Asynchronous Coordination of Stateful Autonomic Managers in the Cloud

Rim Abid, Gwen Salaün, Noel De Palma, and Soguy Mak-Kare Gueye

University of Grenoble Alpes, Inria, LIG, CNRS, France

Abstract. Cloud computing is now an omnipresent paradigm in modern programming. Cloud applications usually consist of several software components deployed on remote virtual machines. Managing such applications is a challenging problem because manual administration is no longer realistic for these complex distributed systems. Thus, autonomic computing is a promising solution for monitoring and updating these applications automatically. This is achieved through the automation of administration functions and the use of control loops called autonomic managers. An autonomic manager observes the environment, detects changes, and reconfigures dynamically the application. Multiple autonomic managers can be deployed in the same system and must make consistent decisions. Using them without coordination may lead to inconsistencies and error-prone situations. In this paper, we present our approach for coordinating stateful autonomic managers, which relies on a simple coordination language, new techniques for asynchronous controller synthesis and Java code generation. We used our approach for coordinating real-world cloud applications.

1 Introduction

Autonomic computing [17] is increasingly used to solve complex systems, since it reduces human errors [19]. It has become popular especially in cloud applications where the management is a crucial feature. Autonomic computing is based on the use of autonomic managers [18]. An autonomic manager is built as a control loop. It observes the application execution, ensures a continuous monitoring, and reacts to events and changes by automatically reconfiguring the application. The increasing complexity of cloud applications implies the use of various and heterogeneous autonomic managers, such as self-healing and self-protecting [5], with the objective to reconfigure automatically themselves.

When multiple autonomic managers monitor the same system, they should take globally coherent decisions. Hence, a manager should be aware of decisions of other managers before reacting. When it reacts without taking into account decisions of other managers handling the same application, error-prone situations may occur (*e.g.,* removing a server that will be needed). In order to avoid performance degradation and system consistency problems, and also to limit energy consumption it is necessary to coordinate all autonomic managers.

In this paper, we present our approach, whose main goal is to synthesize a controller that monitors and orchestrates the reconfiguration operations of the involved managers. The controller also prevents a manager from violating global objectives of the managers. All participants involved in the application interact asynchronously with the controller and messages are stored/consumed into/from FIFO buffers.

More precisely, an autonomic manager is described using a formal model, namely a Labelled Transition System (LTS). We used reaction rules and regular expressions to specify coordination requirements and interaction constraints. As a consequence, each manager is not only able to manage its internal behaviour but also its relationship with other autonomic managers, which is achieved in accordance with the specification of the coordination requirements. As shown in Figure 1, we propose controller synthesis techniques for asynchronously communicating managers. These techniques rely on an encoding of our inputs (LTS models and coordination requirements) into the LNT process algebra [6]. LNT is one of the input languages of the CADP toolbox [11], a state-of-the-art verification toolbox for concurrent systems. CADP compilers and minimization tools are particularly useful for generating a reduced LTS from the LNT specification. The generated LTS corresponds to all possible executions of the controller. It is worth noting that since we rely on formal techniques and tools, all the verification techniques available in the CADP toolbox can be used for validating the generated controller.

Once we have synthesized the controller LTS, Java code is generated using a code generator we developed. This Java code is finally deployed and used for coordinating real applications. We validated our approach on several variants of a N-tier Web application involving several autonomic managers, such as self-sizing or self-repair managers. We emphasize that our approach covers the whole development process from expression of the requirements to the final implementation and deployment of the solution.

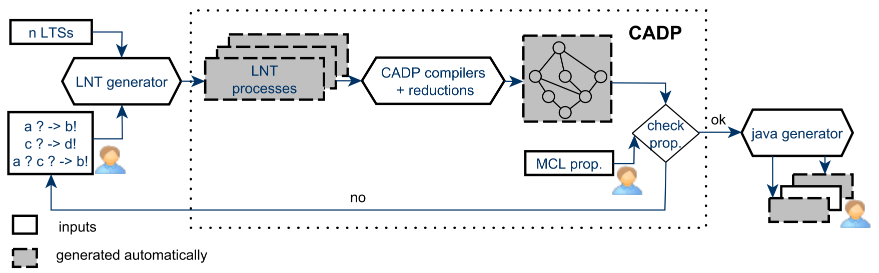

Fig. 1. Overview of our approach

The rest of this paper is structured as follows. In Section 2, we introduce our formal model for autonomic managers, the coordination language, and our running example (a multi-tier Web application). In Section 3, we present our synthesis techniques that mainly rely on an encoding into process algebra and on LTS manipulations. Section 4 introduces the code generation techniques for obtaining Java code from controller models. We discuss related work in Section 5 and we conclude in Section 6.

2 Models

In this section, we first present the abstract model used to represent autonomic managers. In a second step, we introduce reaction rules and regular expressions for specifying how the involved managers are supposed to interact together. Manager models and coordination expressions are used as input to our synthesis techniques (Section 3). At the end of this section, we introduce a typical example of distributed cloud application that we use as running example.

