
A formal architecture-centric approach for safe self repair

Imen LOULOU* — **Imen TOUNSI**** — **Mohamed HADJ KACEM[§]**
— **Ahmed HADJ KACEM^{§§}** — **Mohamed JMAIEL****

ReDCAD-Research unit

** Faculty of Sciences of Gabes*

ioulou@acm.org -

*** National Engineering School of Sfax*

imen.tounsi@redcad.org - mohamed.jmaiel@enis.rnu.tn

§ Higher Institute of Computer Science and Multimedia of Sfax

mohamed.hadjkacem@fsegs.rnu.tn -

§§ Faculty of Economics and Management of Sfax

ahmed.hadjkacem@fsegs.rnu.tn

RÉSUMÉ. Ce papier propose une approche formelle centrée architecture pour la spécification de politiques correctes de réparation architecturale. Une politique est considérée correcte si, une fois exécutée, elle permet de remettre le système dans un état sain vis-à-vis de ses propriétés. Les politiques et les théorèmes de preuve sont formellement définis en notation Z, et les spécifications sont implémentées sous l'outil de preuve Z/Eves. Nous définissons également un profile UML qui vise de décrire la structure d'une politique de réparation conformément à notre approche. Nous validons l'approche sur le style architectural Publier/Souscrire.

ABSTRACT. In this paper, we propose a formal architecture-centric approach for specifying correct architectural repair policies. By correctness, we mean that once executed, the policy will make the system in a correct state w.r.t its properties. Policies and proof theorems are formally defined in Z notation and specifications are implemented in the Z/Eves theorem prover. We also define an UML profile which aims to describe the structure of a repair policy following our approach. We validate the approach on an increasingly commonly used architectural style for component-based systems : Publish/Subscribe style.

MOTS-CLÉS : Auto-réparation, auto-adaptation, politiques correctes, style architectural, style Publier/Souscrire, spécification formelle, profile UML.

KEYWORDS: Self-repair, safe adaptation, sound policies, architectural style, publish/subscribe style, formal specification, UML profile.

1. Introduction

Nowadays a topical issue in the software engineering research area consists of producing software systems able to adapt themselves to unexpected errors (components failure, connections going down, etc.) or surrounding changes. In this work, we focus on architectural self-repair capabilities. A recent branch of work has demonstrated the interest of using architecture styles as a basis for run-time monitoring, error detection, and repair [CHE 02a, GEO 04]. Architectural style was usually introduced as a set of types for architectural elements (components and connections) with a definition of invariant properties and a description of how elements can be combined. As presented in [CHE 05], to support the requirements of run-time self-repair, the usual concept of style should be augmented with specific repair policies and specific reconfiguring operators. Formalized styles make the properties of interest explicit helping error detection. Errors correspond to architectural constraint violation which triggers the appropriate repair policy. One of the key issues in making this adaptation successful is that of ensuring the conformance of the new system state (obtained after reconfiguration) to the defined style. By conformance, we mean a system which satisfies all of the constraints defined by the style.

This paper makes a contribution in these directions by proposing a formal self-repair architecture-centric approach. The key idea is to help architects to define sound repair policies so that one can be sure that once executed, the policy will generate a stylistic correct configuration. Policies and soundness checking process are coded in Z notation [WOO 95] and implemented under the Z-Eves theorem prover [MEI 97]. We also provide a modeling solution to guide and assist design activities for describing repair policies using a visual notation based on the graphical UML notation. We instantiate the approach to an increasingly commonly used architectural style for component-based systems : Publish/Subscribe style. In previous work [LOU 07], we have proposed a formal approach supporting the correct modeling of Publish/Subscribe architectural styles and safe reconfiguration of dynamic architectures for event-based communication. In this paper, we extend this work by adding self-repair architectural capabilities. And we also show that policy soundness property has the merit to be checked using formal methods.

The remainder of this paper is organized as follows : in section 2, we present a survey of related work. Section 3 provides some background information on the Z notation. In section 4, we present the main features of a publish/subscribe architecture style. In section 5, we describe our approach to define repair-policies and we show how we can prove the soundness property. We also present the policy UML profile. Section 6 presents a case study which illustrates our approach. In section 7, we give our concluding remarks and future work perspectives.

