Asynchronous Synthesis Techniques for Coordinating Autonomic Managers in the Cloud

Rim Abid, Gwen Salaün, Noel De Palma

University of Grenoble Alpes, LIG, CNRS, France

Abstract

Cloud computing allows the delivery of on-demand computing resources over the internet on a pay-for-use basis. From a technical point of view, 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 article, we first present a simple language for expressing coordination constraints given a set of autonomic managers. Second, given a coordination expression written with that language, we propose new synthesis techniques for automatically generating an asynchronous controller. These synthesis techniques work in two steps by successively generating a model of the controller and a Java object corresponding to this model. This Java code is finally used for deploying the generated controller. As far as evaluation is concerned, we validated our approach by using it for coordinating real-world cloud applications.

Key words: Asynchronous Coordination, Autonomic Managers, Distributed Cloud Applications, Synthesis Techniques

1. Introduction

Managing complex distributed applications is a challenging problem because manual administration is no longer realistic for complex systems. Autonomic computing is a promising solution for automating the administration functions, which focus particularly on replicating virtual machines, destroying or adding them, and handling virtual machine failures in the cloud. These operations are executed by different autonomic managers considered as control loops. Each manager observes the application execution, ensures a continuous monitoring, and immediately reacts to changes by automatically executing reconfiguration tasks. Several managers can be deployed to supervize the same application and must make consistent decisions. Nonetheless, using them without coordination may lead the system into inconsistencies and error-prone situations (*e.g.,* removing a server that is necessary). As a consequence, the use of multiple managers (*e.g.,* self-repair and self-sizing managers) implemented in the same system requires taking globally consistent decisions. Hence, a manager should be aware of decisions of all managers before reacting.

We present in this article our synthesis techniques for generating a controller, which aims at coordinating several managers. The generated controller prevents every manager from violating global objectives of all the managers. Figure 1 shows an example with two managers (M1 and M2) administrating an application. The right hand part of this figure particularly illustrates the interest of the controller for taking globally consistent decisions (by filtering some event for instance as shown in this example).

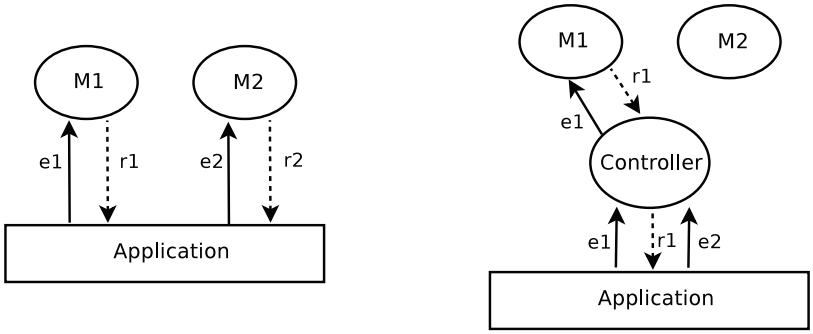

Figure 1: Administration tasks without or with coordination

Our controller synthesis techniques assume that all participants (managers and generated controller) interact using asynchronous communication semantics. This means that all the messages transmitted from/to the managers (controller, resp.) are stored/consumed into/from FIFO buffers. It is worth emphasizing that our approach is twice *asynchronous* in the sense that it applies on asynchronous systems (no global clock) and it relies on asynchronous communication semantics (communication via buffers).

Let us now present our solution with more details, as depicted in Figure 2, which gives an overview of our approach. We consider as input a set of autonomic managers. Each manager is described using a formal model, namely a Labelled Transition System (LTS). As a first contribution, we define a set of reaction rules and regular expressions to specify the coordination requirements and interaction constraints. This simple language aims at expressing in an abstract way the relationship between the managers and the behaviour we expect from the controller to be generated. Given a set of manager LTSs and the coordination requirements, we propose synthesis techniques for generating an abstract model (LTS) for our controller. To do so, we rely on an encoding of our inputs (LTS models and coordination requirements) into the LNT specification language [7]. LNT is expressive enough for representing all the inputs and the way they interact together. Moreover, LNT is equipped with a rich toolbox, called CADP [16], that is used for automatically obtaining an LTS model 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, a Java program is obtained using a code generator we developed. This Java program is necessary to finally deploy and use the synthesized controller for coordinating real applications. In this article, we present a typical example of a N-tier Web application for illustration purposes. We have validated our approach on several variants of this distributed application involving several instances of autonomic managers, such as self-sizing or self-repair managers. Note that our approach covers the whole development process from the expression of the requirements to the final implementation and deployment of our solution.

This article is an extended version of a conference paper published in [2].

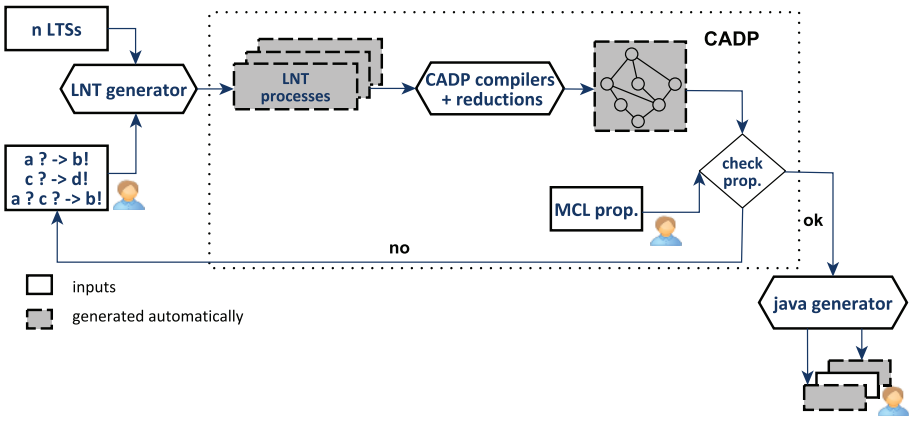

Figure 2: Overview of our approach

The key additions of this journal version are as follows: (i) a more in-depth presentation of the process algebra encoding, (ii) a refined description of the controller LTS and of its generation process, and (iii) the full development of our approach on a real-world case study.

The rest of this article is structured as follows. In Section 2, we introduce our formal model for autonomic managers and the coordination language. 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. Section 5 illustrates our approach on a multi-tier Web application. We discuss related work in Section 6 and we conclude in Section 7.

