Internal self-protecting for consistency and stability in an autonomic manager

Rémi Sharrock, Patricia Stolf, Thierry Monteil, Tom Guéroul

1 CNRS; LAAS; 7 avenue du Colonel Roche, F-31077 Toulouse Cedex 4, France
2 IRIT; 118 Route de Narbonne, 31062 Toulouse Cedex 9, France
3 Université de Toulouse; UPS, INSA, INP, ISAE; UT1, UTM, LAAS; F-31077 Toulouse Cedex 4, France
4 ASCOLA team (EMN-INRIA, LINA); Ecole des mines de Nantes, France
Email: remi.sharrock@inria.fr, stolf@irit.fr, monteil@laas.fr, guerout@laas.fr

Abstract—In this article we describe an approach for autonomic management of legacy software in distributed environments (cluster, grid or cloud). Our propositions have been implemented in a tool (TUNE) based on diagrams formalisms. We describe particularly the property of self-Protecting. The various mechanisms needed to maintain the consistency of the managed system, are described: autonomic creation or destruction, interruption or rollback. Finally experiments using this policy are made on the software DIET.

I. INTRODUCTION

The increasing complexity of current distributed software management needs new solutions. The computer system "selfware" was created in the year 1995 for this purpose. Wooldridge and Jennings [1] identify the fundamental properties of "self-*" by introducing the first multi-agent systems. By applying the properties of "self-*" to the computer systems Kephart and Chess [2] and Brantz [3] define in 2003 the four paradigms to be implemented at least in such systems to become self-managed: self-configuring, self-optimizing, self-healing and self-protecting.

In this article, we focus on the self-protecting property. The rest of the paper is organized as follows. Related work is first discussed in Section II. Our self management approach is presented in Section III. Section IV presents our propositions to verify the consistency and stability. The validation is presented in Section V, then finally, Section VI is our conclusions.

II. RELATED WORK

Some systems implement the self-protecting property. For example, OceanStore [7] is used in the domain of data storage. It uses an observation of geographical and regional breakdowns or detects attacks or distributed deny of service and reacts. In Jade [8], a set of possible autonomic manager could be use. The manager acts on the system based on information provided by the monitoring tasks. Self-protecting is done through the development of a specialized manager according to the application and needs.

We consider that the self-protecting property enables a system to survive by protecting itself in two ways: with an internal protection that bypasses the phenomenon of explosion and an external protection that avoids malicious attacks. The internal protection provides a global consistency and eventually stability to the system whereas the external protection copes with exogenous events like virus infections, infiltrations or attacks.

In [4] the multi-level security problems is introduced in autonomic systems (pro-active detection, intern consistency, after-attack restoring) and says that other autonomic properties like self-healing, self-optimizing or self-configuring must not interfere with the self-protecting property. The author shows that the self-protecting property is usually reduced only to the external protection case and the internal protection is missing. In our approach we consider specifically the internal protection case.

The consistency concept is usually used in the transactional domain for databases [5]. A transaction is defined as a sequence of operations on data that modifies the state. It must respect the four ACID constraints [6] (Atomicity, Consistency, Isolation, Durability). The essential aspect of consistency relies on predefined rules defining which organizations are logical and which ones are not for the considered system. Without precise rules it’s almost impossible to check the global consistency of the system.

The stability concept is usually used in the automation domain. It relies on the definition of thresholds and bounds on some system parameters as well as targeted values for the parameters (for instance, maximum and minimal bounds on CPU load or average targeted CPU load). Like in automation, the time plays an important role for the stability processes.

The system consistency and stability relies on one hand on the system evolution itself and on the other hand on the described evolution frame (rules, thresholds and bounds). If the evolution stays in the evolution frame for a given period of time, then it is consistent and stable.

III. OUR APPROACH FOR SELF-MANAGEMENT

In our approach we aim to add autonomic behavior to proprietary applications (called legacy) in a distributed environment (cluster, grid, cloud). We consider in our approach four phases of autonomic management:

- deployment phase : the software is installed on the hardware resources
- configuration and start phase : configuration files are generated and the application is started
self-management phase: the application is monitored, when events are detected, reconfigurations are applied
stop phase: the application is stopped, everything is cleaned on the resources.

We aim to express those four phases in a generic way. UML profiles are used and offer patterns of actions. The description languages we propose able to implement the self-* properties: self-configuring, self-healing, self-optimizing and self-protecting.

A. Software and hardware descriptions

A Hardware Description Diagram (HDD) describes the clusters, grid or cloud on which the software is deployed. This diagram is based on the UML class diagram: each class describes a set of hosts with the same characteristics. Each characteristic is an attribute of the class; for example, the java installation path, the user login, the deployment directory for the software, the connexion protocol (ssh, oarsh...).