2.1 Autonomic Manager

Each autonomic manager is modelled as a Labelled Transition System, which is defined as follows:

Definition 1. *A Labelled Transition System (LTS) is a tuple defined as $LTS = (Q, A, T, q^0)$ where Q is a finite set of states, $A = A^! \cup A^?$ is an alphabet partitioned into a set of send and receive messages, $T \subseteq Q \times A \times Q$ is the transition relation, and q^0 is the initial state.*

We write $m!$ for a send message $m \in A^!$ and $m?$ for a receive message $m \in A^?$. A transition is represented as $q \xrightarrow{l} q' \in T$ where $l \in A$. We assume that managers are deterministic, which can be easily obtained using standard determinization algorithms [16]. Given a set of manager LTSs (Q_i, A_i, T_i, q_i^0), we assume that each message has a unique sender and a unique receiver: $\forall i, j \in 1..n$, $i \neq j$, $A_i^! \cap A_j^! = \emptyset$ and $A_i^? \cap A_j^? = \emptyset$. Furthermore, each message is exchanged between two different managers: $A_i^! \cap A_i^? = \emptyset$ for all i. Uniqueness of messages can be achieved via renaming.

2.2 Coordination Requirements

In order to coordinate multiple autonomic managers, we use reaction rules and regular expressions with their basic operators (sequence, choice, iteration) to describe the behaviour one expects from the controller. The generated controller aims at orchestrating the execution of the managers. A reaction rule consists of a set of receptions followed by a set of emissions. Basically, it expresses that when the controller receives a set of messages from managers within a certain period of time (left hand part), it must send all the messages specified in the second set (right hand part) once the period is expired. Note that the real period will be chosen during the deployment phase and both sets of actions can be received and emitted in any order.

Definition 2. *Given a set of managers* $\{M_1, \ldots, M_n\}$ *with* $M_i = (Q_i, A_i, T_i, q_i^0)$*, a reaction rule* R *is defined as* $a_1, \ldots, a_m \rightarrow b_1, \ldots, b_p$ *where* $a_j \in A_i^?$ *and* $b_k \in A_i^!$ *for* $1 \leqslant j \leqslant m$ *and* $1 \leqslant k \leqslant p$*.*

The specification of the behaviour one expects from the controller is expressed using a coordination expression.

Definition 3. *A coordination expression* C *is a regular expression over reaction rules* R*:*

$$C ::= R \mid C_1.C_2 \mid C_1 + C_2 \mid C*$$

where $C_1.C_2$ *is a coordination expression* C_1 *followed by* C_2*,* $C_1 + C_2$ *is a choice between* C_1 *and* C_2*, and* $C*$ *is a repetition of* C *zero or several times.*

It is worth noting that all participants, namely the autonomic managers and the controller to be generated, communicate asynchronously using message passing via FIFO buffers. Each participant is equipped with one input buffer. Therefore, it consumes messages from its buffer and sends messages to the input buffer of the message recipient. Once generated and added to the system, all managers communicate through the controller, which means that the controller acts as a centralized orchestrator for the whole system.

2.3 Running Example

Our running example is a JEE multi-tier application (Fig. 2) composed of an Apache Web server, a set of replicated Tomcat servers, a MySQL proxy server, and a set of replicated MySQL databases. The Apache server receives incoming requests and distributes them to the replicated Tomcat servers. The Tomcat servers access the database through the MySQL proxy server that distributes fairly the SQL queries to a tier of replicated MySQL databases.

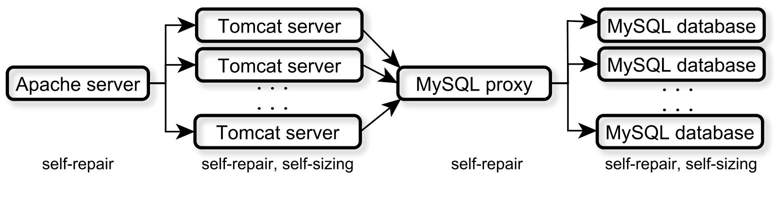

Fig. 2. A multi-tier application

The autonomic manager architecture is based on the MAPE-K (Monitor Analyse Plan Execute - Knowledge) reference model [17]. We describe this architecture using several LTS models. First, we model the behaviour of the monitor, analyse, and execute functions of the managers by what we call the application manager (Fig. 3, right), which sends messages when a change occurs in the system and receives messages indicating actual administrative changes to perform on the application. As for the plan functions, we use two models called self-sizing and self-repair managers, resp. The self-sizing manager (Fig. 3, middle) is in charge of adapting dynamically the number of replicated servers by sending the message add! (remove!, resp.) to the system when detecting an overload (underload, resp.). The overload (underload, resp.) is detected when the average of the load exceeds (is under, resp.) a maximum (minimum, resp.) threshold. We associate one instance of the self-sizing manager to the Tomcat servers and another instance to the MySQL databases. The self-repair manager (Fig. 3, left) asks the system to repair a failure by creating a new instance of the failed server. We have four instances of the self-repair manager, one per tier.