2. Formal Models

In this section, we first introduce the abstract model that we used to represent an autonomic manager. Second, we propose reaction rules and regular expressions for specifying how the involved managers are supposed to interact together through the controller that we want to generate. The models of the managers and the coordination expression are used as input to the synthesis techniques introduced in Section 3.

2.1. Autonomic Manager Model

Autonomic managers (also called administration loops) describe the causal ordering of events raised and emitted to the system. Such a behaviour is

described using Labelled Transition System (LTS) because this is a simple and graphical model for specifying sequential behaviour, non-deterministic choice, and loops. Each autonomic manager is modelled as an LTS, which consists of a set of states and a set of transitions that connect those states. Formally, an LTS is defined as follows:

Definition 1. *A Labelled Transition System is a tuple defined as a tuple* (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 received messages,* $T \subseteq Q \times A \times Q$ *is a transition relation, and* $q^0 \in Q$ *is the initial state.*

A send message $m \in A^!$ is written $m!$ and corresponds to a message destinated to another manager or to the controller. A received message $m \in A^?$ is written $m?$ and corresponds to a message to be consumed by the actual manager. A transition relating two states is represented as $q \xrightarrow{l} q' \in T$ where $q, q' \in Q$ and $l \in A$. We assume that managers are deterministic, which can be easily obtained using standard determinization algorithms [20].

Given a set of manager LTSs defined as (Q_i, A_i, T_i, q_i^0), we assume that each message should have 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. The uniqueness of messages can be achieved via renaming.

2.2. Coordination Requirements Specification

Our approach aims at generating a controller that acts as an orchestrator for coordinating the execution of the autonomic managers. In the coordinated system, all the messages emitted by the managers are caught by the controller, which can take global decisions for the whole system. In order to describe the behaviour one expects from the controller, we use a simple coordination language, which consists of reaction rules and regular expressions with their basic operators (sequence, choice, and iteration). We define a reaction rule as a set of receive messages followed by a set of send messages.

Definition 2. *Given a set of managers* $\{M_1, \ldots, M_n\}$, $M_i = (Q_i, A_i, T_i, q_i^0)$, *a reaction rule* R *is defined as* $a_1?$, ..., $a_m? \rightarrow b_1!$, ..., $b_p!$ *where* a_j $(1 \leqslant j \leqslant m)$ *corresponds to a message received from a manager* M_i ($a_j \in A_i^!$ *because this message is emitted by* M_i*) and* b_k $(1 \leqslant k \leqslant p)$ *corresponds to a message emitted to a manager* M_l ($b_k \in A_l^?$ *because this message is received by* M_l*).*

Informal semantics. Such a rule expresses that when the controller receives a set of messages from the managers within a certain period of time, it must send all the messages specified in the second set, once the period is expired. A rule is triggered when the received messages match *the exact set of input messages* (left part of a rule). If there are two rules with the same left part, this introduces some non-deterministic behaviour in the system. If there is no rule with the set of received messages as left part, no rule is triggered. All the messages involved in those rules are prefixed by the manager identifier for avoiding name clashes. Note that the period is implicit in these rules. The real period will be chosen during the deployment phase.

Example. Let us imagine the following example of reaction rule: m1_msg1?, m2_msg2? $\rightarrow$ m3_msg3!. This rule indicates that when the controller to be generated receives a message msg1 from manager m1 and a message msg2 from manager m2 within a same period of time, then it reacts by emitting a message msg3 to manager m3. It is worth emphasizing that these messages are reversed in the controller with respect to the managers. As an example message msg1 is a send message (msg1!) in the m1 LTS model whereas this is a receive message in the controller model (msg1?).

In some cases, one may want to specify a specific order of application of those rules. This can be achieved in our coordination language using regular expressions of reaction rules. Thus, the specification of the behaviour one expects from the controller is expressed using a coordination expression C defined as follows:

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

Note that all the participants, namely the autonomic managers and the controller to be generated, communicate asynchronously exchanging messages via FIFO buffers [8]. Each participant is equipped with one input buffer. This buffer stores all the messages sent to that participant. Therefore, a participant consumes messages from its buffer and sends messages that are added to the input buffers of the message's recipients. Once the controller is generated (as described in Section 3) and is part of the system,

all the managers communicate through it. This means that the controller acts as a centralized orchestrator for the whole system, as shown in Figure 1. Last but not least, we assume reliable communication, that is, messages are guaranteed to reach their destination (no message loss), in the order they were sent. We use the same communication model in all the steps of our approach (synthesis, implementation, deployment) presented in the rest of this article.

Our main motivation for the coordination language we propose in this section was simplicity. This is why we chose message matching and regular expressions. We validated the usability of this formalism by asking engineers with cloud computing background to use our language on real-world examples. The language turned out to be intuitive and simple enough for specifying coordination requirements for those real-world systems.

In the next section, we will show how, from such an abstract specification of the coordination requirements, we automatically generate an abstract model of the corresponding controller.

3. Asynchronous Synthesis Techniques

In this section, we present new asynchronous controller synthesis techniques, which rely on an encoding of our models (aut format [1]) and coordination requirements into the LNT specification language. From this LNT specification, we can generate the corresponding LTS model of the controller using CADP compilers, hiding, and reduction techniques. We also use CADP verification tools to validate the generated controller. All the steps presented in this section are fully automated using a tool that we developed in Python. This tool generates the LNT code as well as SVL scripts 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

We present successively in this section the encoding into LNT of the different parts of our system, *i.e.,* autonomic managers, coordination requirements, and architecture. We chose LNT for several reasons. First, LNT is an expressive behavioural specification language with user-friendly syntax and

[1] The aut format is the textual format used to represent automata in the CADP toolbox, see http://cadp.inria.fr/man/aut.html

operational semantics. Second, LNT is supported by CADP [16], a toolbox that provides optimized state space exploration techniques and verification tools. CADP tools allow us to compile the LNT specification into an LTS, which represents all the possible executions of the corresponding specification.

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`), non-deterministic 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. For details about the LNT syntax and semantics, the reader may refer to [7].