2. Related work

Considerable research has been done on the description of self-repair capabilities for software architectures. Garlan and al. have proposed an architecture-based ap-

proach to develop auto-adaptive systems [GAR 02]. In their work, architectural style specifications are used as the basis for deciding when to apply modifications which are design-time artifacts. In their approach, reconfiguration is done via repair strategies which are composed of a sequence of repair tactics. To validate the approach, they examine the results of the adaptation ; if the problem, for which the strategy is executed, is resolved the adaptation is supposed correct. However, no tests are applied in order to check the conformity of the new configuration to the architectural style [CHE 02b]. Taylor and al. focus on using dynamically maintained policies, called knowledge-based policies, and system observations allowing the evolution of adaptation policies at runtime [GEO 04] using architectural style. Recently, Taylor and al. have applied this approach to the development of auto-adaptive robotic systems [GEO 08].

Perry and 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 [HAW 05]. They have proposed some enhancements in architectural description languages (ADLs) and system design environments to incorporate explicit support for self-adaptive architectural frameworks. For example, they add to the ADLs new syntactic constructs to specify, define and combine architectural strategies for self-adaptive systems. Yang and al. [QUN 05] propose a dynamic software architecture-based approach to self-healing. They use architectural reflection to make the system architectures observable and controllable. Moreover, architectural style is used to ensure the consistency of the 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. Indeed, after the repair strategy has been selected, the reconfiguration manager executes the reconfiguration by transforming architecture graph. The verification manager then checks the safety of the reconfiguration.

Similarly to these researches, our work focuses on the use of architecture style as a basis for system self-repair. The main contribution of our work is that we propose to check formally the soundness property for every defined repair policy. Compared to other works, in our approach, we check only once all policies. We don't need to verify the safety of adaptation every time we want to execute a repair policy since this policy has already been checked. As we have said in the introduction, we have applied the approach for an increasingly commonly used architectural style for component-based systems : Publish/Subscribe style.

3. Background

The Z notation can be split into two : a notation for discrete mathematics (set theory and predicate calculus) and a notation for describing and combining schemas, called the schema calculus. Together, they make up a mathematical language that is easy to learn and to apply. The mathematical language is used to describe various aspects of a design : objects, and the relationships between them. Mathematical objects

and their properties can be collected together in schemas : patterns of declaration and constraint. The schema calculus is a very important aspect of the Z notation, because it makes Z suitable for describing large systems by handling distinct parts of it and combining them. The schema language can be used to describe the state of a system, and the ways in which that state may change. It can also be used to describe system properties, and to reason about possible refinements of a design. We give here a brief idea on some concepts we will use in this paper.

– **State Schema** : A state schema consists of two parts : a declaration of variables ; and a predicate constraining their values :

<i>schema</i>	_____
<i>Declaration</i>	_____
<i>Predicate</i>	_____

– **Operation Schema** : To describe an operation upon the state, we use two copies of the state : one representing the state before the operation ; the other representing the state afterwards. To distinguish between the two, we decorate the components of the second schema, adding a single prime to each name. There is a convention for including two copies of the same schema, one of them decorated with a prime. If Schema describes the state of a system, then Δ Schema is a schema including both Schema and Schema'. Some operations involve either input to the system and/or output from it. To model such operations, we include additional components in the declaration part of the operation schema. The predicate part can then relate the values of these components to the states before and after the operation. There is a simple convention concerning input and output. If a component represents an input, then its name should end with a query (?) ; if it represents output, then its name should end with a shriek (!). It should be emphasized that these are not decorations, but part of the component name.

<i>Operation</i>	_____
Δ Schema	_____
$in_1?, \dots, in_n? : INPUT$	_____
$out_1!, \dots, out_m! : OUTPUT$	_____
...	_____

– **Relation** : Relation is a set of ordered pairs, a subset of a Cartesian product. If X and Y are sets, then $X \leftrightarrow Y$ denotes the set of all relations between X and Y . The domain of R is the set of elements in X related to something in Y : $domR$. The range of R is the set of elements of Y to which some element of X is related : $ranR$. We also say that X and Y are the source and target sets of R . Relations are directional, and it is always possible to reverse this direction using the relational inverse operator \sim . Source and target are exchanged, and so are the elements of each ordered pair. The relation $R\sim$ maps y to x exactly when R maps x to y .

4. Publish/Subscribe style

The strength of this event-based interaction style lies in the full decoupling in time and space between the components that have generated events, called *producers*, and the receivers, called *consumers*. This decoupling is provided by the event-service which could be centralized and implemented as a single entity called *dispatcher*, or distributed and implemented as a network of dispatchers. This makes the publish-subscribe style relatively easy to add or remove components in a system, introduce new events, register new consumers on existing events or modify the network of dispatchers [LOU 07].