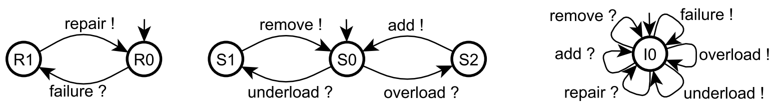

Fig. 3. (left) Self-repair manager LTS, (middle) Self-sizing manager LTS, (right) Application manager LTS

The absence of coordination between these managers may lead the whole system to some undesired situation such as adding two new servers whereas one was enough as a result of a server failure. More precisely, when the self-repair manager repairs a failure, the other replicated servers receive more requests than before the failure, which causes an overload and therefore the addition of another (unnecessary) server by the self-sizing manager.

We present below an excerpt of the requirements for the controller we want to generate for our running example. These rules ensure that all managers globally satisfy the coordination objectives. Each line presents the actions that can be received by the controller in a period T (left parts of reactions rules). At the end of each period, if the received messages match the left part of one fireable rule, it reacts by emitting the messages appearing in the right part of that rule. All messages are prefixed by the manager name (app stands for the application manager) and suffixed by the name of the tier to which is associated the manager.

```
(  app_failure_apache?   -> repair_failure_apache!                    ( )
+ app_overload_tomcat?   -> sizing_overload_tomcat!                   (❷)
+ app_failure_apache?, app_underload_tomcat?-> repair_failure_apache!(❸)
+ app_failure_tomcat?, app_overload_tomcat? -> repair_failure_tomcat!(❹)
+ ...  ) *
```

We distinguish two kinds of rules: (1) those where a unique message appears in the left part of the reaction rule (see, *e.g.*, , ❷). In that case, the corresponding controller immediately transfers that message to the manager; (2) those encoding the coordination we want to impose on managers, *e.g.,* rule ❹ permits to generate a controller that can avoid to add two Tomcat servers by forwarding only one of the two received messages on a same period of time. Last, since there is no specific order between all these rules, we use a simple regular expression where all rules can be fired at any time (combination of + and * operators).

3 Synthesis

In this section, we present our asynchronous controller synthesis techniques, which rely on an encoding of our models and of the coordination requirements into the LNT specification language. From this LNT specification, we can generate the corresponding LTS model using CADP compilers, hiding, and reduction techniques. Validation of the generated controller is also possible using CADP verification tools. This section ends with an illustration of all these techniques on our running example. All the steps presented in this section are fully automated by a tool that we developed in Python. This tool generates the LNT code as well as SVL scripts [11] that are used for invoking CADP exploration and reduction tools, which finally results in the generation of the controller LTS.

3.1 Process Algebra Encoding

The backbone of our solution is an encoding of all managers and of the coordination requirements into the LNT process algebra. The choice of LNT is motivated by several reasons. First, LNT is an expressive behavioural specification language which has a user-friendly syntax and provides expressive operators. Second, LNT is supported by CADP [11], a toolbox that contains optimized state space exploration techniques and verification tools. CADP tools allow to compile the LNT specification into an LTS, which enumerates all the possible executions of the corresponding specification. Third, CADP is a verification toolbox dedicated to asynchronous systems consisting of concurrent processes interacting via message passing. It provides many tools that can be used to make different kinds of analysis, such as model checking.

The behavioural part of the LNT specification language consists of the following constructs: action with input/output parameters, assignment (`:=`), sequential composition (`;`), conditional structure (`if`), loop (`loop`), parallel composition (`par`), nondeterministic choice (`select`), and empty statement (`null`). Each process defines an alphabet of actions, a list of typed parameters, and a behaviour built using the aforementioned operators. Communication is carried out by rendezvous on actions with bidirectional transmission of multiple values. The parallel composition explicitly declares the set of actions on which processes must synchronize. If the processes evolve independently from one another (interleaving), this set is empty.

In the rest of this section, we successively present the encoding into LNT of the different parts of our system.

Autonomic Manager An LNT process is generated for each state in the manager LTS. Each process is named using the state identifier. The alphabet of the process contains the set of messages appearing on the LTS transitions. The behaviour of the process encodes all the transitions of the LTS going out from the corresponding state. If there is no such transition, the body of the process is the `null` statement. If there is a single transition, the body of the process corresponds to the message labelling this transition, followed by a call to the process encoding the target state of the transition. If there is more than one transition, we use the `select` operator. Let us assume that two transitions $\mathtt{q} \xrightarrow{l} \mathtt{q'}$, $\mathtt{q} \xrightarrow{l'} \mathtt{q''} \in T$ have the same source state `q`. The behaviour of the process encoding `q` in LNT is `select l; q'[...] [] l'; q'' end select`, where the LNT operator `select` encodes a nondeterministic choice between `l` and `l'`.

Since a message name can be used in different autonomic manager LTSs, each message is prefixed with the manager name to avoid further name clashes. We encode emitted messages (received messages, resp.) with a `_EM` (`_REC`, resp.) suffix. These suffixes are necessary because LNT symbols ! and ? are used for the data transfer only. As an example, $m1 \in A^!$ is encoded as $m1\_EM$, and $m2 \in A^?$ is encoded as $m2\_REC$.