Autonomic manager encoding. An LNT process is generated for each state in the manager LTS. Each process is named using the identifier of the corresponding state. The alphabet of the process consists of the set of send and receive messages appearing on the LTS transitions. The behaviour of the process encodes all the transitions of the LTS going out from the corresponding state and if necessary calls the processes encoding the target states of those transitions. 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. Finally, if there is more than one transition, we use the `select` operator, which corresponds to a non-deterministic choice between several possible behaviours (similarly to '+' or '[]' in the CCS [23] and CSP [19] process algebras, resp.). Let us assume that two transitions $\texttt{q} \xrightarrow{l} \texttt{q'}$, $\texttt{q} \xrightarrow{l'} \texttt{q''} \in T$ have the same source state `q`. The behaviour of the process encoding `q` in LNT is:

select `l; q' [...] [] l'; q''[...]` **end select**

where `select` encodes a choice between `l` and `l'`, and `q'` and `q''` are two processes encoding the behaviour of both target states.

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.

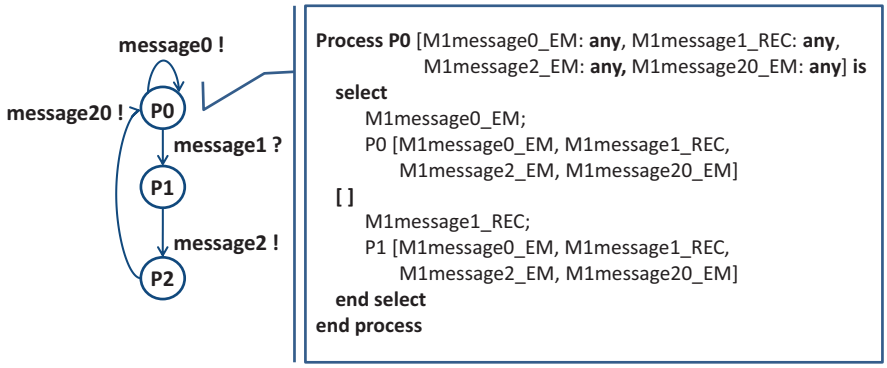

Figure 3: Example of a manager modelled as an LTS (left), and the LNT process encoding the initial state (right)

Figure 3 shows the LTS model of a manager example, as well as the encoding of its initial state P0 into the LNT process algebra. We observe that the process name is the same as the identifier of that state, whereas the process alphabet is composed of all the messages appearing in the LTS, prefixed with the name of the manager (M1) and suffixed by _EM in the case of a send message and by _REC in the case of a receive message. As far as process alphabets are concerned, they consist of all messages appearing in the LTS because, for each state there is a call to the target state. This means that from any state you need in the alphabet all messages reachable from that state, and since in most cases managers are encoded as looping behaviours, we need to have all messages in the alphabet of each process.

Coordination requirements encoding. The coordination requirements specified using reaction rules and regular expressions give an abstract view of the controller to be generated. These requirements are encoded into an LNT process called *coordination*. The process alphabet is composed of all the messages exchanged between the controller and the involved managers, that is, all the 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 (*i.e.,* left hand part and right hand part) using the LNT sequential composition (`;`). In order to make explicit in the controller LTS the logical interval of time that will be chosen at 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`. Those time delimiters are used by the controller to give the pace at which the coordinator reacts. The left hand part of the rule (inside `TBEGIN` and `TEND`) is translated using the `par` operator without synchronization (pure interleaving) since all the messages can be received in any order (see Fig. 4). After the execution of the `TEND` action, the right hand part of the reaction rule is translated using the sequential composition to avoid unnecessary interleavings of actions, which would result in larger LTSs.

As far as the regular expression is concerned, a sequence (.) of rules is encoded using the sequential composition `(;)` and a choice (+) between several rules is translated using the **select** construct. An iteration (∗) is encoded using the **loop** operator as follows, where "..." stands for the translation of the expression on which the operator "*" applies:

```
loop L1 in
      select break L1 [] ... end select
end loop
```

Architecture encoding. In this section, we present how all the participants, *i.e.,* managers and coordination expression, are composed altogether. The communication between them is achieved asynchronously. The coordination expression represents an abstract description of the future controller that we aim to generate, and all the 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 consumes the oldest message from its buffer. When it sends a message to another participant, the message is stored in the input buffer of that participant. LNT functions are used to describe basic operations on these buffers (*e.g.,* addition and retrieval of messages). We present below, an example of function that removes a message from a FIFO buffer (*i.e.,* from the beginning).

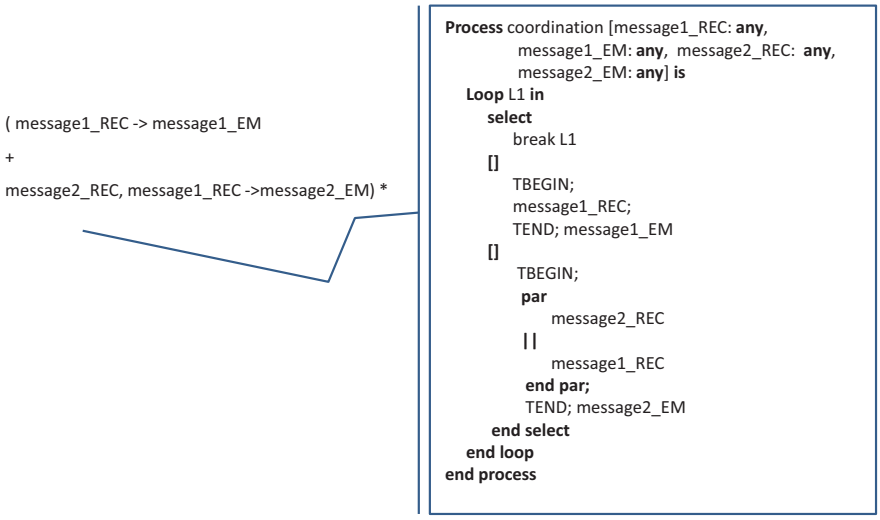

Figure 4: Example of coordination requirements encoded into an LNT process

```
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
```

The function takes as input a buffer (`q`) of type `TBuffer`, which consists of a list of messages (`TMessage`). If the buffer is empty, nothing happens. If it is not empty, the first message is removed.

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 if at some point it must be changed (*e.g.,* after a modification in memory requirements), 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 [5]. As an intermediate solution, one can use the recent results presented in [3] 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 should be used as the minimum buffer bound for both our synthesis techniques and the deployment of the application.

