Formal Methods for the Development and Verification of Autonomic IT Systems

Radu Calinescu¹, Shinji Kikuchi² and Marta Kwiatkowska³

¹Aston University and Oxford University Computing Laboratory, UK
²Fujitsu Laboratories Limited, Japan
³Oxford University Computing Laboratory, UK

Abstract

This chapter explores ways in which rigorous mathematical techniques termed formal methods can be employed to improve the predictability and dependability of autonomic computing. Model checking, formal specification and quantitative verification are presented in the contexts of conflict detection in autonomic computing policies, and of implementation of goal and utility-function policies in autonomic IT systems, respectively. Each of these techniques is illustrated using a detailed case study, and analysed to establish its merits and limitations. The analysis is then used as a basis for discussing the challenges and opportunities of this endeavour to transition the development of autonomic IT systems from the current practice of using ad-hoc methods and heuristic towards a more principled approach.

Introduction

The development of IT systems with self-managing capabilities – termed autonomic computing – is a relatively young area of research (Kephart & Chess, 2003; Murch, 2004). Now past the period of initial hype characteristic of any major new paradigm in computer science, autonomic computing looks set to become an established approach to addressing the continual increase in the scale and complexity of today’s IT systems. There are numerous indicators of this trend, including the emergence of generic development platforms for autonomic computing (Calinescu, 2009a; Garlan, Schmerl, & Cheng, 2009; Twidle, Dulay, Lupu, & Sloman, 2009; Vassev & Mokhov, 2009) and the use of autonomic IT systems across a wide range of application domains (Huebscher & McCann, 2008; Parashar & Hariri, 2006).

While this healthy pace of progress is well in line with the ambitious plan put forward by the autonomic computing manifesto (IBM Corporation, 2001), one concern remains. The vast majority of autonomic IT systems – whether under test in research labs or deployed in production – implement the high-level objectives that guide their operation using heuristics derived from and validated through a combination of experimentation, simulation and testing. As it is well known from more established areas of computer science, this is insufficient for developing IT systems that are highly predictable and dependable. Yet, autonomic IT systems are required to excel in precisely these characteristics (Dai, 2005; Sterritt, 2003; Sterritt & Bustard, 2003).

This chapter proposes that the major problem identified above is addressed by using rigorous mathematical techniques termed formal methods (Boca, Bowen, & Siddiqi, 2009). Building on the authors’ previous work in the area (Calinescu & Kwiatkowska,
2009a, 2009b; Kikuchi, Tsuchiya, Adachi, & Katsuyama, 2007), the chapter explores ways in which formal methods can help overcome the discrepancy between what autonomic IT systems can deliver in terms of predictability and dependability, and what is expected of them. The next three sections look in turn at several aspects of autonomic computing that can benefit from the use of existing or enhanced techniques from the area of formal methods.

First, next section describes the use of model checking (E. M. Clarke, Grumberg, & Peled, 2000) to detect policy conflicts in autonomic computing systems. Given the potentially significant damage that conflicting policies can cause to autonomic systems, it is critical to ensure that policies expressing different system objectives do not interfere with each other. This section explains why model checking represents a better-suited technique for conflict detection than alternative approaches such as testing or simulation.

This is followed by two sections that present formal approaches to implementing two important classes of autonomic computing policies: goal policies and utility-function policies. Goal policies describe constraints that an autonomic system needs to observe at all times. Together, these constraints provide a formal specification (Abrial, 1996; Woodcock & Davies, 1996) for the system, and a technique termed model synthesis (Jackson, 2006) can be employed to update the system configuration in response to changes in its environment in ways that comply with this specification. This approach is presented in detail in the chapter. Utility-function policies provide a quantitative measure of the degree to which an autonomic system achieves its high-level objective, and request that the system configuration is adjusted automatically so that maximum utility is obtained in the presence of changes in the system state, workload and environment. The chapter describes the use of quantitative verification (Kwiatkowska, 2007) to implement this type of autonomic computing policy.

The final section concludes the chapter with a brief summary, and discusses the directions in which the field of formal methods for autonomic computing is headed. This discussion includes overviews of two emerging platforms that aim to employ formal techniques for the end-to-end development of autonomic IT systems. One development platform involves the model-driven, automatic generation of many of the components of an autonomic IT system starting from a quantitative model that defines the behaviour of its components. The other platform requires that the development process starts with the definition of a formal specification of the autonomic system, and generates its parts in a series of automated development steps.

The intended readership for the chapter includes engineers, scientists, practitioners and researchers interested in transitioning from the current practice of using ad-hoc methods and heuristics in their autonomic computing work towards a principled, rigorous approach to autonomic IT system development.