Dispatchers cooperate together to route information from the producers to the subscribed entities. This interaction is governed by a principle of information propagation requiring that produced information have to reach all subscribed consumers [LOU 07].

In addition, several interconnection topologies have been proposed : hierarchical, acyclic peer-to-peer, and general peer-to-peer [CAR 01]. The hierarchical topology is an extension of the centralized approach in which a set of interconnected dispatchers maintain a master/client relationship with other dispatchers. In the acyclic peer-to-peer topology, dispatchers communicate with each other symmetrically as peers, adopting a protocol that allows a bidirectional flow of subscriptions, advertisements and notifications. In this topology, any two dispatchers are connected with at most one path. Removing the constraint of acyclicity from the acyclic peer-to-peer topology, leads to the general peer-to-peer topology.

All these properties are related to stylistic constraints which must be preserved during a repair process. In [LOU 07], we have shown that in order to ensure the property of information dissemination, some constraints have to be made. For example, if two dispatchers communicate together then it is necessary that the information coming from the first reaches the second. So, the communication has to be bidirectional (constraint C1). Another constraint (C2) requires that all event dispatchers have to be interconnected (direct or transitive connection).

Now, if a communication path between two dispatchers goes down, the constraint C2 is violated. One solution to repair this error could propose to shift their communication path to another dispatcher. However, the network of dispatchers has a well defined topology, so the repair solution must ensure among others that the topology will remain unchanged and the constraint C2 will be re-established. We will show in the next how to preserve stylistic constraints in order to make a self-repair successful and sound.

5. Approach

In previous work [LOU 07], we have proposed to model Publish/Subscribe architectural styles as graphs and specify semantics formally in Z notation. As an extension of this work, we also suggest to specify repair policies in the same formal notation.

We implement the specification as well as the verification process under the Z-Eves theorem prover.

5.1. Architectural style Specification

In [LOU 07], we have proposed a formal approach supporting the correct modeling of Publish/Subscribe architectural styles and safe reconfiguration of dynamic architectures for event-based communication. We have elaborated a set of patterns and we have defined the corresponding composition rules to build correct architectural styles. We deem necessary to give an idea on how styles are modeled and formally defined in order to understand the work presented in this paper.

In figure 1, we present an example of what may a graph look like. The presented graph describes a set of producing entities (S) of type *Sensor*, and a manager playing the role of a single event dispatcher (M). The communication links are modeled as directed edges.

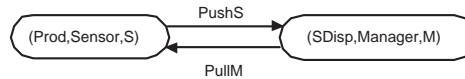


Figure 1. A communication graph between producers and a single dispatcher

The formal semantics of this graph is specified with the schema *Graph* which defines the state space.

<i>Graph</i>	
$S : \mathbb{F} \textit{Sensor}$	
$M : \textit{Manager}$	
$\textit{PushS} : \textit{Sensor} \leftrightarrow \textit{Manager}$	
$\textit{PullM} : \textit{Manager} \leftrightarrow \textit{Sensor}$	
$\textit{PushS} = \textit{PullM}^{\sim}$	[P1]
$\text{dom } \textit{PushS} = S$	[P2]
$\text{ran } \textit{PullM} = S$	[P3]
$\text{ran } \textit{PushS} = \{M\}$	[P4]
$\text{dom } \textit{PullM} = \{M\}$	[P5]

The first predicate [P1] states that the communication between sensors and the manager can be established through the *push* mode (when the sensor is the initiator), and the *pull* mode (when the manager is the initiator of communication). The four last

predicates are about the domain and the range of the defined relations. In addition, [P2] and [P3] prohibit to have an isolated sensor. Each sensor has to be connected to the manager. To be brief, we did not add here the specification expressing that the manager component has the role of an event dispatcher and that sensors behave as producer components. As we can see, architectural styles determine the types of components and connections and specify the constraints which have to be preserved during a system lifetime.

5.2. Repair Policy Specification

A software system is running correctly w.r.t. the constraints defined by its architectural style until a violation of a constraint is detected. The detection will trigger an appropriate policy in order to fix the problem. A policy could propose more than one solution to repair the system depending on what was the cause of the problem. When a communication path between a client and a dispatcher goes down, the problem could stem for example from the client (low battery of the device) or the overload of the dispatcher. If it is due to the dispatcher overload, the solution could shift the communication path of the client to another dispatcher. If it is due to the client device low battery, the solution may re-establish the connection (after battery recharge).