Coordination Requirements The coordination requirements specified using reaction rules and regular expressions correspond to an abstract version of the controller to be generated. These requirements are encoded into an LNT process called *coordination*. The process alphabet is composed of all received and emitted messages appearing in the reaction rules. The body of this process encodes the regular expression of reaction rules. Each reaction rule is translated to LNT separating both sides of the rule using the sequential composition construct (`;`). In order to make explicit in the controller LTS the logical interval of time that will be chosen in the implementation step and during which the controller receives messages, the left hand part of the reaction rule starts with an action `TBEGIN` and ends with an action `TEND`. The left hand part is translated using the `par` operator without synchronization since all messages can be received in any order. After execution of the `TEND` action, the right hand part of the reaction rule is translated using the parallel composition too, to express that all emissions can be sent in any order. As far as the regular expression is concerned, a sequence (.) of rules is encoded using the sequential composition, a choice (+) between several rules is translated using the `select` construct and an iteration ($*$) is encoded using the `loop` operator.

Architecture In this section, we present how all participants (managers and coordination expression) are composed together. The communication between them is achieved asynchronously. The coordination expression represents an abstract description of the future controller, and all messages must go through this controller, which acts as a centralized orchestrator. Each participant is equipped with an input FIFO buffer. When a participant wants to read a message, it reads the oldest message in its buffer. When a participant sends a message to another participant, it sends the message to the input buffer of that participant. LNT functions are used to describe basic operations on these buffers (*e.g.,* adding and retrieving messages). We present below, an example of function that removes a message from a FIFO buffer (*i.e.,* from the beginning).

```
function remove_MSG (q: TBUFFER): TBUFFER is
  case q in
     var hd: TMessage, tl: TBUFFER in
          nil        -> return nil
        | cons(hd,tl) -> return tl
  end case
end function
```

It is worth noting that our synthesis techniques allow one to choose buffer bounds. One can either decide to fix an arbitrary bound for buffers or to use unbounded buffers. In the first case, the only constraint is that the same buffer bound should be used when deploying the controller, otherwise unexpected behaviours and erroneous situations may occur. In the second case (unbounded buffers), the risk is to attempt to generate a controller whose corresponding state space is infinite [3]. As an intermediate solution, one can use the recent results presented in [2] for identifying whether the interactions between managers with unbounded buffers can be mimicked with bounded buffers. If this is the case, the lower bound returned by these techniques is used as the minimum buffer bound for both synthesis techniques and the deployment of the application.

A buffer in LNT is first encoded using an LNT list and classic operations on it. Then, for the behavioural part, a buffer is encoded using a process with a buffer data type as parameter. This process can receive messages from the other participants, and can synchronize with its own participant when that one wants to read a message. We generate a process encoding each couple (*participant*, *buffer*) that corresponds to a parallel composition (`par`) of the participant with its buffer. The synchronization set contains messages consumed by the participant from its buffer.

Finally, the whole system (`main` process in LNT, see below) consists of the parallel composition of all these couples. It is worth noting that since autonomic managers communicate via the controller, they evolve independently from one another and are therefore composed using the `par` operator without synchronizations. In contrast, the couple (*coordination*, *buffer*) must synchronize with all couples (*manager*, *buffer*) on all emissions from/to the managers, and this is made explicit in the corresponding synchronization set of this parallel composition.

```
process main [message_1:any, ..., message_n:any] is
   par message_p, ..., message_k in
     couple_buffer_coordination [...]
    ||
     par
       couple_buffer_manager_1 [...]
     || . . . ||
       couple_buffer_manager_n [...]
    end par
   end par
end process
```

3.2 Compilation and Verification

Now that we have encoded our inputs (models and coordination requirements) into LNT, we can use compilers to obtain the LTS corresponding to all behaviours of the LNT specification. In order to keep only the behaviour corresponding to the most permissive controller [26], we need to hide message exchanges corresponding to consumptions of the managers from their buffers and emissions from managers to the coordination expression buffer. All these messages are replaced by internal actions. We use minimization techniques available in CADP for eliminating all internal actions, removing duplicated paths, and determinizing the final LTS. Finally, we preserve only local emissions/receptions from the coordination expression point of view (messages shown in the dashed grey rectangle in Fig. 4). Transitions figuring in the final LTS are labelled with the messages corresponding to the process alphabet of the couple (*coordination*, *buffer*).

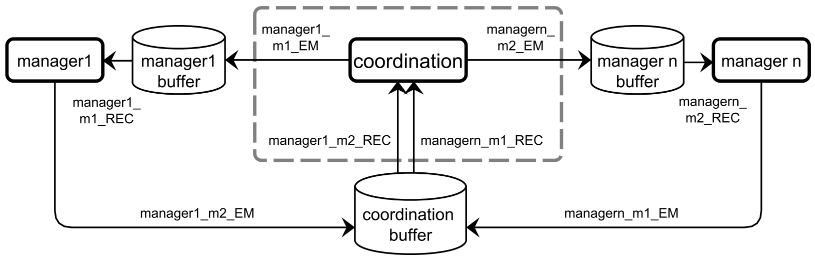

Fig. 4. Exchange of messages between the coordination expression and the managers

Last but not least, let us stress that, since the writing of the coordination expression is achieved manually by a designer, this step of our approach may lead to an error-prone expression. However, we can take advantage of the encoding into LNT to check either the controller LTS (and thus the coordination expression) or the LTS corresponding to the composition of all participants. To do so, one can use the CADP model checker, which takes as input an LTS model and a temporal property specified in MCL [20]. We distinguish two types of properties: (i) those that depend on the application (*e.g.,* the controller must eventually transmit a specific message to a certain manager), (ii) those that do not depend on the application (*e.g.,* checking the absence of deadlock).