Conflict detection in autonomic computing policies

Autonomic systems are defined as systems that "manage themselves according to an administrator's goals" (Kephart & Chess, 2003). The effectiveness of the self-management depends on the quality of the autonomic computing policies used to express these goals. Poorly-defined policies lead to ineffective self-management; conflicting policies can be downright damaging to the autonomic system. This explains the significant research effort dedicated to detecting policy conflicts in all types of policy-based management systems, including autonomic IT systems.

This section describes how a formal technique termed model checking can be used to detect conflicts in autonomic computing policies. Note that this approach uses model checking to verify the correctness of self-management policies specified by the
administrator of an autonomic IT system rather than to specify or implement such policies. We start by introducing model checking and explaining why it is better suited for detecting conflicts in autonomic computing policies than heuristic-based testing approaches. We then describe an approach to model checking autonomic computing policies, and illustrate its application to a case study from the area of data-centre resource management. We conclude with a summary of related work.

Background

*Model checking* (E. M. Clarke et al., 2000) represents a formal technique for verifying whether a system satisfies its specification. The technique involves building a mathematically-based model of the system behaviour, and checking that system properties specified formally in a temporal logic hold within this model. For each refuted property, the technique yields a counterexample consisting of an execution path for which the property does not hold. The result is based on an exhaustive analysis of the state space of the considered model - a characteristic that sets model checking apart from complementary techniques such as testing and simulation. Sophisticated algorithms have been devised over the past two decades to make possible the verification of ever larger systems without individually examining every single state of their model. Software tools that implement these algorithms are called *model checkers*, and are available both commercially and as free, open-source applications.

The system model most commonly used in model checking is termed a *Kripke structure* (E. M. Clarke et al., 2000). It consists of a state transition graph 

\[ M = (S, S_0, R, L) \]

where \( S \) represents the finite set of states in which the system can exist, \( S_0 \subseteq S \) is the set of the initial states, \( R \subseteq S \times S \) is a relation that defines all possible transitions between states, and \( L : S \rightarrow 2^{AP} \) is a labelling function that labels each state with the set of atomic propositions that are true in that state.

Commonly used temporal logics include *linear temporal logic* (LTL) (E. Clarke & Lerda, 2007) and *computation tree logic* (CTL) (E. M. Clarke, Emerson, & Sistla, 1986). The approach to verifying autonomic computing policies described in this section uses LTL, which is a logic that adds the temporal operators in Table 1 and calculation for them to first-order logic. An LTL formula such as \( \phi \) and \( \psi \) in this table is a combination of atomic propositions, logical operators (i.e., \( \neg \), \( \land \), and \( \lor \) ) and the LTL temporal operators \( \diamond \), \( [] \), \( \diamond \), \( U \) and \( R \). Given a Kripke structure \( M = (S, S_0, R, L) \) and an LTL formula \( \phi \), the notation \( M \models \phi \) is used to state that the system model \( M \) satisfies the LTL formula \( \phi \). For example, if \( a \in AP \) is an atomic proposition, \( M \models [](\diamond a) \) states that \( a \) is eventually true on the every path from each state \( s \in S \).

Numerous model checkers have been developed in recent years, and employed in application domains ranging from circuit design and security protocol analysis to mission-critical system verification. Some of the most effective and widely used model checkers include NuSMV (Cimatti, Clarke, Giunchiglia, & Roveri, 1999), UPPAAL (Bengtsson, Larsen, Larsson, Pettersson, & Yi, 1995) and SPIN (Holzmann, 2003). The model checker used to illustrate the verification of autonomic computing policies in this section is SPIN, a tool developed at Bell Labs, and winner of a 2001 ACM System Software Award. SPIN can be downloaded freely from [http://spinroot.com](http://spinroot.com).
Our approach to verifying autonomic computing policies requires that two types of information are available for the autonomic system:

- A **structural model** that specifies the system parameters that need to be monitored or controlled for the considered application, and their value domains. Consider, for example, an autonomic computing application that self-configures the number of servers allocated to services running within a data centre. In this example, the monitored system parameters to include in the structural model are the total number of servers in the data centre and the workloads of the services. The controlled parameters (i.e., the parameters adjusted by the self-managing system) are the numbers of servers allocated to individual services; these parameters and their value domains (i.e., the range of values they are allowed to take) have to be included in the structural model. Note that the information provided by the structural model defines the possible states of the system.