A repair-solution determines an adaptation action plan possibly based on some asserted conditions. Actions deemed necessary could change the system structure (such as "add component" or "remove connection",...), or find the dispatcher with the best bandwidth, etc. Actions may be primitive so that the execution is made without any pre-condition. Otherwise, they could be guarded with pre-conditions as we will show in section 6.

After these explanations, we can now define formally the notion of a repair policy :

<p><i>Policy</i></p> <hr/> <p>$\Delta State$ $in_1? ; in_2? ; \dots ; in_n? : Type$ $out_1! ; out_2! ; \dots ; out_m! : Type$</p> <hr/> <p><i>Pre – conditions</i> <i>Triggering Event</i> if <i>condition1</i> then <i>Solution₁</i> else ...<i>Solution_n</i>)</p>

Where :

- $\Delta State$ indicates that the policy will change the system state.
- $in_i?$ is an input parameter of the policy.
- $out_i!$ is an output parameter of the policy. Policies don't necessary have outputs.

- *Pre – conditions* are specified to guard the policy. They will prohibit the policy execution if they are not satisfied.
- *Triggering Event* describes the detected error which triggers the policy.
- *Solution_i* defines a repair solution.

Let's take a closer look on the line $\Delta State$. Which state the change is about? As depicted in figure 2, when an error is detected, the system will be in an incorrect configuration which does not conform to the style any more; it rather belongs to a "skewed style" which is less restrictive, since the invalid configuration satisfies all of the stylistic constraints except the one (those) which is (are) violated (*ViolConst*). The figure shows the space of valid (resp. invalid) states which is the set of correct (resp. incorrect) configurations w.r.t the style. The symbol C_i is used to describe a configuration, and it is clear that every configuration belonging to the style is also conform to the skewed style.

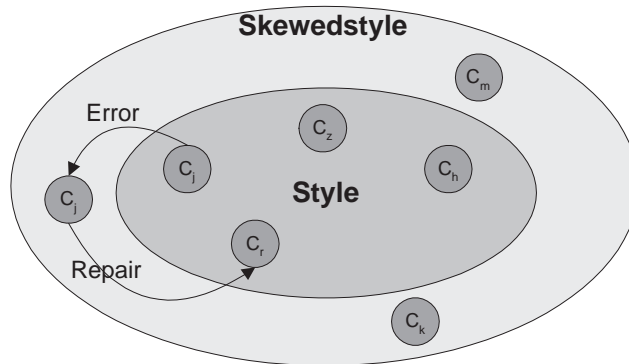


Figure 2. Valid and invalid state space

Now, let's define the repair-solution formally. As we have said, it describes the necessary adaptation action plan provided asserted conditions (if they exist) are satisfied :

<i>Solution</i> $\Delta SkewedStyle$ $in_1? ; in_2? ; \dots ; in_n? : Type$
<i>Pre – conditions</i> <i>Action_Plan</i>

Where *Action_Plan* describes the change in state that results when the mentioned actions are executed sequentially.

As we have explained above, actions could be expressed in terms of structural reconfiguration. They will change the system state which is an instance of the skewed style actually. We define a reconfiguring action with an operation schema as follows :

<i>Action</i> $\Delta SkewedStyle$
<i>Possible pre – conditions</i> <i>Structural changes</i>

Defining repair-policies is good. However, it is not enough. How one could be sure that the defined policies will make the system in a correct state w.r.t. its architectural style ?

5.3. Policy Correctness Checking

Once executed, the *Policy* will make the system in a new architecture which is represented by *SkewedStyle'*. Since, as we have explained in section 3, the $\Delta SkewedStyle$ includes two copies of State : one representing the state before the operation (*SkewedStyle*) ; the other representing the state afterwards (*SkewedStyle'*). As depicted in figure 2, if the *Policy* is correct, the new architecture should belong to the valid state space. Accordingly, it should be referenced by *Style'*. In order to verify that every execution of the policy will make the system in a correct state w.r.t the original style, the following conjecture should be a theorem :

Theorem Correctness
 $\forall SkewedStyle ; in? : INPUT$
 $\quad | precondition(SkewedStyle, in?)$
 $\quad \bullet \exists Style' ; out! : OUT \bullet Policy$

Where :

- *SkewedStyle* is the invalid state belonging to the skewed style state space,
- *in?* the input parameters of the policy,
- *precondition(SkewedStyle, in?)* are the preconditions upon state variables and input parameters which must be satisfied beforehand,
- *Style'* the state obtained after the execution of the policy and belonging to the style state space,
- *out?* the outputs of the policy once executed,
- *Policy* is the policy schema name.