We remind that parallel composition in LNT is described using the `par` construct. This operator consists of a set of messages (m_i below) on which processes (p_j) in parallel must synchronize:

```
par m1, ..., mn in
  p1 [...] || ... || pk [...]
end par
```

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 synchronizes 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 the couple (*coordination*, *buffer*) with all the couples ($manager_i$, $buffer_i$) generated for all the managers. Note that since autonomic managers communicate via the controller, they evolve independently one from another and are therefore composed using the `par` operator without synchronizations. In contrast, the couple (*coordination*, *buffer*) must synchronize with all the other couples on all send messages from the managers/to the buffers, and this is made explicit in the corresponding synchronization set of this parallel composition ($message_p$, .., $message_k$).

```
process main [message1:any, ..., messagen:any] is
   par messagep, ..., messagek in
     couple_buffer_coordination [...]
    ||
     par
       couple_buffer_manager1 [...]
     || ... ||
       couple_buffer_managern [...]
    end par
   end par
end process
```

3.2. Controller Generation and Verification

Generation of the controller LTS. The controller model is an LTS consisting of messages to be consumed from its buffer and messages to be sent to the other participants. The controller LTS also keeps track of the messages received by its own local buffer. This information is useful in some specific cases (non-determinism for instance) in which the controller can execute different actions and, depending on the buffer content, it will decide what behaviour must be executed. By looking at the buffer content, the controller can make the right decision and avoid engaging in a branch that may lead to an erroneous execution of the whole system (impossibility to read some action for example).

Definition 4. *A controller LTS is a tuple* (S, s^0, Σ, T) *where:* S *is a set of states,* $s^0 \in S$ *is the initial state,* $\Sigma = \Sigma^! \cup \Sigma^? \cup \Sigma^B$ *is a finite alphabet partitioned into a set* $\Sigma^!$ *(*$\Sigma^?$*, resp.) of send (receive, resp.) messages and a set of messages received by its buffer* Σ^B*, and* $T \subseteq S \times \Sigma \times S$ *is the transition function.*

Once all the inputs (models and coordination requirements) are encoded into LNT, we can use compilers available in the CADP tools to obtain the controller LTS corresponding to all behaviours of the LNT specification. More precisely, we preserve local send/receive messages from the coordination expression point of view (messages shown in the dashed grey rectangle in Fig. 5) and the receive messages from all the managers. To do so, we hide all message exchanges corresponding to consumptions of the managers from their buffers. All these messages are replaced by internal actions τ. Then, we use the reduction techniques available in CADP for getting rid of internal actions, removing duplicated paths, and determinizing the final LTS. The reductions can be achieved using on-the-fly techniques modulo: (1) strong equivalence for replacing duplicated sequences of transitions by a single sequence and (2) weak trace equivalence for determinizing the generated LTS.

Verification of the generated controller. Since the writing of the coordination expression is achieved manually by a designer, this step of our approach may lead to an error-prone controller. Indeed, the designer can provide a wrong specification of the coordination requirements, which does not

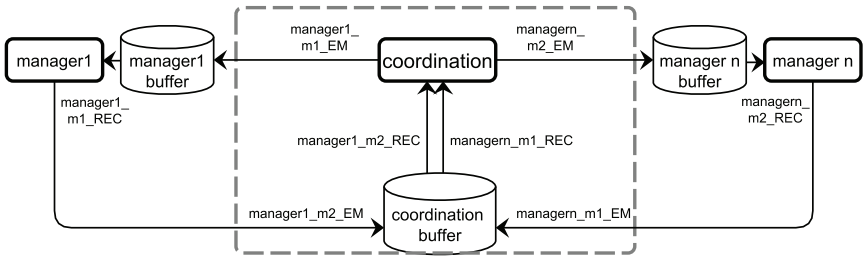

Figure 5: Exchange of messages between the coordination expression and the managers

satisfy the global objectives of all the managers. However, we can take advantage of the encoding into LNT to analyze and validate either the controller LTS alone (and thus the coordination expression) or the LTS corresponding to the final system (*i.e.,* the composition of all the participants). To do so, one can use the CADP model checker (Evaluator), which takes as input an LTS model and temporal properties specified in MCL [22]. We distinguish two types of properties:

1. properties which depend on the application, *e.g.,* the controller must eventually transmit a specific message to a certain manager;

2. properties which do not depend on the application, *e.g.,* checking the absence of deadlock.

We would like to recall that actions `TBEGIN` and `TEND` are parts of both the controller LTS and of the LTS corresponding to the whole system. Therefore, one can also verify properties using those specific actions. For instance, one can formalize in the MCL logic properties such as "*there is never two actions A and B in sequence without an action TEND in-between*" or "*an action C is always systematically followed by an action TEND*". Those kinds of properties are useful if the designer wants to check that some situation will never occur (safety properties) or will eventually occur (liveness properties).

Another source of error may occur when one action appears unexpectedly between `TBEGIN` and `TEND`. In that case, no reaction rule would match the exact set of events, resulting in a deadlocking behaviour. This erroneous case can be detected using verification techniques by checking for the presence of deadlock in the controller LTS. Once detected, the solution for solving this

problem is to add a reaction rule to the coordination expression for handling this specific case.

4. Java Code Generation and Deployment

We present in this section our techniques for automatically generating the Java code, which corresponds to the controller LTS obtained during the synthesis phase (see Section 3.2). This Java code is generated without human intervention and allows one to deploy controllers in the context of real-world applications as we will show in Section 5.

4.1. Java Code

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` representing 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 additional 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 triggers successive moves in the controller according to the current state of the controller and to the received messages. This method computes the messages that the controller has to send as a 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. The choice of this period cannot be made during the synthesis phase and is achieved just before deployment. This period is computed using sampling techniques and implemented using the `sleep` method in the LTS class introduced previously. By sampling techniques, we mean simulation and empirical study in order to determine the best period of time with respect to the application being monitored. The choice of this

period mainly depends on the occurrence rate of events raised by the system. The goal of this step is to find the best trade-off between stability and reactivity. To put it another way, the more unstable the system is, the shortest the period of time should be. If two events appear one after the other very closely but on two periods of time whereas we would like to catch them on a same period, we will miss the problem and eventually add then remove an unnecessary server for instance. However, these boundary cases do not occur frequently if the period is chosen as explained above and for those cases our approach converges quickly to a stable situation. Convergence is ensured if the coordination expression also takes the original automomic manager behaviours into account, that is, allows the managers to behave as originally designed without coordination. In that case, the system will work, not optimally from a reconfiguration point of view (possible addition of unnecessary servers for instance), but it will make the required modification imposed by the (uncoordinated) managers.