3.3 Running Example and Experiments

We present below an example of LNT process encoding the repair manager shown in Figure 3 and its buffer. This manager synchronizes with its buffer on the `repair_failure_REC` message when this message is available in the buffer. Note that the buffer process (`buffer_repair`) is equipped with a parameter corresponding to the buffer data type, that is the structure where messages are stored, initialized to `nil`.

```
process couple_buffer_repair [repair_failure_REC: any, repair_repair_EM:
any, repair_failure_EM: any] is
  par repair_failure_REC is
    repair_R0 [repair_failure_REC, repair_repair_EM]
  ||
    buffer_repair [repair_failure_EM, ...] (nil)
  end par
end process
```

From the encoded LNT specification obtained when calling the LNT code generator, we use CADP compilers to generate the LTS describing the whole system for our running example (consisting of 194,026,753 states and 743,878,684 transitions). Then, we use hiding and minimization techniques to generate the LTS of the controller (consisting of 28,992,305 states and 46,761,782 transitions). An excerpt of the controller LTS, which focuses on the failure and overload detection in the same period of time, is shown in Figure 5. We recall that we use specific labels (namely `TBEGIN` and `TEND`) for characterizing the messages received during a same period of time. This LTS shows that when the controller receives a failure and an overload message (of a Tomcat server in this example) during a same period, it forwards only the failure message and drops the overload message. In contrast, when the controller receives these two messages in two different periods, it forwards them to the repair and sizing manager, resp.

We show below two examples of liveness properties, the first one is checked on the controller LTS and the second one on the LTS of the whole system:

– The reception of a failure message by the controller is eventually followed by an emission of a repair message

```
[true* .app_failure_tomcat_REC] inev (app_repair_tomcat_EM)
```

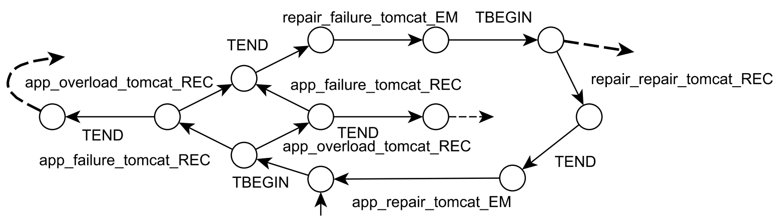

Fig. 5. Excerpt of the controller LTS for the running example

– The emission of an overload message by the application manager is eventually followed by an emission of a reparation or addition message by the controller

```
[true* .app_overload_tomcat_EM]
       inev (app_repair_tomcat_EM or app_add_tomcat_EM)
```

This property shows that the overload message is handled by the repair manager when both Tomcat failure and overload occur within a same period of time. Otherwise, it is handled by the sizing manager.

Both properties use the macro `inev (M)`, which indicates that a transition labelled with `M` eventually occurs. This macro is defined as follows:

```
macro inev (M) = mu X .( < true > true and [ not (M) ] X ) end macro
```

Our approach was applied for validation purposes on many illustrative examples of our dataset (managers and coordination requirements). Table 1 summarizes some of our experiments. Each managed application used as input is characterized using the number of managers and the coordination requirements. We give the size of the LTS (states/transitions) of the whole system as well as the controller LTS obtained after minimization (*wrt.* a strong bisimulation relation). The last column gives the overall time to synthesize the controller.

|Managers|	**Whole system LTS**		**Controller LTS**		**Time**
	|states|	|transitions|	|states|	|transitions|	(m:s)
2	2,307	6,284	118	157	0:10
3	103,725	365,845	1,360	2,107	1:15
4	145	267	38	44	0:06
5	10,063,873	39,117,110	17,662	28,003	43:59
6	1,900	4,945	186	285	0:08
10	300,000	1,686,450	1,786	3,471	6:54

Table 1. Experimental results: LTSs size and synthesis time

We observe that, for some examples (gray lines), the size of the generated controller LTSs and the time required for generating those LTSs grow importantly when one of the managers exhibit looping behaviours, and particularly cycles with send messages (see, *e.g.*, the 4th example in Table 1). On a wider scale, we note that LTS sizes and generation times increase with the number of managers in parallel (see, *e.g.*, the last line of Table 1).

4 Code Generation and Deployment

We present in this section our Java code generation techniques, which allow to deploy controllers in the context of real-world applications. In particular, we show some experimental results for our running example where the autonomic managers are coordinated using a centralized controller generated with our approach.