A Software Description Diagram (SDD) describes the software to deploy. This diagram is based on the UML class diagram: during the deployment phase this diagram is interpreted to create a component architecture. It describes the deployment by intension, instead of describing by extension all the software instances that have to be deployed. Each class represents a part of the software which can be instantiated in several component replicas. A link between two classes generates bindings between the components instantiated from these elements. The link allows navigation in the component architecture. The attributes of the classes describe the characteristics of the software. One of the attribute precises a file which contains the management functions (this file is called the wrapper). This diagram represents a pattern to respect during reconfiguration actions.

These two formalims (HDD and SDD) are parsed during the deployment phase, they are detailed in [9].

B. Reconfiguration policy descriptions

A Software Wrapper Description Language based on XML (SWDL) describes the management functions which can be applied on the software. It precises how to interact with the software during the configuration phase, during the start and during the reconfiguration procedures. A wrapper provides start, stop and configure functions. The configure operations are used to generate the configurations files of the software and to give values to the attributes defined in the SDD. Then, these values can be modified dynamically during the reconfiguration procedures. The functions enumerated in the SWDL reference Java methods which implement actions on the software [9]. Those actions could be done using file or function call on legacy.

Generic probes are deployed with the application described in the SDD in order to monitor events like overload, crash, QoS threshold ... When a probe detects an event a notification is sent to manager which then call an appropriate reconfiguration procedure. To describe that procedure, a UML profile based on activity diagrams is introduced, it is called a Policy Description Diagram (PDD).

A PDD describes the workflow of operations/tasks to be run in order to configure or reconfigure the global environment (software and hardware). It calls functions of the SWDL and verify the self-protecting property.

Fig. 1. Global view of TUNe mechanisms

Operations in a PDD can create and destroy instances, add or remove links between them, change attributes values, or invoke methods of the SWDL. Filtering operations can also be used to designate which components are concerned by the reconfiguration procedure. Each action is represented by a node that contains its particular expression following a specific syntax [10]. Fork and Join nodes are used to parallelize the PDD execution flow by creating Threads and synchronizing them. Decision and Merge nodes can be used to create mutual exclusive paths crossed under conditions, and have to be merged before the end of the PDD. A PDD diagram can call another PDD diagram with a PDD reference. A PDD diagram can return a code with a return code node. A PDD can have optional inputs or outputs parameters.

Figure 1 shows the connection between the different diagrams. Our approach has been implemented in a prototype for autonomic management of software called TUNe (Toulouse University Network) [10].

IV. SELF-PROTECTING

A. Internal consistency rules for TUNe

First we describe the evolution framework used during the management phase of TUNe (see Section III). It is based on the architectural pattern given by the SDD. We introduce some consistency rules that have to be satisfied.

As said before, the SDD is represented by a class diagram and the system representation (SR) is an instantiation of this diagram as a component model. Therefore the SR is the internal representation of the system managed by TUNe. From the user point of view the SR is the set of the SDD class instances and the links between them. From the TUNe point of view the SR is the set of Fractal components and bindings. Finally, from
the application point of view it is the set of the distributed processes running.

Using the PDD, the user is able to modify the SR by adding/removing instances and modifying their links. The sequence of structural modifications build up the evolution phase of the SR that may end up in an inconsistent state if it’s not complying with the SDD. Indeed, the cardinality of this diagram gives a constraint on the links between the instances. These constraints specify the consistency rules regarding the SDD.

The SR is initially constructed with the initial attributes and the minimum cardinalities of the SDD. We note initial(A) and initial(B) the values of the initial attributes of two classes A and B linked together. We note \( A : [t_{min} \cdots t_{max}] \rightarrow B : [u_{min} \cdots u_{max}] \) as the minimal/maximal cardinality bounds (each instance of A has to be linked to a minimum of \( u_{min} \) instances of B and each instance of B to a minimum of \( t_{min} \) instances of A). We also note \( Nb(A \rightarrow B) \) the number of instances of A that are linked to an instance of B (and inversely).

Initially the SR is constructed so that:

1) \( \text{initial}(A) \cdot u_{min} \leq \text{initial}(B) \)
2) \( \text{initial}(B) \cdot t_{min} \leq \text{initial}(A) \)

The consistency rules for the minimal and maximal cardinalities can be written as:

3) \( Nb(A \rightarrow B) \geq u_{min} ; Nb(B \rightarrow A) \geq t_{min} \)
4) \( Nb(B \rightarrow A) \leq t_{max} ; Nb(A \rightarrow B) \leq u_{max} \)

When using method calls on components, it is possible for the user to navigate through the SR (through the links between the class instances) with a dotted syntax (the Software Wrapper Description Language SWDL). Every time a method is called (with an SWD method call), all the links used in the SWDL have to actually exist otherwise we consider the method call as being inconsistent. Therefore we call this case the local inconsistency SWD method call and the rule is:

5) \( \forall (X,Y) \in \text{SWD method call links} : Nb(X \rightarrow Y) > 0 \)

Finally, for an evolution phase to be consistent regarding the SDD, it has to satisfy rules (3) (4) and (5). Otherwise, we introduce consistency recovery mechanisms so that the system tends towards a consistent state.