The main behaviour of the controller (`run` method) consists of an infinite reactive loop, which successively receives events from the application and computes reactions (messages to be sent by the controller). The result produced by each call of the `run` method (*i.e.,* controller reactions) has to be interpreted and executed. Therefore, all the messages returned by that method are encoded as events too. A part of the Java program is dedicated to handling the events raised by the application by converting them into the input format of the `react` method, and conversely handling the output of the `react` method by translating it into a list of events executed by the system. Each event contains the corresponding message and additional information, for instance, a failure event also has as parameter the impacted server and further information (identifier, port, etc.). Therefore, all the messages received/emitted by the generated controller are related to the corresponding events. As far as asynchronous communication is concerned, different existing software can be used, such as RabbitMQ, which is a classic industrial solution for implementing message queuing interactions.

5. A Multi-tier Application Supervised by Autonomic Managers

We introduce in this section a JEE multi-tier application supervised by two sorts of autonomic managers, namely a self-repair and a self-sizing manager. We illustrate our approach on this example, by presenting successively

the application, the coordination requirements, the encoding into LNT, the controller model generation, and the final deployment of the controller.

5.1. Multi-Tier Application

The JEE multi-tier application (Fig. 6) consists 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 the SQL queries to a tier of replicated MySQL databases fairly.

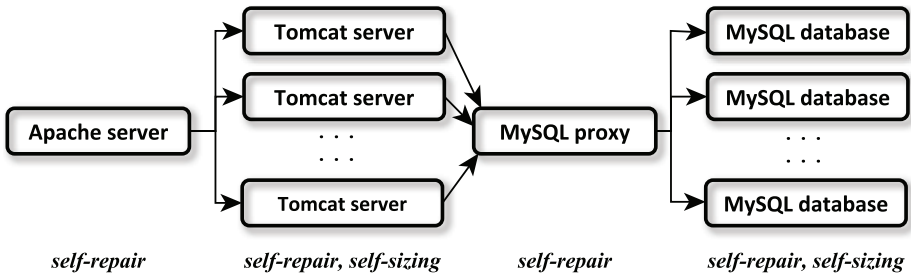

Figure 6: A multi-tier application

5.2. The Managers Role

The dynamic sizing plays an important role in the load balancing and energy consumption of the replicated servers on this type of application. When starting the application, it is difficult to predict the number of requests and to estimate the number of required servers. Therefore, adjusting dynamically the number of replicated servers according to the number of requests is necessary. Moreover, once a server fails, the failure must be immediately detected and repaired in order to preserve the application consistency. Managers are used to optimize the load balancing and preserve the application consistency. To sum up, a self-sizing manager handles the dynamic sizing of the application and a self-repair manager is in charge of detecting failures and repairing them.

Figure 7 shows how the architecture of the managers is based on the `MAPE-K` model (Monitor Analyse Plan Execute - Knowledge). For instance,

the self-sizing manager continuously observes a load of replicated servers through the Monitor function. It computes the average of the load and detects the presence of an overload or an underload through the Analyze function. Once an overload (underload, resp.) is detected, the manager makes a decision about the addition or removal of a server through the Plan function. This decision is executed by the Execute function.

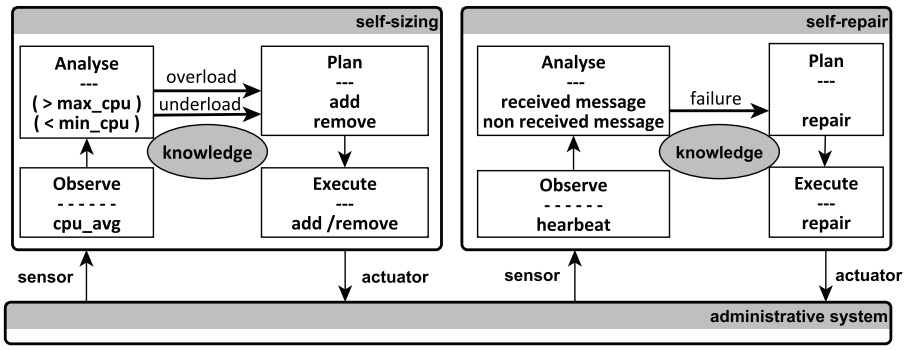

Figure 7: Architecture of the self-sizing (left) and of the self-repair (right) manager

5.3. The Managers Models

We describe the managers behaviours 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. 8, 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 generated controller aims at coordinating the messages transmitted from the Analyze function to the Plan function and from the Plan function to the Execute function.

The self-sizing manager. The self-sizing is in charge of adapting the number of replicated servers dynamically by sending the message `add!` (`remove!`, resp.) to the system when detecting an overload (underload,

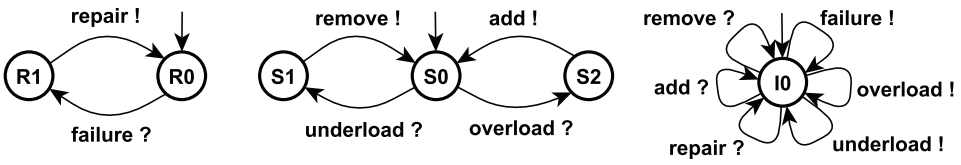

Figure 8: (left) Self-repair manager LTS, (middle) Self-sizing manager LTS, (right) Application manager LTS

resp.). The overload (underload, resp.) is detected when the average of the load exceeds (is under, resp.) a maximum (minimum, resp.) threshold (Fig. 8, middle). We associate one instance of the self-sizing manager to the Tomcat servers and another instance to the MySQL databases. We distinguish these two instances of the same manager as follows: both models have the same states and transitions as the LTS shown in Figure 8 (middle), but suffixed by `_tc` (`_mq`, resp.) when the instance is associated with the Tomcat servers (MySQL databases, resp.).

The self-repair manager. The self-repair manager detects failures and requires the creation of new instances of the failed servers (Fig. 8, left). We have four instances of the self-repair manager, one per tier. Therefore, we consider here that we have four managers. All the models have the same states and transitions as the LTS shown in Figure 8 (left), but suffixed by `_ap` (`_tc`, `_pq`, `_mq`, resp.) when the instance is associated to the Apache server (Tomcat servers, MySQL proxy server, MySQL databases, resp.).

5.4. Coordination Problems

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 or removing a server that is needed as a result of a server failure. We distinguish two types of problems: those occurring in the same tier (*i.e.,* the replicated servers) and those occurring in different tiers.