The theorem states that whatever was the invalid state (*SkewedStyle*) satisfying the asserted pre-conditions, a *correct after* state can be shown to exist (*Style'*).

In order to check that a policy is style preserving, we develop the proof of this theorem under the Z/EVES theorem prover. The proof may require some iterations with Z/Eves. Two benefits can be taken when trying to prove this theorem. If the policy is correct, the proof process can uncover any missing hypotheses which characterize the collection of before states and inputs for which some after state can be shown to exist. If the policy is not correct, the proof process will not return false. It is interesting to see that it rather requires some assumptions which guide the user to identify the causes of the incorrectness, actually. Accordingly, we can correct the policy.

5.4. Repair Policy Profile

In order to give a visual aspect to our approach, we propose to model the structure of the repair policy of software architecture using a visual notation based on the graphical UML notation.

We propose a UML profile, as depicted in figure 3, to describe the repair policy of software architectures. It is composed of a meta-model associated with appropriate new notations. The meta-model is defined as a set of meta-classes defined by the UML specification and stereotypes that we define. The meta-model, extends UML 2.0 state machines diagrams [OMG 03].

Policy : As depicted by figure 3, the stereotype *Policy* is composed by exactly one *PolicyName*, one *SkewedStyleName*, one or more *Solutions* and several *Transitions*. The stereotype *PolicyName* describes a name identifying the repair policy to be modeled. The stereotype *SkewedStyleName* describes a name identifying the skewed architectural style. A *Policy* could propose more than one solution to repair the system depending on what was the cause of the problem.

Solution (from BehaviorStatemachines) : A *Solution* determines an adaptation *ActionPlan* possibly based on some asserted conditions. The stereotype *Solution* is composed of two parts : *ActionPlan* and *Transition*. The stereotype *ActionPlan* contains *Actions* and *Pseudo states*. The meta-class *Transition* specifies, in this case, the *pre-condition* that should be verified before triggering the solution.

Transition (from BehaviorStatemachines) : The meta-class *Transition* is a directed relationship between a source and a target. It may be a part of a compound transition, which ensures the sequence of *Actions* in an *ActionPlan*.

Action Plan : The stereotype *ActionPlan* describes the change in state that results when the condition is verified and the mentioned *Actions* are executed sequentially. A given *ActionPlan* may be composed by several *Actions*.

Action (from BehaviorStatemachines) : The stereotype *Action* can model structural changes such as "add component" or "remove connection", etc. and operations that query the state of the system execution in which some conditions are verified.

PseudoState (from BehaviorStatemachines) : The meta-class *Pseudostate* is typically used to connect multiple transitions into more complex transitions paths. The

attribute `kind : PseudoStateKind` defines the type of the *PseudoState* and can be one of : *initial, terminate, fork, join, choice, synch, junction* or *final*.

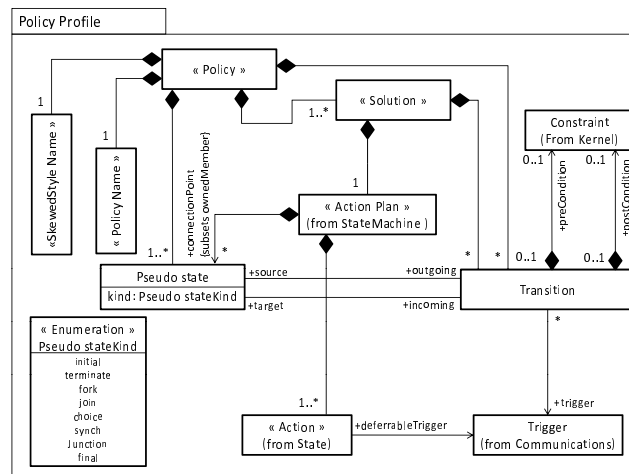


Figure 3. Repair Policy Profile

The profile defines a set of concepts that can be used for modeling the structure of a repair policy following our approach. It is a convenient way to explicit the repair process proposed by a policy. The triggering event can be mentioned and the proposed solutions can be defined according to the order imposed by the policy.