- A **performance model** that defines the relationships between the system parameters defined in the structural model, and between these parameters and any internal parameters that the system may have. For instance, for the data-centre autonomic application described above, the performance model will include the relationship between the number of servers allocated to a service and the maximum workload that the service can handle without violating its service-level agreement. We assume that these properties are obtained using a method such as benchmarking. Note that the information provided by the performance model specifies properties associated with different states of the system.

The sets of autonomic computing policies that can be verified using the method described in this section comprise two classes of policies:

- **Action policies** (also termed operation rules), i.e., "if condition then action"-style rules defining the actions the system can take to change its configurations and the set of conditions under which these actions may be taken. Note that these policies specify the possible transitions between system states.

- **Goal policies**, which express invariants that the system should fulfil at all times, or "final state" conditions that the system must comply with after the operation rules are executed.
The system information and the set of policies presented so far are used to derive a Kripke structure and a set of LTL formulas that must be satisfied by this structure in order to ensure that the autonomic computing policies are conflict free. The steps involved in building this Kripke structure $M = (S, S_0, R, L)$ and the LTL formulas are depicted in Figure 1 and detailed below:

1. The set of system states $S$ and the set of initial system states $S_0 \subseteq S$ are derived from the structural model of the autonomic system.
2. The labelling function $L : S \rightarrow 2^{AP}$ is extracted from the performance model for the system. As illustrated by the case study presented later in this section, the atomic propositions correspond to key parameter values associated with specific system states.
3. The state transition relation $R \subseteq S \times S$ is defined by the operation rules (i.e., by the action policies) for the system.
4. Finally, system constraints specified by a goal policy correspond to LTL formulas $\phi$ that model $M$ must always satisfy: $M \models []\phi$. Likewise, the final-state conditions expressed by goal policies correspond to LTL formulas $\psi$ that model $M$ must “eventually” satisfy: $M \models \Diamond \psi$.

The assertions $M \models []\phi$ and $M \models \Diamond \psi$ obtained in step 4 are verified using a standard LTL model checker. If these assertions are true, then the policy set is conflict free, i.e., its action policies take the system from any initial state to a valid final state, transitioning only through intermediate states that satisfy the invariants specified by the constraint goal policies. If one or more assertions are not true, the policy set contains conflicts. The counterexample generated by the model checker for each such assertion can be used to identify reachable states that do not comply with the system invariants and/or unreachable final states, and thus represent a starting point for resolving the policy conflict.
Case study

To illustrate the application of model checking to conflict detection in autonomic computing policies, we consider a case study involving the on-demand resource allocation within the autonomic data centre in Figure 2. This data centre handles user requests using a standard two-tier architecture: a web server, front-end tier and an application tier. Load balancers are deployed in each tier to distribute its workload among the servers running within that tier. Other than this “primary system” that is actively handling user requests, the data centre comprises a “spare system” that contains spare, standby servers.

As the data-centre workload increases, the primary system can be reinforced by adding standby servers to one or both of its tiers. Conversely, if the request rate reduces, active servers can be removed from the two tiers of the primary system and placed into the spare system, thus reducing the operational cost for the data centre without impacting its performance. An appropriate set of policies is necessary to ensure that the autonomic data centre provides the required level of service with minimal cost, by appropriately adding or removing servers from the primary system based on the rate of user requests and on the capacity of the servers. A set of autonomic computing policies that aims to achieve this objective is proposed later in this section, and model checking is used to verify its correctness.

Structural model and performance model We start by defining the structural and performance model for the system below.

(1) Structural model. The monitored and controlled data-centre parameters relevant for this case study are:

- The request rate \( x > 0 \) that the system should accommodate;
- The number of spare servers \( A_i \) allocated to tier \( i \), \( 1 \leq i \leq 2 \).

We will assume that \( 0 \leq x \leq 300 \) requests/second; and that in the initial state \( A_1 = A_2 = 0 \).
Performance model. The data-centre comprises two sets of “internal” parameters (i.e., parameters that are neither monitored nor controlled, but are calculated based on the parameters defined in the structural model):

- The total number of servers \( N_i > 0 \) in tier \( i \) of the primary system, \( 1 \leq i \leq 2 \);
- The processing time \( T_i > 0 \) required to handle a user request in tier \( i \), \( 1 \leq i \leq 2 \).