Coordination problems in the same tier. A failure of a server in a set of replicated servers triggers an overload of the remaining servers. When the self-repair manager receives a message indicating the detection of this failure, it sends a message to the application manager requesting it to repair the failure by creating a new replica. Before completion of this repair phase, the other replicated servers receive more requests than before the failure, which causes an overload. Upon reception of this overload detection from

the application manager, the self-sizing manager sends a message back asking the addition of another server. In this scenario, as a result of a server failure, two new servers are added to the application whereas one was enough.

Coordination problems in different tiers. A failure of a server, which is hosted on the first tier and connected to other servers hosted on another tier, triggers an underload in the second tier. When the self-repair manager receives a message indicating the detection of this failure, it sends a message to the application manager requesting it to repair the failure by creating a new replica. Before completion of this reparation, the servers hosted on the second tier receive fewer requests than before the failure, which causes an underload. Upon reception of this underload detection from the application manager, the self-sizing manager sends a message back calling for the removal of a server. Then, once the failed server is repaired, the servers hosted on the second tier receive more requests than before the server reparation, which causes an overload and therefore the addition of another server by the self-sizing manager. Therefore two unnecessary reconfiguration operations (removal and addition of a server) are executed as a result of a server failure in that example.

5.5. Managers Coordination

We present below an excerpt of the requirements for the controller we want to generate for our example. These rules ensure that all the 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 the 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_ap?    -> repair_failure_ap!                          (❶)
+ app_failure_tc?    -> repair_failure_tc!                          (❷)
+ app_overload_tc?   -> sizing_overload_tc!                         (❸)
+ app_underload_tc?   -> sizing_underload_tc!                       (❹)
+ app_failure_mysql? -> repair_failure_mysql!                       (❺)
+ app_failure_px?    -> repair_failure_px!                          (❻)
+ app_failure_ap?, app_underload_tc? -> repair_failure_ap!          (❼)
+ app_failure_tc?, app_overload_tc?  -> repair_failure_tc!          (❽)
+ ...  ) *
```

We distinguish two kinds of rules:

1. those where a unique message appears in the left part of the reaction rule (see, *e.g.,* ❶, ❷). In that case, the corresponding controller transfers that message to the manager;

2. those encoding the coordination we want to impose on managers, *e.g.,* rule ❽ 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 the rules can be fired at any time (combination of + and * operators).

5.6. Encoding into LNT

Let us first show some excerpts of LNT obtained when calling our LNT code generator on this example. We first show the LNT processes encoding an instance of the repair manager, which handles the tier of the Tomcat servers. The first process has the same name as the initial state identifier `R0_tc`. The process alphabet is the set of labels used in this manager LTS (`repair_failure_tc_REC` and `repair_repairing_tc_EM`). Each message is prefixed with the manager name. The body of the first process, for instance, consists of a failure receive message (`repair_Failure_tc_REC`) followed by a call to the process encoding the target state, that is, `R1_tc`.

```
process R0_tc [repair_failure_tc_REC: any,
               repair_repairing_tc_EM: any] is
     repair_failure_tc_REC;
     R1_tc [repair_failure_tc_REC, repair_repairing_tc_EM]
end process

process R1_tc [repair_failure_tc_REC: any,
               repair_repairing_tc_EM: any] is
      repair_repairing_tc_EM;
      R0_tc [repair_failure_tc_REC, repair_repairing_tc_EM]
end process
```

We show now an example of process encoding a couple *(manager, buffer)*, particularly the couple corresponding to the repair manager handling the

tier of the Tomcat servers. This manager synchronizes with its buffer on the repair_failure_tc_REC message, which is emitted by the buffer and received by the manager. Note that the buffer process (buffer_repair_tc) 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_tc [repair_failure_tc_REC: any,
      repair_repairing_tc_EM: any, repair_failure_tc_EM: any] is
  par repair_failure_tc_REC is
    R0_tc [repair_failure_tc_REC, repair_repairing_tc_EM]
  ||
    buffer_repair_tc [repair_failure_tc_EM, ...] (nil)
  end par
end process
```

After generating all the couples (autonomic manager/buffer, application manager/buffer, and controller/buffer), the main process is encoded using parallel compositions. The managers do not interact directly together. Therefore, the couples *(manager, buffer)* are translated using the par construct without synchronization (pure interleaving). All the managers communicate together through the controller. These interactions are expressed using another parallel composition where the synchronization set makes explicit all the messages sent by the managers to the controller buffer, or sent by the controller to the managers buffers.

```
process main [repair_failure_ap_REC:any, ...] is
   par sys_failure_ap_EM, sys_failure_tc_EM,  ... in
     (* couple coordination/buffer  *)
        couple_buffer_coordinaiton [...]
   ||
     par
       (* couple application manager/buffer *)
        couple_buffer_AM [...]
     ||
     (* couple repair manager/buffer for the Apache server*)
        couple_buffer_repair_ap [...]
     ||
     (* couple repair manager/buffer for the Tomcat servers*)
        couple_buffer_repair_tc [...]
     ||
```

```
    (* couple repair manager/buffer for the MySQL proxy server*)
       couple_buffer_repair_px [...]
    ||
    (* couple repair manager/buffer for the MySQl databases*)
       couple_buffer_repair_mq [...]
    ||
    (* couple sizing manager/buffer for the Tomcat server *)
       couple_buffer_sizing_tc [...]
    ||
    (* couple sizing manager/buffer for the MySQL databases *)
       couple_buffer_sizing_mq [...]
  end par
 end par
end process
```

5.7. Controller Generation and Verification

CADP compilers take as input the specification corresponding to the LNT encoding and compute as output the corresponding LTS describing the whole system for our example. Then, we apply hiding and minimization techniques on that LTS to generate the LTS corresponding to the controller. The final controller LTS consists of 28,992,305 states and 46,761,782 transitions. An excerpt of the controller LTS, which focuses on the failure and overload detection of a Tomcat server in the same period of time, is shown in Figure 9. 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 use the Evaluator model checker to verify temporal properties expressed in the MCL logic. For illustration purposes, we present two examples of liveness properties. The first one is checked on the controller LTS and the second one holds on the LTS of the whole system:

- The reception of a failure message by the controller is eventually followed by the sending of a repair message to the application manager in order to request it to repair the Tomcat server