6. Case study

To illustrate our approach, we choose a case study in the context of CEO-like (Crisis Emergency Operation) activities occurring during a crisis situation [GUE 08]. The application involves structured groups of robots or military personnel who cooperate for the realization of a common mission. An emergency operation team involves various jobs : a global controller of the mission, several coordinators, and several sections of investigators. The controller manages the whole coordinators and each coordinator supervises a section of investigators.

Each job is associated with the following functions [GUE 08] :

- The controller is the entity that supervises all the application ; he/she receives the reports of all coordinators that synthesize the current context and inform him/her of mission advance.
- According to actions and objectives assigned by the controller, a coordinator manages a section of investigators by giving orders and assigning tasks to be perfor-

med. He/She must also collect, interpret and synthesize information received from its investigators and post them towards the controller.

– Investigators explore the operational field, to observe, analyze and submit reports describing the situation to their assigned coordinator. Investigators also act for helping, saving and repairing.

We propose to model this CEO application following the event-based architecture style. So, the propagation of events between participants is mediated through a network of event dispatchers called managers. Both investigators and coordinators are associated with producer-consumer components and the controller behaves as a consumer entity. According to the approach which we presented in [LOU 07], the architectural style is designed by composition and modeled with an oriented graph as it is depicted in figure 4. The formal specification is built progressively as the style was designed, based on the applied composition rules. It is defined in Z notation with the state schema *Emergency*.

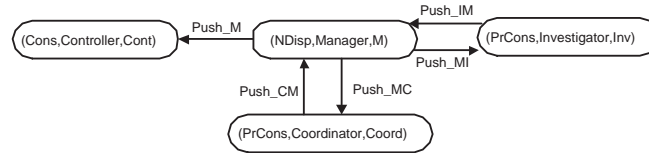


Figure 4. CEO Architectural style

<i>Emergency</i>	
$Cont : \mathbb{F}$ Controller	
$M : \mathbb{F}$ Manager	
$Inv : \mathbb{F}$ Investigator	
$Coord : \mathbb{F}$ Coordinator	
$Push_M : Manager \leftrightarrow Controller$	
$Push_IM : Investigator \leftrightarrow Manager$	
$Push_MI : Manager \leftrightarrow Investigator$	
$Push_MC : Manager \leftrightarrow Coordinator$	
$Push_CM : Coordinator \leftrightarrow Manager$	
$Push_MM : Manager \leftrightarrow Manager$	
$Push_IM = Push_MI^{\sim}$	[Pred1]
$Push_CM = Push_MC^{\sim}$	[Pred2]
$Push_MM = Push_MM^{\sim}$	[Pred3]
$dom Push_CM = Coord$	[Pred4]
$ran Push_MC = Coord$	[Pred5]
...	

Once the style is designed and the specification is built, the architect can refine the style with additional properties. We will ignore here the specification of the interconnection topology underlying the network of managers for sake of simplicity. All the properties described in the predicate part must be preserved during a healing process. Accordingly, we can define the necessary adaptation policies which will be triggered if a constraint violation is detected.

6.1. Policy specification

In this paper, we give an example of a repair policy which is triggered when the communication path from a *coordinator* to a *Manager* goes down. The considered coordinator becomes isolated and the constraints [Pred4] and [Pred5] are violated.

The system is in an incorrect configuration which does not conform to the style *Emergency* any more. More precisely, it satisfies all of the stylistic constraints except [Pred4] and [Pred5]. If we remove the concerned constraint from the *Emergency* schema, we obtain the style (*skewedstyle*) for which this incorrect configuration belongs to. In this case, we could propose two solutions for example : if the manager is overloaded, we propose to shift the communication path to another manager. If not, we rather propose to re-establish the connection (The disconnection could be due to the *coordinator* device low battery). Accordingly, the repair policy is specified with the schema *Policy* which takes as inputs the coordinator and the manager the problem is about.

<i>Policy</i>	
$\Delta Skewed_Emergency$	
$mg? : Manager$	
$crd? : Coordinator$	
<hr/>	
$mg? \in M \wedge crd? \in Coord$	
$dom Push_CM \cup \{crd?\} = Coord$	
$\wedge ran Push_MC \cup \{crd?\} = Coord$	[Triggering event]
if $mg?.load < Max$	
then $Solution1[m1? := mg?, c1? := crd?]$	
else $Solution2[m2? := mg?, c2? := crd?]$	

The first solution is specified with the schema *Solution1* which will be executed only if the manager is not overloaded (we suppose that *Max* represents the load threshold). The connection will be re-established between the coordinator and the manager through the execution of *Action_Plan1* which is defined by the reconfiguration operation *Connect_CM*.