The set of relationships between all the system parameters comprise:

- The relationship between the total number of servers and the number of spare servers in the two tiers:
  \[ N_1 = 2 + A_1 \quad \text{and} \quad N_2 = 1 + A_2. \]
- The relationship between the tier-one average processing time \( T_1 \), the request rate \( x \) and the number of servers in the first tier \( N_1 \). This is derived assuming an M/M/1 queuing model for this tier:
  \[ T_1 = \frac{\alpha}{1 - \frac{\alpha}{N_1}}, \]
  where \( \alpha > 0 \) represents the service rate of a single server.
- The relationship between the tier-two average processing time \( T_2 \), the request rate \( x \) and the number of servers in the second tier \( N_2 \). This is assumed to be proportional to the ratio \( x/N_2 \):
  \[ T_2 = \beta \cdot \frac{x}{N_2} + \gamma, \]
  where \( \beta \) and \( \gamma \) are two positive constants.

Autonomic computing policies The set of autonomic computing policies for the data centre comprise the policies described below.

(1) Action policies. Two operation rules express the conditions in which the system configuration is modified:

- ADD Rule: If \( T_1 + T_2 \geq 0.6 s \), then add a spare server to tier 1 if \( T_1 \geq T_2 \) or add a spare server to tier 2 if \( T_1 < T_2 \).
- REMOVE Rule: If \( T_1 + T_2 < 0.25x \) and \( A_1 + A_2 > 0 \), randomly select a spare server that is in use, remove it from the primary system and return it to the spare system.

(2) Goal policies. The invariant and “final state” conditions that the system must achieve are:

- INVARIANT: At all times, \( A_1 + A_2 \leq 3 \).
- FINAL STATE: After applying the operation rules, the average processing time should eventually be at most 1s, i.e., \( T_1 + T_2 \leq 1.0 s \).
**Policy verification** To verify the correctness of this policy set, we need to construct the Kripke structure and to derive the LTL formulas associated with the structural and performance system models, and with the set of autonomic computing policies. This process is described below.

(1) Constructing a Kripke structure $M = (S, S_0, R, L)$ for the autonomic data centre. Each combination of values that can be taken by the monitored and controlled parameters of the system corresponds to a different state $s \in S$. For our data centre, the only monitored parameter is $x$, and the controlled parameters are $A_1$ and $A_2$, so the set of states is

$$S \equiv \{ (x, A_1, A_2) \mid 0 \leq x \leq 300, A_1 \geq 0, A_2 \geq 0 \}.$$ 

Since initially $A_1 = A_2 = 0$, the set of initial states $S_0 \subseteq S$ is given by

$$S_0 \equiv \{ (x,0,0) \mid 0 \leq x \leq 300 \}.$$ 

In order to define the set of atomic propositions $AP$ for the Kripke structure, we first take into account the fact that each state $s \in S$ represents a certain configuration for the system. Therefore, we start by including in $AP$ atomic propositions representing the values of the configuration parameters for the system. To do so, we use the notation $[X = a]$ as an atomic proposition stating that the value of the parameter $X$ is $a$. In addition, we need to determine the truth values for the conditions representing the invariants and final states defined by the goal policies for the system. In our running example, the invariant $(A_1 + A_2 \leq 3)$ and the condition for final state $(T_1 + T_2 \leq 1.0s)$ have the generic form $X + Y \leq a$, where $X$ and $Y$ are variables and $a$ is a constant. Therefore, we also include within $AP$ atomic propositions of the form $[X + Y \leq a]$. As a result, the complete set of atomic propositions $AP$ used to define the labelling function $L : S \rightarrow 2^{AP}$ consists of all atomic propositions

$$AP = \{ [X = a] \mid X \in \{ x, A_1, A_2, T_1, T_2, N_1, N_2 \}, a \geq 0 \} \cup \{ [X + Y \leq a] \mid X, Y \in \{ x, A_1, A_2, T_1, T_2, N_1, N_2 \}, a \geq 0 \}.$$ 

For example, given the state $s = (150, 1, 1)$ in which the system is receiving user requests at 150 transactions/second and one spare server is deployed to each tier, several of the atomic propositions that hold in state $s$ are $[x = 150]$, $[A_1 = 1]$, $[A_2 = 1]$ and $[A_1 + A_2 \leq 2]$. Therefore, $L(s)$ contains these propositions, as well as all other propositions in $AP$ that are true in state $s$. Finally, the possible state transitions $R$ are derived from the action policies. The ADD Rule enables all transitions between encoded by the pairs of states in the sets.
and the REMOVE Rule enables the transitions between associated with the state pairs in

\[ R_1 \equiv \{ (s_j, s_k) \mid T_1(s_j) + T_2(s_j) \geq 0.6, T_1(s_j) \geq T_2(s_j), x(s_j) = x(s_k), \]
\[ A_1(s_j) = A_1(s_j) + 1, A_2(s_j) = A_2(s_j) \} \]

\[ R_2 \equiv \{ (s_j, s_k) \mid T_1(s_j) + T_2(s_j) \geq 0.6, T_1(s_j) < T_2(s_j), x(s_j) = x(s_k), \]
\[ A_1(s_k) = A_1(s_j), A_2(s_k) = A_2(s_j) + 1 \}