4.1 Java Code Generation Techniques

Our Java code generation techniques are based on the use of object-oriented programming. They take as input the controller LTS synthesized beforehand, and automatically generate all java classes, methods, and types necessary for deploying it. The controller LTS is encoded as an instance of a Java class `LTS`. This class relies on two classes, namely a class `State` and a class `transition` which represents the transitions between the states. The `LTS` class also defines an attribute `cstate` representing the current active state in the controller model. This variable is initialized with the LTS initial state. Some Java code is necessary to interface the controller with the running application. We particularly define a method called `react` that takes as input a list of messages received within a period of time and applies successive moves according to the received messages, the current state of the controller, and the behaviour of the generated controller. This method computes the messages that the controller has to send as reaction to these received messages, and updates the current state of the controller.

4.2 Deployment

Our generated Java code can be deployed and applied on concrete applications using the event-based programming paradigm. The period of time described using special actions `TBEGIN` and `TEND` in the controller LTS has to be instantiated with a real value. This period is computed using sampling techniques and implemented using the `sleep` method in Java. The choice of this period cannot be realized during the synthesis phase and is achieved just before deployment. A wrong choice of this period may lead to the reception of these actions in different periods.

The main behaviour of the controller (`run` method) consists of an infinite reactive loop, which successively receives events from the application, computes reactions (messages to be sent by the controller), and encodes these messages as events too. A part of the Java program is dedicated to converting the events raised by the application into the input format of the `react` method, and conversely translates the output of the `react` method into a list of events executed by the system. Each event contains the corresponding message and additional information, for instance a failure event has also as parameter the impacted server and further information (identifier, port, etc.).

4.3 Experiments on our Running Example

In this section we present some experiments we performed when deploying and running our controller for the multi-tier application introduced previously. To do so, we used a virtualized experimental platform based on Openstack, which consists of six physical machines on which we instantiate virtual machines.

The JEE multi-tier application is initially configured and deployed with a server at each tier, *i.e.*, an Apache Web server, a Tomcat server, a MySQL proxy, and a MySQL database. The initial deployment phase is automated using a dynamic management protocol allowing to connect and start the involved servers and database in the right order [1, 10]. In a second step, we use jmeter to inject increasing load on the Apache server and thus to simulate the clients that send HTTP requests on the managed system. Once we have at least two active Tomcat servers and two MySQL databases, we start simulating failures using a failure injector. When we start injecting failures, we stop augmenting the workload on the Apache server and keep the same load for the rest of the execution. The failure injector is flexible and can be used for affecting any active server (Apache, Tomcat, MySQL, etc.), any number of times (single failure or multiple failures of the same or of different servers), and at any time (same period of time, different periods of time, etc.). We conducted our experiments on applications with or without controller. We have considered different scenarios with failures of the Apache server and of the MySQL proxy as well as failures/load variation of the Tomcat servers and of the MySQL databases.

Figure 6 shows an excerpt of the system behaviour after 500 minutes since the application deployment. We observe that, at this moment, the application is composed of five Tomcat servers and three MySQL databases. Figure 6 presents several cases of failure injection. As an example, at minute 508, a failure of a replicated MySQL database causes a workload increase on the other replicated servers. These two actions happen in the same period, but the controller forwards only the failure detection to the repair manager. Accordingly, a single MySQL database is added by the repair manager and the workload returns at once to its average value.

We made several experiments in which we varied the number of failures, the Apache load, and the minimum/maximum thresholds of the Tomcat servers and of the MySQL databases. In all these cases, we observe that the controller succeeds in detecting and correcting the problems while avoiding undesired opera-

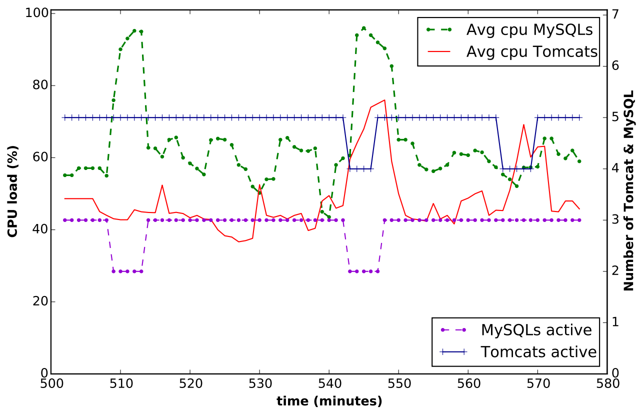

Fig. 6. Tomcat and MySQL failure/overload in a coordinated environment

tions, that is, the unnecessary addition/removal of VMs. Figure 7 shows experimental results obtained with different number of failures. For instance, we see that when injecting 14 failures to our running application, the controller applies 18 reconfiguration operations on the system (instead of 40 without controller), and thus avoids 22 undesired operations.