B. Consistency recovery mechanisms

1) Automatic links creation: The first check is for the consistency rule (5), that is used by the SWD method calls. We introduce an audit with the following steps: before each method call, we list all the routes used by the method. We then check that all these links are actually present. If they are not present, there is a local inconsistency for calling the SWD. The Resolution of this inconsistency is to create automatically the link according to the SDD.

2) Automatic instances creation: If the creation of the links fails an autonomic instances creation procedure is done. We implement a highest level mechanism which create as many instances as needed to return to a consistent state. The resolution of inconsistency tends to lead the system in a locally consistent state iteratively using a loop creation. Note that we limit the autonomic creation of instances at one level of creation, that is new instances created automatically do not generate new creations. This choice is done because it becomes difficult to guarantee the consistency of the system with multiple level of instance creation. The system could create an infinite loop. Either way a user can still manually express in the PDD all the necessary creations.

3) Automatic interruption: Different cases cause an interruption of the PDD execution:

- manual specification of actions when errors occur in a creation link
- autonomic instances creation problem

It is also possible to express in the PDD with a condition what to do if an error occurs. Figure 2 gives an example where a decision is evaluated after an explicit link creation. If the link can not be done, the else transition is executed and the PDD is interrupted.

activity autointerruptPDDautoActivity: diagram autointerruptPDDauto

![Fig. 2. PDD: demonstration of autonomic interrupt](image)

4) Automatic rollback: At the end of a PDD execution a last verification is done to check the general consistency. For each instance of the SR the rule (4) has to be respected. This check occurs on the final node of the PDD. During this final check the same mechanisms of autonomic link and instance creation are applied. At the end if the system is not in a consistent state an autonomic rollback procedure is followed. During the execution of the PDD all the actions are stored included the autonomic actions; all those actions are timestamped. The rollback procedure follows all those actions in the reverse order. Figure 3 explains the rollback action for each action done.

V. VALIDATION

A. Case of study: DIET

As an example, we chose to deploy and manage a middleware named DIET [11]. DIET stands for Distributed Interactive Engineering Toolbox and is used as a hierarchical load balancer for dispatching computational jobs over a grid. DIET
architecture consists of a set of agents: some Master Agents (MA) are linked to Local Agents (LA) that manage a pool of computational SErver Deamons (SED). These servers can achieve specialized computational services. Communications between agents are driven by the omniORB system (OMNI). MAs listen to client requests and dispatch them through the architecture to the best SED that can carry out this service. We attached generic probes to the OMNI, MA, LA and SED to monitor the CPU load average and if the process is alive.

### B. Goals

We want to show that the property of “self-Protecting” in DIET allows the system to remain consistent with the specification given by the SDD in the reconfigurations specified by the PDD. For this we use the PDD that define the architecture reconfigurations of DIET by adding and removing instances of DIET agents (SED, LA, MA). Depending on the context of evolution defined by the SDD of DIET architecture we show the consistency mechanisms. During the management phase, the number of SED is regulated (creation/deletion) to maintain the targeted global load. That is, we want to show that it is possible to achieve some stability.

### C. SDD and PDD to manage DIET architecture

For these experiments, we use the SDD in Figure 4. This SDD is a simplified version for readability. We see that for the consistent property a system must have a server named OMNI linked to 2 MA, 10 LA and 500 SED. A MA can be linked to 5 LA. Each server is connected to its own probe. These generic sensors periodically send load (attribute refreshrate in ms) to the appropriate PDD.

We introduce the high-level global policy for managing DIET architecture especially for the management of SED. The overall policy is described by a PDD shown in figure 5. This policy is executed when TUNe receive a notification called: refreshloads.

The first input parameter sedsloads contains the references of elements monitored and the respective values of their loads. The action seds.load < − loads update all the loads values in the SR. If the load average of all SEDs exceeds 150, a policy called create new seds is called to create new instances of SEDs. If the load average drops below the value 50, the timer t1 is armed for 120 seconds and the policy called destroy seds is executed to destroy instances of SEDs. The timer ensures that the PDD destroy seds is executed only once every 120 seconds.

We define three PDDs: create new seds, create new la and create new ma that are responsible to create and start each new DIET agents. These PDDs are called by reference. In the next section, we describe the create new seds.