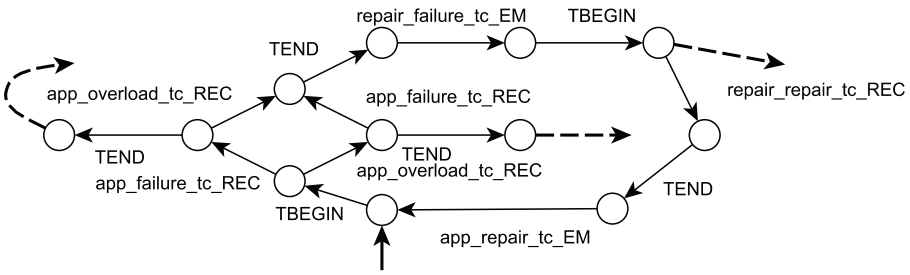

Figure 9: Excerpt of the controller LTS for the example

```
[true* .app_failure_tc_REC] inev (app_repair_tc_EM)
```

- 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, but not both messages

  ```
  [true* .app_overload_tc_EM]
         inev (app_repair_tc_EM xor app_add_tc_EM)
  ```

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

Both properties use the macro `inev (M)`, which is used especially when specifying liveness properties that are inevitability assertions. Inevitability assertions can be expressed using fixed point operators that indicate that a transition labelled with `M` eventually occurs. The `inev` 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

|Managers|	Whole system LTS		Controller LTS		Time
	|states|	|transitions|	|states|	|transitions|	(m:s)
2	2,424	12,518	1,124	5,140	0:10
3	103,725	365,845	9,744	31,636	2:39
4	145	267	38	44	0:06
5	10,063,873	39,117,110	85,297	621,482	389:65
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

system as well as the controller LTS obtained after minimization (*wrt.* strong and weak trace relations). The last column gives the overall time to synthesize the controller.

Let us now shortly comment on the results presented in Table 1. First of all, when using acyclic managers (see, *e.g.*, the 3^{rd} example in Table 1), the size of the generated controller LTSs and the time required for computing those LTSs are very small. In contrast, we observe that these numbers grow importantly for other examples (gray lines). In those cases, the increase in LTS size and computation time is due to the presence of looping behaviours in manager models (as those presented in Figure 7). Since we rely on enumerative techniques for generating LTSs, looping behaviours induce more possible combinations of correct executions, resulting in larger LTSs than for acyclic models. We also see that LTS sizes and generation times increase with the number of managers in parallel (see, *e.g.*, the last line of Table 1).

5.8. Deploying and Running the Generated Controller

In this section, we present some experiments we performed when deploying and running our controller for the multi-tier application introduced previously (see Fig 6). To do so, we use a virtualized experimental platform based on Openstack[2], which consists of six physical machines on which we instantiate virtual machines with 1 vCPU, 2GB of memory and 8GB of disk.

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

[2]https://www.openstack.org/

using a dynamic management protocol allowing us to connect and start the involved servers and database in the right order [1]. In a second step, we use the Apache JMeter application 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 various 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. We made long-running experiments for validation of our approach where we ran the application for hours with random injection of failures.

Figure 10 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 10 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, and the controller forwards only the failure detection to the repair manager. Accordingly, a single MySQL 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 reconfiguration operations, that is, the unnecessary addition/removal of servers/databases. Figure 11 shows experimental results obtained with different numbers of failures. For instance, we see that when injecting 14 failures to our running application, the controller applies 18 reconfiguration operations on the system, *i.e.*, 18 addition or removal of

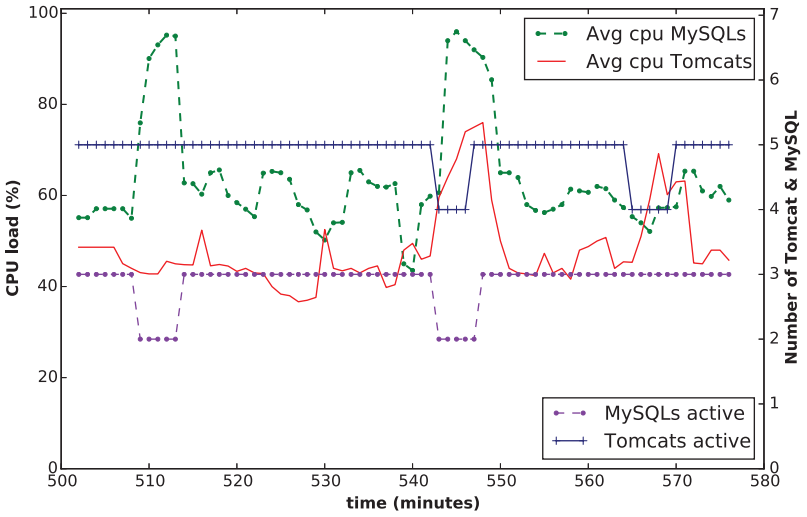

Figure 10: Tomcat and MySQL failure/overload in a coordinated environment

servers/databases. Without coordination, the number of reconfigurations for this scenario increases to 40. This means that 22 undesired reconfiguration operations have been avoided by using a controller-based approach.

The results presented in Figure 11 were obtained by running one simulation for each number of failures (3, 6, 10, 14, etc.). This figure shows that the number of reconfiguration operations decreases importantly by using our controller-based solution. Without coordination, the number of reconfigurations is at least twice larger. However, in the coordinated case, the number of unnecessary reconfiguration operations is not completely removed and is therefore not optimal. Indeed, there are situations when there are several failures in a row or when messages arrive on two periods of time as discussed in Section 4. In those cases we cannot avoid unnecessary reconfiguration operations and this explains the variation in our results for instance between 30 failures (about 30 reconfiguration operations in the coordinated case) and 50 failures (about 70 reconfiguration operations in the coordinated case).

