflection to make the system architecture observable and controllable. Once the repair strategy has been selected, the reconfiguration manager executes the reconfiguration by transforming the architecture graph. The architectural style is used to ensure the consistency of changes. However, the safety of reconfiguration has to be checked every time we need to execute a repair strategy, even if it is the same.

Similar to the aforementioned research, our work focuses on the use of the architectural style as a basis for system self-repair. Our major contribution is that we propose to check the soundness property for every defined repair policy, once a time and for all. We do not need to verify the safety of the adaptation every time we want to execute a repair policy because it has already been checked. Concerning QoS properties, they should be checked after adaptation.

6.2 | UML

UML is generally used for describing software architecture. We notice that there are mainly three axes for using UML: without extension ‘As-is’, the light extension ‘lightweight’, and the heavy extension ‘heavyweight’:

- Using UML ‘as-is’ consists of using UML without any modification.
- The ‘lightweight’ extension [24, 25] enables adapting meta-models for specific purposes by extending existing meta-classes. Adaptations are defined by stereotypes that are grouped in a profile.
- The ‘heavyweight’ extension [1, 26, 27] allows modifications on the existing meta-models and creates new ones or replaces existing semantics by modifying the meta-model of UML.

In the literature, we note that few works have used UML to describe architecture reconfiguration. Hadj Kacem et al. extend UML meta-models for modelling structural and dynamic architectures [1]. They model software architecture reconfiguration operations with a UML profile. Their notation is inspired by graph grammar, which proposes a graphic view allowing the
addition and the removal of components and/or connections. They extend this notation by new concepts, and they simplify and organize the specification of the architectural style and reconfiguration operations.

With the basic UML standard, we cannot model repair policies. We extend UML meta-models by proposing a new UML profile to resolve such a problem. Our approach follows the heavyweight extensibility. This extension consists of enriching the syntax and the semantics of the UML notation with new stereotypes related to the domain of self-healing.

6.3 | From UML to a formal language

UML is a popular language for modelling software systems. Its popularity derives from several factors such as simplicity, ease of use, and a certain degree of intuitiveness and flexibility in the notation. However, UML’s lack of formality hampers its applicability to critical systems. To overcome these deficiencies, a number of approaches combine UML notation with formal languages. Some use this strategy to specify static and dynamic software architecture aspects [28, 29]. Some works propose translating UML to a formal language so that formal languages can be used [1, 30, 31].

Hadj Kacem et al. adopt an approach based on the transformation from UML towards formal specifications [1, 32]. This approach allows an automatic transformation of the architectural style and of reconfiguration operations towards Z language to check the consistency and the conformity of the architecture with regard to the architectural style. Compared with this work, Hadj Kacem et al. transform the architectural style and reconfiguration operations to Z specifications, but MeidyaCoM-Policy transforms repair policies to Z specifications. In MeidyaCoM-Policy, we use the previously generated specifications of the architectural style and reconfiguration operations as a basis for our repair process.

Dupuy et al. propose to translate the UML class diagram into the Z language by means of translation rules. They developed a tool (RoZ) and integrated it in Rational Rose environment [30]. Compared with this work, Dupuy et al. propose translating the UML class diagram into the Z language. However, in our work, we translate repair policy models that extend the activity diagram of UML 2.0, so we are transforming two different models from UML.

Like these studies, we adopt an approach based on transforming semi-formal models to formal specifications. The major advantage of this approach is to overcome the lack of precision of UML language.

Other works have proposed the combination of the UML language with a formal language without transformation. Souza et al. [16, 33] have proposed the notion of awareness requirements and introduce the formalization of these requirements based on the Object Constraint Language, which is a declarative language for describing rules that apply to UML models. Their proposal includes a language for expressing awareness requirements, as well as techniques for elicitation and implementation based on the FEAT requirements monitoring framework. Compared with this work, we address the planning step in the monitoring, analysis, planning, and execution loop; however, they work within the monitoring step.

7 | CONCLUSIONS AND FUTURE WORK

We have outlined our MeidyaCoM-Policy approach, which is an extension of the MeidyaCoM project and based primarily on three steps. The first consists of defining a new UML profile for modelling sound repair policies. This profile is used to guide and assist design activities for describing repair policies using a visual notation. The second consists of the automatic transformation from the UML model to Z specifications. The third consists of checking the soundness and consistency of these policies to guarantee a safe reparation. The checking process ensures that once the policy is executed, it will re-establish the violated constraint, and it will generate a correct configuration with regard to the structural properties of architectural style.