Additionally, the variation in the request rate \( x \) is encoded by transition relation

\[ R_3 \equiv \{ (s_j, s_k) \mid T_1(s_j) + T_2(s_j) \leq 0.25, A_1(s_j) > 0, x(s_j) = x(s_k), \]
\[ A_1(s_j) = A_1(s_j) - 1, A_2(s_j) = A_2(s_j) \} \]

\[ R_4 \equiv \{ (s_j, s_k) \mid T_1(s_j) + T_2(s_j) \leq 0.25, A_2(s_j) > 0, x(s_j) = x(s_k), \]
\[ A_1(s_k) = A_1(s_j), A_2(s_k) = A_2(s_j) - 1 \}

The complete state transition relation for the Kripke structure is given by the union

\[ R \equiv R_1 \cup R_2 \cup R_3 \cup R_4 \cup R_5. \]

(2) Deriving the LTL properties to be verified. The translation of the two goal policies into LTL formulas is straightforward. Thus, the INVARIANT “always \( A_1 + A_2 \leq 3 \)” maps to the formula

\[ [\square [A_1 + A_2 \leq 3], \]

and the FINAL STATE goal that “eventually \( T_1 + T_2 \leq 1.0 s \)” can be represented by the formula

\[ \diamond [T_1 + T_2 \leq 1.0]. \]

The LTL properties derived so far were verified using the SPIN model checker (Holzmann, 2003). Version 4.2.6 of SPIN was run on a Red Hat Enterprise Linux, 3GHz Pentium-4 PC with 1 GByte of memory. To improve the efficiency of SPIN, we took advantage of its implementation of the state compression algorithm and of the partial-order reduction algorithm. The verification of the LTL properties was carried out for two system models characterised by different server performance parameters:

- Model A: \( \alpha = 0.012, \beta = 0.001, \gamma = 0.15; \)
- Model B: \( \alpha = 0.01, \beta = 0.002, \gamma = 0.02. \)

The purpose of choosing these two models was to demonstrate that the validity of autonomic computing policies depends on the scenario – or system model – to which they are applied.
When the verification of the LTL properties was performed for model A, SPIN found no invalid states, thus confirming that the policy set is conflict free in this scenario. More precisely, the “ADD Rule” and “REMOVE Rule” action policies are guaranteed to lead the system to suitable final states without causing any constraint violations: the autonomic data centre can cope with any request rate in the range 0 to 300 transactions/second.

In contrast, when the policy set was checked for system model B, SPIN detected a possible constraint violation. The counterexample trace generated by SPIN is depicted in Figure 3. This counterexample shows that the system can reach state \( s_6 = (295,2,2) \) that violates the constraint \( A_1 + A_2 \leq 3 \) by starting from the initial state \( s_0 = (199,0,0) \) and transitioning through the intermediate states in the sequence \( \langle s_1, s_2, s_3, s_4, s_5, s_6 \rangle \). This demonstrates that managing the allocation of spare servers to primary-system tiers by means of the “ADD Rule” and “REMOVE Rule” action policies can result in a violation of the IN Variant goal policy in the scenario encoded by system model B.

![Counterexample trace for model B](image)

Figure 3. Counterexample trace for model B

An important lesson to learn from this case study is that a set of autonomic computing policies that is valid for one scenario can exhibit conflicts when the autonomic system operates in a different scenario. Model checking provides the unique capability to verify autonomic computing policies exhaustively and for the precise scenario in which the self-managing system operates.

Finally, note that a simple calculation applied to the invalid state \( s_6 \) from the counterexample in Figure 3 shows that if one spare server were removed from the second tier and added to the first tier, both goal policies would be achieved for system model B. However, the “REMOVE Rule” action policy precludes the removal of spare servers from any tiers when \( T_1 + T_2 \geq 0.25 \). This observation spawned by the analysis of the counterexample generated by the model checker represents a first step towards identifying the root cause of the policy conflict and resolving it.
Table 2: Statistics for the case study

<table>
<thead>
<tr>
<th>Model</th>
<th>States</th>
<th>Time(sec)</th>
<th>Memory(Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model A</td>
<td>939463</td>
<td>6.98</td>
<td>204.5</td>
</tr>
<tr>
<td>Model B</td>
<td>40317</td>
<td>1.46</td>
<td>164.2</td>
</tr>
</tbody>
</table>

Figure 4. Max number of transactions and number of states

Performance evaluation Performance statistics for the case study are shown in Table 2, which lists the number of verified states, the verification time, and the memory used for the analysis of each model. The results in this table show that the verification time is reasonably small for both models. Note that for Model B (for which a policy conflict was identified), the number of states and verification time were much smaller than for Model A. This is because the state-space exploration stopped for Model B when the conflict was identified.