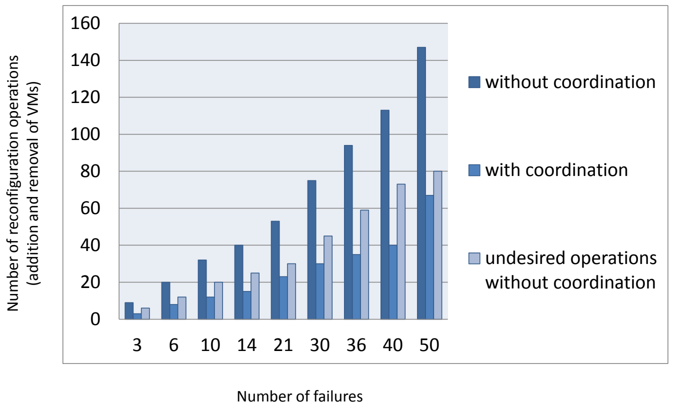

Fig. 7. Number of reconfiguration operations with/without coordination and number of undesired operations avoided by coordination

5 Related work

Controller synthesis for discrete event systems was originally introduced by Ramadge and Wonham [26, 24]. In [26], the authors present a controllable language as a solution for the supervisory of hybrid control systems. This solution generates controllers from a given system called plant and designed as a finite automaton. [24] proposes a supervisor synthesis algorithm, which allows to automatically generate a controller from a plant modelled as a finite automaton and properties to be ensured by the controller. The generated controller permits all possible legal executions. This synthesis approach is based on a classical two-person game approach. These approaches can be characterized as restrictive because they directly influence and impact the controlled system.

In [9], the authors introduce an approach based on contract enforcement and abstraction of components to apply a modular discrete controller synthesis on synchronous programs. These programs are presented by Synchronous Symbolic Transition Systems. The authors integrate this approach in a high-level programming language combining data-flow and automata. Another decentralized supervisory control approach for synchronous reactive systems is presented in [25]. This work is based on finite state machines and computes local controllers that act on the subsystems to ensure a global property. The local controllers are automatically generated and this approach was applied to several examples for validation purposes. This approach allows decentralized control whereas we generate a centralized controller. Moreover, they rely on synchronous systems and synchronous communication semantics, whereas we assume asynchronous systems and communication, meaning that the controllability hypothesis is impossible in our context.

In [22], the authors propose a generic integration model that focuses terms of reciprocal interference. This generic model can be used to manage the synchronization and coordination of multiple control loops, and it was applied to a scenario in the context of cloud computing and evaluated under simulation-based experiments. This paper does not provide any synthesis techniques for coordinating the multiple loops, and coordination is achieved in a rather manual way.

[21] presents a framework for the coordination of multiple autonomic managers in the cloud computing context. These works use a protocol based on synchronous mechanisms and inter-manager events and actions along with synchronization mechanisms for coordinating these managers. The main difference compared with our work is that this paper focuses on quality of service whereas our focus was on behavioural and functional aspects of the system execution.

Other recent works [8, 12, 13] propose some techniques based on synchronous discrete controller synthesis for coordinating autonomic managers, such as self-repair and self-sizing managers. The communication between the generated controller and the managers is synchronous and uses a synchronous language BZR, which cannot impose a specific order between requirements and contains multiple and complicated operations. This approach uses a background in synchronous systems and languages, whereas our approach assumes that communication is achieved asynchronously.

[7] presents the Aeolus component model and explains how some activities, such as deployment, reconfiguration, and management phases of complex cloud applications, can be automated in this model. Aeolus takes as inputs high-level application designs, user needs, and constraints (*e.g.,* the number of required ports that can be bound to a client port) to provide valid configuration environments. This work presents some similarities with ours, but does not propose solutions for verifying that the constraints are satisfied in the target configurations.

In [4], the authors present an extension of TOSCA (OASIS Topology and Orchestration Specification for Cloud Applications) in order to model the behaviour of component's management operations. More precisely, they specify the order in which the management operations of an instantiated component must be executed. In this work, the authors explain how management protocols are described as finite state machines, where the states and transitions are associated with a set of conditions on the requirements and capabilities of the components.

In [23], the authors introduce AutoMate, a framework for coordinating multiple autonomic components hosted on Grid applications, using high-level rules for their dynamic composition. The rules are executed using a decentralized deductive engine, called RUDDER, and composed of distributed specialized agents. RUDDER deploys the rules and coordinates their execution. It assigns priorities to these rules in order to resolve conflicting decisions between them. However, it uses a manual administration to evaluate and update the interaction rules.

6 Conclusion

In this paper, we propose new controller synthesis techniques to generate a centralized controller that allows to orchestrate a set of autonomic managers. These managers are modelled as LTSs and the set of coordination requirements is specified using reaction rules and regular expressions. The generated controller communicates with the autonomic managers asynchronously using message passing via FIFO buffers. Our solution for controller synthesis relies on an encoding of our models and of the coordination requirements into the LNT process algebra. From this encoding, an LTS can be generated using CADP compilers, and hiding and reduction techniques. This LTS exhibits all the possible executions of the controller. One can also take advantage of this encoding to validate the generated controller with the CADP verification tools, such as the Evaluator model checker. Indeed, since coordination requirements are written by a human being, they can be erroneous, which results in that case in an erroneous controller as well. Finally, we propose code generation techniques to automatically obtain the Java code corresponding to the controller LTS. We validated our approach with many variants of the multi-tier Web application we used as running example in this paper.