Solution1

$\Delta Skewed_Emergency$

$m1? : Manager$

$c1? : Coordinator$

$Action_Plan1[mg? := m1?, c? := c1?]$

$Action_Plan1 \hat{=} Connect_CM$

Connect_CM

$\Delta Skewed_Emergency$

$c? : Coordinator$

$mg? : Manager$

$Cont' = Cont$

$M' = M$

$Inv' = Inv$

$Coord' = Coord$

$Push_M' = Push_M$

$Push_IM' = Push_IM$

$Push_MI' = Push_MI$

$Push_MC' = Push_MC \cup \{(mg?, c?)\}$

$Push_CM' = Push_CM \cup \{(c?, mg?)\}$

$Push_MM' = Push_MM$

The second solution is described with the schema *Solution2*, where *Action_Plan2* is used to shift the communication path to another manager. It connects the coordinator to the least loaded Manager among the nearest neighbors.

Solution2

$\Delta Skewed_Emergency$

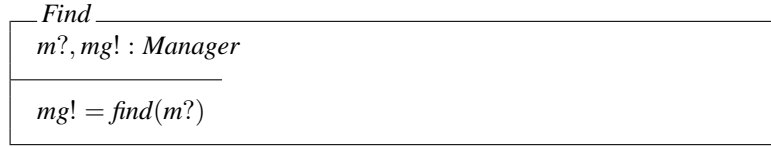
$m2? : Manager$

$c2? : Coordinator$

$find(m2?) \in M$

$Action_Plan2[m? := m2?, c? := c2?]$

$Action_Plan2 \hat{=} Find \gg Connect_CM$



The pipe operator (\gg) is used to connect operations, where the result of one operation serves as the input for another operation. The operation *Find* takes as input the overloaded manager and gives the least loaded Manager among its nearest neighbors. The operation *Connect_CM* will connect the coordinator to this manager.

In the following, we model the structure of the proposed policy by instantiating the UML profile of repair policies presented in section 5.4. The model is depicted in figure 5.

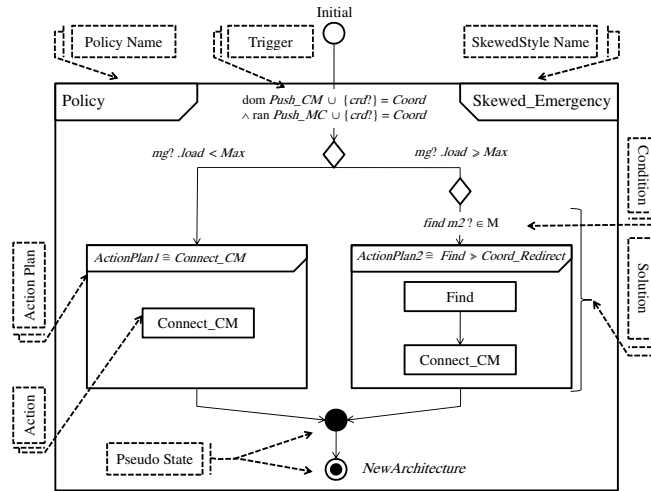


Figure 5. Repair Policy model

6.2. Correctness checking

As we have explained in section 5, in order to verify that every execution of the policy will make the system in a correct state w.r.t the original style *Emergency*, we have to prove the following theorem :

Theorem Correctness $\forall Skewed_Emergency ; mg? : Manager ; crd? : Coordinator$ $| mg? \in M \wedge crd? \in Coord$ $\wedge \text{dom } Push_CM \cup \{crd?\} = Coord$ $\wedge \text{ran } Push_MC \cup \{crd?\} = Coord$ $\bullet \exists Emergency' \bullet Policy$

We developed the proof of this theorem under the Z-Eves theorem prover. We obtain the value Y (Yes) in the column Proof, which means that the theorem was successfully proved. Accordingly, this means that every execution of the policy leads to a new configuration that respects all the stylistic constraints defined by the original style *Emergency*. With an other manner, it means that *Policy* is proved to be sound and style preserving.