To examine the effect the domain size of a system parameter has on performance, we performed the validation of the policy set for Model A when increasing the maximum number of transactions the system should accommodate from 100 to 500 transactions/second, in steps of 100 transactions/second. To ensure that the state space is searched exhaustively, we also changed the property to be checked so as not to cause any policy violations. Figure 4 shows the relation between the maximum number of transactions and the size of the state space verified in this experiment, and Figure 5 shows the dependence of the computational time and memory usage on the number of states. As expected, the number of states increases dramatically as the maximum number of transactions increases. At the same time, the computational time and memory usage also increase as the number of states increases.

These experimental results indicate that in order to be feasible, the proposed approach for conflict detection in autonomic computing policies has to be applied to small average-sized models. Obtaining such models requires that the number of parameters included in the model and their value ranges are carefully chosen. Model abstraction algorithms such as (Wang, Hachtel, & Somensi, 2006) can alternatively (or additionally) be employed to reduce the size of the state space further.

Related work

There has been a lot of research on policy validity checking of policy-based man-
agement systems. For example, the policy inconsistency detection method for firewalls is proposed by Al-Shaer and Hamed (Al-Shaer & Hamed, 2003), but this type of analysis is concerned with static policy, so policy-based systems that change status dynamically by defined policy are beyond its scope. Another method uses event calculus (EC) (Kowalski & Sergot, 1986) framework for policy verification in (Bandara, Lupu, & Russo, 2003) (Charalambides et al., 2005). In this method, the situation in which an inconsistency occurs is identified by abductive inference on EC, which adds the notion of time to the framework of the first order predicate logic. However, in this method, all of the inference rules needed to identify inconsistencies must be described previously, but it is very difficult to describe all possible inconsistencies before they are detected. Other than the above papers, while there has been research on policy conflict detection and resolution (Dunlop, Indulska, & Raymond, 2002), there has been little research on policy verification and validation that considers the relation between the policy and the system to which the policy is applied, which we do in our research.

![Figure 5. Number of states and resource consumption](image)

**Implementation of goal policies**

As explained in the previous sections, goal policies represent high-level descriptions of the constraints, invariants and success states that an autonomic system is required to comply with or achieve. Often specified using Boolean logic or first-order logic expressions, goal policies do not prescribe how the system objectives they define should or can be achieved. Instead, the configuration-change procedure required to implement goal policies (and thus to achieve the system objectives) has to be synthesised by the autonomic system.

This part of the chapter shows how a combination of related formal techniques, namely formal specification and model synthesis, can be employed to implement goal autonomic computing policies. Note that the implementation of such goal policies adds self-management capabilities such as self-configuration and self-optimisation to IT systems.

The section starts with a brief description of the two formal techniques mentioned above. This is followed by a detailed presentation of our formal method for the implementation of goal policies. The section concludes with a summary of related work.
**Background**

*Formal specification* is a technique for the development of IT systems that consists of expressing the requirements of a system using mathematical notation drawn from set theory and first-order logic. By starting from a set of requirements defined in a formal specification language like Z (Woodcock & Davies, 1996) or B (Abrial, 1996), developers of high-integrity IT systems can reason formally about their designs, develop code that matches these requirements closely, and generate effective assertions and unit tests for their code. Furthermore, tool-supported formal specification frameworks such as Alloy (Jackson, 2006) and B-Method (Schneider, 2001) enable developers to start from an abstract specification of a system and derive a compliant model or a concrete realisation of it in a number of (semi)automatic, *refinement* steps.

Our method for the implementation of goal autonomic computing policies is based on the refinement process described above. The method uses as input an Alloy-encoded formal specification derived from a set of goal policies, and employs the Alloy Analyzer tool (Alloy, 2010) to synthesise a model of the system that satisfies all these constraints. The synthesised model corresponds to a system configuration that fulfils the Alloy specification, and which can be extracted and used to implement the original goal policies.