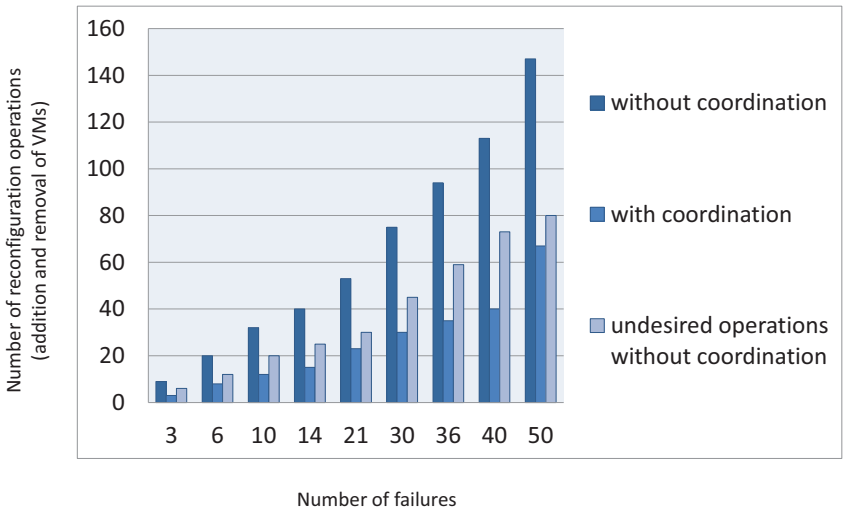

Figure 11: Number of reconfiguration operations with/without coordination and number of undesired operations avoided by coordination

6. Related Work

Controller synthesis techniques. Controller synthesis for discrete event systems was originally introduced by Ramadge and Wonham [29, 27]. In [29], 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. [27] proposes a supervisor synthesis algorithm for automatically generating 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 [12], 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 [28]. 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.

Coordination of autonomic managers. In [25], the authors propose a generic integration model that focuses, first, on categorizing all autonomic loops in 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.

[24] presents a framework for the coordination of multiple autonomic managers in the cloud computing context. Managers are classified in two categories: application-related and infrastructure-related managers. 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 to our work is that this paper focuses on quality of service whereas we provide controller generation techniques considering behavioural and functional aspects of the system execution.

Other recent works [11, 17, 18] 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 we focus on asynchronous systems and assume that communication is achieved asynchronously. Our approach is thus tailored for these hypotheses.

Configuration of cloud applications. [10] 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. Beyond Aeolus, other frameworks have attempted to provide deployment and reconfiguration algorithms for distributed cloud applications, see, *e.g.*, [21, 15, 14, 13].

In [6], 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.

Coordination of component-based systems. In [26], 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.

[4] presents an approach relying on the GCM/ProActive execution environment, where applications are composed of distributed components communicating by asynchronous requests with futures. The components take adaptation decisions and evolve in an autonomic way employing features designed to describe their behaviours. This approach relies on loosely coupled components, which communicate asynchronously by message sending and share references for the futures. The two main differences between this work and ours is in the communication model and in the way control/reconfiguration is implemented ([4] relies on the notion of *membrane* to do so).

7. Conclusion

We have focused in this article on the coordination of several autonomic managers in the context of cloud applications. We have proposed in this article new asynchronous synthesis techniques for generating a controller allowing the monitoring and orchestration of such autonomic managers. These techniques take as input a set of abstract behavioural models for the managers as well as the coordination requirements described as reaction rules. Then, the synthesis process works in two successive steps. We first generate a controller model using as intermediate step a process algebra encoding. Second, we rely on Java code generation techniques for deploying the generated controller in order to coordinate real-world applications. The generated and deployed controller interacts asynchronously with the managers via FIFO buffers and allows them to take globally coherent decisions.

This article shows how the approach covers the whole development process from an expression of the requirements to the final implementation and 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.

As far as future is concerned, we plan first to extend our coordination language in order to specify concurrent flows (*e.g.,* by means of fork/join operators), which are quite common in orchestration and distributed coordination. Furthermore, our coordination expressions cannot describe specific requirements (*e.g.,* elastic behaviours with oscillations or cool-down periods). Expressing this type of needs requires the extension of our coordination language and LTS models with data-awareness and real-time constraints. The addition of such numerical values and constraints may, however, result in larger state spaces, especially when addressing more complicated applications with an important number of managers. To tackle the state space explosion problem and to be able to handle larger systems, we plan to integrate alternative generation and model checking techniques, such as symbolic approaches with OBDDs and partial order reductions [9]. Another idea to reduce the size of the generated controller is to adopt a distributed coordination solution (divide and conquer). The main advantage of this distribution is that every controller is much smaller in terms of states and transitions compared to a unique centralized controller. This generation is however not that easy be-

cause those distributed controllers need to interact together at certain points in time in order to apply the coordination requirements in a consistent way from a global point of view. We plan to add these synchronization points at the process algebra encoding level.

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] R. Abid, G. Salaün, N. De Palma, and S. Mak Karé Gueye. Asynchronous Coordination of Stateful Autonomic Managers in the Cloud. In Proc. of FACS'15, volume 9539 of LNCS, pages 48–65. Springer, 2015.

[3] L. Akroun, G. Salaün, and L. Ye. Automated Analysis of Asynchronously Communicating Systems. In Proc. of SPIN'16, volume 9641 of LNCS, pages 1–18. Springer, 2016.

[4] F. Baude, L. Henrio, and C. Ruz. Programming Distributed and Adaptable Autonomous Components - The GCM/ProActive Framework. Softw., Pract. Exper., 45(9):1189–1227, 2015.

[5] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. J. ACM, 30(2):323–342, 1983.

[6] 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.

[7] 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.

[8] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, Asynchronous, and Causally Ordered Communication. Distributed Computing, 9(4):173–191, 1996.

[9] E.M. Clarke, W. Klieber, M. Novek, and P. Zuliani. Model Checking and the State Explosion Problem. In B. Meyer and M. Nordio, editors, Tools for Practical Software Verification, volume 7682 of LNCS. Springer, 2012.

[10] R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Aeolus: A Component Model for the Cloud. Inf. Comput., 239:100–121, 2014.

[11] 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.

[12] G. Delaval, H. Marchand, and E. Rutten. Contracts for Modular Discrete Controller Synthesis. In Proc. of LCTES'10, pages 57–66. ACM, 2010.

[13] F. Durán and G. Salaün. Robust and Reliable Reconfiguration of Cloud Applications. Journal of Systems and Software, 122:524–537, 2016.

[14] 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.

[15] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A Deployment Management System. In Proc. of PLDI'12, pages 263–274. ACM, 2012.

[16] 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.

[17] 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.

[18] 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.

[19] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.

[20] J. E. Hopcroft and J. D. Ullman. Introduction to Automata Theory, Languages and Computation. Addison Wesley, 1979.

[21] C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling Dynamic Reconfigurations in Reo Using High-Level Replacement Systems. Sci. Comput. Program., 76(1):23–36, 2011.

[22] 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.

[23] R. Milner. Communication and concurrency. Prentice Hall, 1989.

[24] 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.

[25] 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.

[26] 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.

[27] P.J.G. Ramadge and W.M. Wonham. The Control of Discrete Event Systems. Proc of the IEEE, 77(1):81–98, 1989.

[28] 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.

[29] 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.