7. Conclusion

The work presented in this paper focuses on the use of architecture style as a basis for system self-repair. We presented a formal architecture-centric approach for specifying sound architectural repair policies. Architectural style and policies are coded in Z notation. In order to guarantee a safe adaptation, we elaborated a checking process which allows to verify the soundness of the defined repair policy. This process has been implemented and successfully validated under the Z-Eves theorem prover. Each policy is checked once and for all. If it is sound, one can be sure that, once executed, it will re-establish the violated constraint and it also will generate a correct configuration w.r.t. the architectural style. In order to describe the structure of a repair policy following our approach, we proposed a UML profile extending UML 2.0 state machines diagrams. We applied the approach for an increasingly commonly used architectural style for component-based systems : Publish/Subscribe style.

We are developing a plugin into Eclipse allowing software architects to design repair policies using box-and-line as an extended UML-based notation. The proposed functions include also the automatic generation of valid and correct Z specifications.

8. Bibliographie

- [CAR 01] CARZANIGA A., ROSENBLUM D. S., WOLF A. L., « Design and Evaluation of a Wide-Area Event Notification Service », *ACM Transactions on Computer Systems*, vol. 19, n° 3, 2001, p. 332–383.
- [CHE 02a] CHENG S. W., GARLAN D., SCHMERL B. R., AO, SPITNAGEL B., STEENKISTE P., « Using Architectural Style as a Basis for System Self-repair », *WICSA 3 : Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, Deventer, The Netherlands, The Netherlands, 2002, Kluwer, B.V., p. 45–59.

- [CHE 02b] CHENG S.-W., GARLAN D., SCHMERL B., STEENKISTE P., HU N., « Software Architecture-Based Adaptation for Grid Computing », *HPDC '02 : Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, Washington, DC, USA, 2002, IEEE Computer Society, page 389.
- [CHE 05] CHENG S. W., GARLAN D., SCHMERL B., « Making Self-Adaptation an Engineering Reality », *Proceedings of the Conference on Self-Star Properties in Complex Information Systems*, vol. 3460 de LNCS, Springer-Verlag, 2005.
- [GAR 02] GARLAN D., SCHMERL B., « Model-based adaptation for self-healing systems », *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, New York, NY, USA, 2002, ACM, p. 27–32.
- [GEO 04] GEORGAS J. C., TAYLOR R. N., « Towards a knowledge-based approach to architectural adaptation management », *WOSS '04 : Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, New York, NY, USA, 2004, ACM, p. 59–63.
- [GEO 08] GEORGAS J. C., TAYLOR R. N., « Policy-based self-adaptive architectures : a feasibility study in the robotics domain », *SEAMS '08 : Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, New York, NY, USA, 2008, ACM, p. 105–112.
- [GUE 08] GUENNOUN K., DRIRA K., WAMBEKE N. V., CHASSOT C., ARMANDO F., EXPOSITO E., « A framework of models for QoS-oriented adaptive deployment of multi-layer communication services in group cooperative activities », *Comput. Commun.*, vol. 31, n° 13, 2008, p. 3003–3017, Butterworth-Heinemann.
- [HAW 05] HAWTHORN M., PERRY D., « Architectural Styles for Adaptable Self-Healing Dependable Systems », *ICSE'05 : IEEE/ACM International Conference on Software Engineering*, St. Louis, Missouri, USA, 2005, ACM/IEEE.
- [LOU 07] LOULOU I., KACEM A. H., JMAIEL M., DRIRA K., « Formal Design of Structural and Dynamic Features of Publish/Subscribe Architectural Styles », OQUENDO F., Ed., *Proc. of the 1st European Conference on Software Architecture (ECSA 07)*, vol. 4758 de LNCS, Spain, 2007, Springer, p. 44-59.
- [MEI 97] MEISELS I., SAALTINK M., « The Z/EVES Reference Manual (for Version 1.5) », Reference manual, 1997, ORA Canada.
- [OMG 03] OMG, « UML 2.0 Superstructure Specification, Final Adopted Specification », Omg document, 2003.
- [QUN 05] QUN Y., XIAN-CHUN Y., MAN-WU X., « A framework for dynamic software architecture-based self-healing », *SIGSOFT Software Eng. Notes*, vol. 30, n° 4, 2005, p. 1–4, ACM.
- [WOO 95] WOODCOCK J., DAVIES J., *Using Z : Specification, Refinement, and Proof*, University of Oxford Prentice-Hall International, Series in Computer Science, 1996., 1995.