We have presented the implementation of the repair policy profile as an Eclipse plug-in, allowing architects to design repair policies. The plug-in also allows the automatic transformation of graphical models to formal notation by means of transformation rules that are implemented with the XSLT language. MeidyaCoM-Policy is applicable to all architectural styles and is instantiated for the publish/subscribe style.

In future works, we plan to make repair policies dynamically adaptable. The adaptation of the policy includes the dynamic change in policy parameters (stability issue of adaptation) and the choice of the appropriate policy that should be triggered by a predefined set of policies at runtime (multi-triggering issue of repair policies). The choice of the policy may be achieved through the learning mechanism, which offers us the opportunity to generate new policies if necessary. That is, if we fine the appropriate solution to a problem (the solution that ensures the best qualities of service), this solution will be automatically executed the next time.

We are also working to apply our solution to self-adaptive IoT applications in the domains of health and agriculture.

REFERENCES

1. Kallel, S., Hadj Kacem, M., Jmaiel, M.: Modelling and enforcing invariants of dynamic software architectures. Softw Syst Model 11(1), 127-149 (2012)
2. Loulou, I., et al.: P/S-CoM: building correct by design Publish/Subscribe architectural styles with safe reconfiguration: Building correct by design publish/subscribe architectural styles with safe reconfiguration. J Syst Software 83(3), 412-428 (2010), https://doi.org/10.1016/j.jss.2009.09.031
3. Cheng, S.W., et al.: Using architectural style as a basis for system self-repair. In: Wiese J. (Ed.) IFIP 17th World Computer Congress - TC2 Stream/3rd Conference on Software Architecture, pp. 45-59. Kluwer, B. V., Deventer, The Netherlands (2002)
4. Georgas, J.C., Taylor, R.N.: Towards a knowledge-based approach to architectural adaptation management. In: WOSS'04: the 1st ACM SIGSOFT Workshop on Self-Managed Systems, pp. 59-63. ACM, New York (2004). http://doi.acm.org/10.1145/1075405.1075417

5. Loulou, I., et al.: A formal architecture-centric approach for safe self-repair. In: The 2nd International Conference on Web and Information Technologies (ICWIT'2009), Sfax, Tunisia (2009)

6. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall International Series in Computer Science (1996)

7. Meisels, I., Saaltink, M.: The Z/EVES reference manual (for version 1.5). Reference Manual, ORA Canada (1997)

8. Loulou, I., et al.: Making sound policies for self-healing systems. In: The 9th Annual International Conference on New Technologies of Distributed Systems (NOTERE'2009), pp. 178-185, Montreal, Canada (2009)

9. Hadj Kacem, M., Kallel, S., Jmaiel, M.: Meidya: Modelling and Enforcing Invariants of Dynamic Software Architectures http://www.redcad.org/projects/meidya/ (2021). Accessed 31 May 2021

10. Kirk, R., Newmarch, J.: A location-aware, service-based audio system. Consumer Communications and Networking Conference, Las Vegas (2005)

11. Eugster, PT, et al.: The many faces of publish/subscribe. ACM Comput Surv 35, 114-131 (2003). http://doi.acm.org/10.1145/857076.857078

12. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Trans Comput Syst 19(3), 332-383 (2001). DOI https://doi.org/10.1145/380749.380767. http://doi.acm.org/10.1145/380749.380767

13. Object Management Group (OMG): UML 2.0 Superstructure Specification, Final Adopted Specification. OMG document (2003)

14. Michelsen, C.D., Dominic k, W.D., Urban, J.E.: A methodolog y for the objective evaluation of the user/system interfaces of the madam system using software engineering principles. In: The 18th Annual Southeast Regional Conference, pp. 103-109. ACM, New York (1980)

15. Sacha, K.: Evaluation of software quality. In: The Conference on Software Engineering: Evolution and Emerging Technologies, pp. 381-388. IOS Press, Amsterdam (2005)

16. Souza, V.E.S., et al.: Awareness requirements for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 60-69. ACM, Honolulu (2011)

17. Guennoun, K., Drira, K., Chassot, C.: Architectural adaptability management for mobile cooperative systems. In: International Conference on Multimedia and Ubiquitous Engineering, pp. 1130-1135. IEEE Computer Society Los Alamitos, CA, USA (2007)

18. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering, FOSE'07, pp. 259-268. IEEE Computer Society, Washington (2007). https://doi.org/10.1109/FOSE.2007.19

19. Kramer, J., Magee, J.: A rigorous architectural approach to adaptive software engineering. J Comput Sci Technol, 24(2), 183-188 (2009)

20. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: WOSS'02: the 1st Workshop on Self-Healing Systems, pp. 27-32. ACM, New York (2002). http://doi.acm.org/10.1145/582128.582134