Before describing the method in more detail, we provide a brief overview of Alloy specifications and the Alloy Analyzer tool. An Alloy specification (Jackson, 2006) is a formal description of the constraints and behaviour of a system that is expressed in a declarative language based on first-order logic. In addition to the standard first-order logic operators (e.g., ∧, ∨, ¬ and →), the Alloy language supports the constructs and operations in Table 3.

Table 3. Alloy constructs and operations; the examples are adapted from (Jackson, 2006)

<table>
<thead>
<tr>
<th>Type</th>
<th>Keyword</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>sig</td>
<td>Formal definition of system components and their relations</td>
<td>sig Man{wife: lone Woman} (Entity Man has a field wife of type Woman and multiplicity lone - each Man is associated with no more than one Woman as wife.)</td>
</tr>
<tr>
<td>Fact</td>
<td>fact</td>
<td>Constraint that always holds</td>
<td>fact{some m:Man</td>
</tr>
<tr>
<td>Predicate</td>
<td>pred</td>
<td>Parameterised constraint</td>
<td>pred spouse(p:Man,q:Woman){q in p.wife} (If q is p’s wife, predicate spouse(p,q) is true)</td>
</tr>
<tr>
<td>Function</td>
<td>Fun</td>
<td>Expression that returns a value</td>
<td>fun husband(q:Woman): set Man {q.(~wife)} (Function that returns a Man entity whose wife field is q)</td>
</tr>
<tr>
<td>Assertion</td>
<td>assert</td>
<td>Assumption to be checked</td>
<td>assert A1{no q:Woman</td>
</tr>
<tr>
<td>Model finding operation</td>
<td>run</td>
<td>Find a model that satisfies all fact declarations</td>
<td>run husband</td>
</tr>
<tr>
<td>Assertion evaluation operation</td>
<td>check</td>
<td>Establish whether an assertion holds or not under the given facts</td>
<td>check A1</td>
</tr>
</tbody>
</table>
Alloy Analyzer (Alloy, 2010) is a free, open-source tool that supports two analysis operations (Table 3): model finding (or synthesis) and assertion evaluation. Given a formal specification expressed in the Alloy declarative language, model finding consists of synthesising a model that satisfies all fact declarations. Assertion evaluation involves establishing whether an assertion holds at all times for a given Alloy specification. Note that the two operations are equivalent because identifying a scenario in which an assertion does not hold is equivalent to finding a model that satisfies all fact declarations and the negation of the considered assertion.

Several characteristics of the Alloy specification language and of the Alloy Analyzer tool make the Alloy platform particularly suitable for the implementation of goal autonomic computing policies using the method detailed later in this section:

- Expressiveness - The specification language supports the definition of the complex structural constraints and relationships encountered in autonomic IT systems.
- Ability to represent state transitions - The pred Alloy construct can be used to specify the operations associated with changes in the autonomic system configuration.
- Effectiveness - The search for a model in model finding operations explores the entire state space of the system. This exploration is conducted very efficiently, as it is based on translating Alloy specifications into a 3-SAT conjunctive normal formula (CNF) and using a powerful SAT solver such as SAT4J (Lens, 2010) or miniSat (Niklas & Niklas, 2003) to determine its satisfiability, and thus the existence of a model that satisfies the original specification.

Synthesis of configuration-change procedures from goal autonomic computing policies

Description of the approach

Our method for synthesizing a configuration procedure for an autonomic IT system starting from its goal policies is depicted in Figure 6. This method comprises four steps:

1. Structural and performance system models similar to those described in Section 2.2 are derived in this step. The system characteristics defined by the two system models are:

- The types and numbers of system components;
- The relationships between these components (e.g., “Server A is connected to Switch B” or “Application software C is available for operating system D”);
- The types and values of the component parameters that are relevant for the intended autonomic functionality (e.g., “Server A has a parameter called memorySize, and its value is 4GByte” or “Web server E has a parameter termed maxConnections, whose value is 300”).

We assume this information can be derived from monitoring the running system, e.g., by means of discovery tools such as Tivoli Application Dependency Discovery Manager (IBM-Tivoli, 2010). The structural and performance system models need to be encoded in a well-defined, which for our realisation of the method is a customized XML format used to integrate system and management information called Resource Control eXtensible Markup Language (RCXML) and introduced in (Katsuno et al., 2007). Note that RCXML is capable of representing the various
types of relations between the components of a system. Therefore, we use it to specify a system consisting of various types of components, and then translate it into the states of a model that encodes the configuration of the system. Details about RCXML are provided when we present the case study used to illustrate the application of the method later in this section. However, note that any equivalent format can be used for the representation of this information, as long as the model transformation technique employed in the second step of the method is adjusted to cope with that equivalent format.