It is worth noting that our approach covers all the development steps from the design of the coordination requirements to the actual deployment of the synthesized controller, which helps to coordinate at runtime real-world applications. In addition, these synthesis techniques can be used to control other applications where components are modelled as LTSs and communicate asynchronously. This is the case in application areas such as Web services, multi-agent systems, or hardware protocols.

A first perspective is to generate distributed controllers instead of a centralized controller. This would permit to preserve the degree of parallelism of the system, where the involved participants could exchange messages without systematically passing through a unique controller. Another perspective aims at applying performance evaluation for the whole system using IMC (Interactive Markov Chain) theory [15, 14].

Acknowledgements. This work has been supported by the OpenCloudware project (2012-2015), which is funded by the French *Fonds national pour la Société Numérique* (FSN), and is supported by *Pôles* Minalogic, Systematic, and SCS.

References

1. R. Abid, G. Salaün, F. Bongiovanni, and N. De Palma. Verification of a Dynamic Management Protocol for Cloud Applications. In *Proc. of ATVA'13*, volume 8172 of *LNCS*, pages 178–192. Springer, 2013.
2. S. Basu and T. Bultan. Automatic Verification of Interactions in Asynchronous Systems with Unbounded Buffers. In *Proc. of ASE'14*, pages 743–754. ACM, 2014.
3. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.
4. A. Brogi, A. Canciani, J. Soldani, and P. Wang. Modelling the Behaviour of Management Operations in Cloud-based Applications. In *Proc. of PNSE'15*, volume 1372 of *CEUR Workshop Proceedings*, pages 191–205, 2015.
5. R. Buyya, R.N. Calheiros, and X. Li. Autonomic Cloud Computing: Open Challenges and Architectural Elements. In *Proc. of EAIT'12*, pages 3–10. IEEE Computer Society, 2012.
6. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 2011.
7. R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Aeolus: A Component Model for the Cloud. *Inf. Comput.*, 239:100–121, 2014.
8. G. Delaval, S.M.K. Gueye, E. Rutten, and N. De Palma. Modular Coordination of Multiple Autonomic Managers. In *Proc. of CBSE'14*, pages 3–12. ACM, 2014.
9. G. Delaval, H. Marchand, and E. Rutten. Contracts for Modular Discrete Controller Synthesis. In *Proc. of LCTES'10*, pages 57–66. ACM, 2010.
10. X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. De Palma. Reliable Self-deployment of Cloud Applications. In *Proc. of SAC'14*, pages 1331–1338. ACM, 2014.
11. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.

12. S.M.K. Gueye, N. De Palma, E. Rutten, and A. Tchana. Coordinating Multiple Administration Loops Using Discrete Control. *SIGOPS Oper. Syst. Rev.*, 47(3):18–25, 2013.
13. S.M.K. Gueye, E. Rutten, and A.Tchana. Discrete Control for the Coordination of Administration Loops. In *Proc. of UCC'12*, pages 353–358. IEEE Computer Society, 2012.
14. H. Hermanns. *Interactive Markov Chains: And the Quest for Quantified Quality*, volume 2428 of *LNCS*. Springer-Verlag, 2002.
15. H. Hermanns and J.P. Katoen. Automated Compositional Markov Chain Generation for a Plain-Old Telephone System. *Science of Computer Programming*, 36(1):97 – 127, 2000.
16. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
17. M.C. Huebscher. and J.A. McCann. A Survey of Autonomic Computing Degrees, Models and Applications. *ACM Comput. Surv.*, 40, 2008.
18. J.O. Kephart. Research Challenges of Autonomic Computing. In *Proc. of ICSE'05*, pages 15–22. ACM, 2005.
19. J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.
20. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.
21. F.A. De Oliveira, T. Ledoux, and R. Sharrock. A Framework for the Coordination of Multiple Autonomic Managers in Cloud Environments. In *Proc. of SASO'13*, pages 179–188. IEEE, 2013.
22. F.A. De Oliveira, R. Sharrock, and T. Ledoux. Synchronization of Multiple Autonomic Control Loops: Application to Cloud Computing. In *Proc. of COORDINATION'12*, volume 7274 of *LNCS*, pages 29–43. Springer, 2012.
23. M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. AutoMate: Enabling Autonomic Applications on the Grid. *Cluster Computing*, 9(2):161–174, 2006.
24. P.J.G. Ramadge and W.M. Wonham. The Control of Discrete Event Systems. *Proc of the IEEE*, 77(1):81–98, 1989.
25. A. Belhaj Seboui, N. Ben Hadj-Alouane, G. Delaval, É. Rutten, and M. Yeddes. An Approach for The Synthesis of Decentralised Supervisors for Distributed Adaptive Systems. *International Journal of Critical Computer-Based Systems*, 2(3/4):246–265, 2011.
26. W.M. Wonham and P.J.G. Ramadge. On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization*, 25(3):637–659, 1987.