21. Cheng, S.W., et al.: Software architecture-based adaptation for grid computing. In: HPDC'02: the 11th International Symposium on High Performance Distributed Computing, p. 389. IEEE Computer Society, Washington (2002)

22. Hawthorn, M., Perry, D.: Architectural styles for adaptable self-healing dependable systems. In: ICSE05: International Conference on Software Engineering. ACM/IEEE, St. Louis (2005)

23. Qun, Y., Xian-Chun, Y., Man-Wu, X.: A framework for dynamic software architecture-based self-healing. SIGSOFT Softw. Eng. Notes 30(4), 1-4 (2005). http://doi.acm.org/10.1145/1082983.1083007

24. Selonen, P., Xu, J.: Validating UML models against architectural profiles. SIGSOFT Softw. Eng. Notes 28(5), 55-67 (2003). http://doi.acm.org/10.1145/773126.773138

25. Marija, M.R., Nikanj, M., Nenad, M.: Architectural style requirements for self-healing systems. In: WOSS'02: the 1st Workshop on Self-Healing Systems, pp. 49-54. ACM, New York (2002). http://doi.acm.org/10.1145/582128.582138

26. Pérez-Martínez, J.E., Sierra-Alonso, A.: UML 1.4 versus UML 2.0 as languages to describe software architectures. In: EWSA'04: the 1st European Workshop Software Architecture, EWSA 2004, St Andrews, Lecture Notes in Computer Science, 3047, pp. 88-102. Springer (2004)

27. Pérez-Martínez, J.E.: Heavyweight extensions to the UML metamodel to describe the C3 architectural style. SIGSOFT Softw. Eng. Notes 28(3), 55-67 (2003). http://doi.acm.org/10.1145/773126.773134

28. Kim, S.K., Carrington, D.: A formal model of the UML metamodel: the UML state machine and its integrity constraints. In: ZB'02: the 2nd International Conference of B and Z Users on Formal Specification and Development in B and Z, Lecture Notes in Computer Science, 2272, pp. 497-516. Springer, London (2002)

29. Pradilla, M., Rossi, M., Mandrioli, D.: Architrio: a UML-compatible language for architectural description and its formal semantics. In: Forte, pp. 381-395 (2005)

30. Dupuy, S., Ledru, Y., Chabre-Pezoul, M.: An overview of RoZ: a tool for integrating UML and Z specifications. In: CASE'06: the 12th International Conference Advanced Information Systems Engineering, Stockholm.Lecture Notes in Computer Science, vol. 1789, pp. 417-430. Springer (2000)

31. Tounsi, I., et al.: Transformation of compound SOA design patterns. In: Proceeding of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp. 415-426. ACM, New York (2010)

32. Tounsi, I., et al.: Transformation of compound SOA design patterns. In: Proceeding of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp. 415-426. ACM, New York (2010)

33. Souza, V.E.S., et al.: Awareness requirements. In: Software engineering for self-adaptive systems II, pp. 153-161. Springer Heidelberg (2013)

How to cite this article: Hadj Kacem, M., Tounsi, I., Hadj Kacem, A.: MeidyaCoM-policy: approach for modelling and checking repair policies for self-healing systems. IET Soft. 15(4), 260–276 (2021). https://doi.org/10.1049/sfw2.12026
APPENDIX

This appendix summarizes the complete syntax of the Z modelling language grammar used in the paper. They appear in the order of schemas, schematic decorations, set-theoretic symbols, predicates, and relations.

APPENDIX Z NOTATION

Schemas

= schema definition

Δ Changeable

sequential composition

\( \gg \) piping

SCHEMATIC DECORATIONS

\( n \) initial state of a variable

\( n' \) final state of a variable

\( n? \) an input parameter

\( n! \) an output parameter

SET THEORETIC SYMBOLS

\( \{} \) set braces.

\( \{ \ldots \} \) set definition, predicate definition.

\( \in \) set membership.

\( \cup \) set union.

\( \notin \) non-membership.

\( \subseteq \) finite set.

\( \subseteq \) subset relation.

PREDICATES

\( \forall \) universal quantification

\( \exists \) existential quantification

\( \land \) And

\( \lor \) Or

\( = \) Equality

\( \neq \) Inequality

\( \leftrightarrow \) equivalence

\* such that

RELATIONS

\( \leftrightarrow \) relation

\( \text{dom} \) domain

\( \text{ran} \) range

\( \text{dom} \) domain restriction

\( a \rightarrow b \) or \( (a, b) \) ordered pair

\( R^- \) relational inversion