### D. Self protecting with create new seds

The PDD in Figure 6 represents the manual actions on a white background and autonomic actions on a gray background. The aim is to calculate the number of new SEDs to create a stable target average load (to maintain the system stability), using the following equation:

\[
 x = \left\lceil \sum_{i=1}^{n} C_i \frac{T}{T} - n \right\rceil
\]
with $x$ is the number of SEDs to create $C_i$ the load of the SED number $i$, $n$ the current number of SEDs and $T$ the target load. When `create_new_seds` is executed, new instances of SEDs are connected to the appropriate LA. Indeed, \( \text{list_{la}} = \min(\text{LA.load}) \) creates a filtered list of LAs by selecting only those that minimize their CPU load. If some LAs have the same minimal CPU load value when the creation of link is started (\( \text{bind \ list_{newsed} \ list_{la}} \)), those that are already linked to a minimum of SEDs are selected. If all LAs are already connected to the maximum allowed SEDs (defined as the maximum cardinality between LA and SED), new LAs are created (the action of connection fails and the follow the `else` way). In this case new LAs are created with the call of PDD `create_la`. This PDD is very similar to that of the creation of SEDs. It uses the creation of MAS (PDD `create_new_ma`).

Once the new SEDs are connected to the LAs, they are started with the method called `listenewseds.start`. The startup method of SEDs defined in the SWD needs to navigate through the links to the OMNI (to communicate with the server name) and to the LA (to connect the agent Local). This call has a local inconsistency since the link to the OMNI was not created. Our approach therefore creates an autonomous action `bind listenewseds OMNI` placed just before the method call. This action helps to restore consistency to have a successful call to the local SWD. Once the SEDs are started the PDD returns the OK code meaning that the execution has reached the final reconfiguration step.

Before running the endpoint of the PDD, our approach checks the consistency and possibly execute final procedures for resolving inconsistencies or cancellation. The first consistency control check for all instances of the SR if the cardinalities are correct. Here, for the new SEDs created, the cardinality of the link between probe and SED is incorrect (PROBESED (1-1)) since the link was not created. The first procedure in Resolution of this inconsistency adds action to create link `bind listenewseds PROBESED`. As each probe has to be connected to only one SED, new probes should be created. The second inconsistency resolution procedure therefore creates instances of PROBESED with the following standard actions: `probeseds PROBESED = + +`, `probeseds.configure`, `probeseds.start`. After this, a new cardinalities check is done.

Figure 7 shows on the left how the average load of all SEDs is affected when the policy of creating new instances of SEDs is performed. During the first six minutes, the average load of all SEDs is rapidly increasing and exceeds the maximum (maximum threshold) three times. A number of new computing SEDs are created to achieve the chosen optimal target CPU load of 100 %. Three new SEDs are created the first time, four in the second time and six in the third time. We can see that the average load continues to rise a little above the maximum threshold every time the maximum threshold is reached. This is the fact that it takes a few seconds to create and start all new SEDs.

VI. Conclusion

We introduced an approach to allow adding the property of "self-Protecting" systems managed by TUNe. We have focused our approach on the internal "Self-Protecting", that is to say, the protection of the system to inconsistencies. In TUNe, the context of consistency is given by the SDD, including the cardinalities between classes of SDD. Each cardinality provides information on the minimum and maximum number of instances that must be connected. During the execution of reconfigurations policies expressed using PDDs, the change in the internal system representation (SR) can be in an inconsistent state. We introduced consistency checks at two levels: at the policy implementation and at the end of a policy.

Consistency rules are introduced to detect inconsistencies. If an inconsistency is detected, various autonomic actions attempt to return the system to a consistent state. These actions are automatically added dynamically to the various policies and are able to create links and instances, start new instances and interrupt the execution of the policy. At the end of the execution of a policy and all autonomic actions, a final consistency check is performed overall. If it fails, a cancellation procedure is initiated. This procedure, like the transaction for the databases, brings the system in the initial consistent state before the execution of this policy. This procedure is based on the cancellation ("rollback") of actions.

The implementation of our approach on the DIET system shows that it is possible to apply the properties of "self-Protecting" on legacy systems. Thus, the management phase with TUNE apply the various policies of reconfiguration in a coherent framework defined by the SDD. The various solutions for resolving inconsistencies introduced can stay within the framework of evolution and avoid explosions of the system.

Our propositions aim to verify the consistency of the SR according to the SDD. However, our approach does not take into account the verification of synchronization between the SR and the real world. PDDs manage the SR because they allow structural modifications (with the actions of structural change). They also manage legacy software in the real world because they allow method calls (through the SWDL). So the SR and the real world evolve in parallel. For the moment, at the end of the PDD’s execution, a final verification is missing to ensure that the two worlds are equivalent.
REFERENCES