Figure 6. Synthesis of configuration-change procedures from goal policies

(2) The system information obtained in step one is translated into Alloy *sig* declarations that define the system components and their parameters, and *fact* declarations that specify the “facts” that hold in the initial system state (i.e., prior to executing any configuration change procedures synthesized by our method). The result is a partial Alloy specification that is augmented to a full specification in the next step of the method. In our realisation of the method, we use a Java translator that automates the generation of the Alloy *sig* and *fact* declarations from the RCXML representation of the structural and performance system models.

(3) In this step, expert system management knowledge is acquired from domain experts in different aspects of the autonomic system, and is encoded in Alloy. For a typical IT system, the domain experts from Figure 6 may include network experts, database experts, operating systems experts and/or application experts. The system management knowledge that each of these experts contributes with consists of constraints, descriptions of system configuration operations with pre- and post-
conditions, and goal autonomic computing policies. All of these are encoded in the Alloy declarative language, either by these experts if they have experience with Alloy or, more likely, by the developer of the autonomic capabilities for the system. Thus, constraints are expressed as fact Alloy declarations encoding the relationships that must be maintained in all states. Next, the operations and their pre-/post-conditions are expressed as a combination of pred and fact Alloy constructs, or in some cases, merely as fact declarations. All of these Alloy definitions complete the partial Alloy specification from step two, producing the Alloy specification required to synthesize the configuration-change procedure.

(4) In the final step of the method, the Alloy Analyzer tool is employed to identify a model that satisfies the specification built in the previous steps of the method. This model is produced as a sequence of value assignments to variables from the operation definitions in the specifications. As a result, the model maps directly to a sequence of state transitions satisfying all given fact declarations, or to a synthesized configuration procedure that takes the system from the initial/current state to a state satisfying its goal policies without violating any constraints.

Case study

The case study used to illustrate the application of the method involves synthesising the procedure for system configuration changes in an IT system that employs Xen virtual machines (Barham et al., 2003) to run a three-tier software application. The system comprises three physical servers (Server_A, Server_B and Server_C), an OS-image storage device (OS_image) and a switch (Switch_S1) that can be used to organise any subset of the other components into a VLAN. The software application consists of an Apache web server, an Interstage application server (Fujitsu, 2010) and a MySQL database server that are each running within their own Linux virtual machines (VMs).

Figure 7 depicts the initial state of the system, in which Server_A, Server_B and OS_image are connected to Switch_S1, and belong to the same VLAN segment V1, while Server_C is not connected to the network. The three components of the software application are deployed as follows: Apache on the VM Guest_OS_A1 running on Server_A; Interstage on the VM Guest_OS_A2 which is also running on Server_A; and MySQL on Guest_OS_B1 running on Server_B.
To avoid overcommitting physical server resources, we quantify the capacity of the physical servers and the amount of resources necessary to run each VM. Thus, the CPU, memory and hard-disk (HDD) resources of each physical server are assigned a "size" parameter that can take values in the range 0 to 32. A size of one unit for any of these resource types corresponds to the capacity of that resource type for the entry-level VM offered by the Amazon EC2 service (Amazon-EC2, 2010), namely a 1.0-1.2 GHz Intel Xeon/AMD Opteron VM with 1.7 GB memory and 160 GB of local instance storage. Similarly, each guest OS (i.e., virtual machine) is assigned a "size" parameter, and the system is considered to be in a valid state if no physical server is running a set of VMs whose combined size exceeds the size of any server resource. For instance, the overall size of the two VMs running on Server_A from the system in Figure 7 is $8 + 2 = 10$, which is less than the sizes of the CPU (i.e., 12), memory (i.e., 12) and storage (i.e., 16) available on this server.

The remainder of this section describes how the four steps of the method for the synthesis of a configuration-change procedure are carried out for the autonomic IT system from our case study. The high-level objective of this autonomic system is to consolidate the virtual machines running the three-tier software application on as few servers as possible, so the synthesised configuration-procedure will involve migrating VMs across physical servers, and re-organising the VLAN and the access to the OS_image storage device.

(1) In the first step of the method, RCXML-encoded structural and performance system models are derived from the information provided by the annotations in Figure 7. A fragment of the RCXML representation of the structural and performance model is shown in Figure 8. This fragment defines the physical configuration of Server_A, with its three physical components (CPU, memory and hard disk) and their relationships with the Server_A element (represented by means of "componentOf" links), and their performance as the "size" parameters. A fragment of the structural model is also shown in Figure 9, where "Link" XML elements with attributes such as "connectedTo" or "runningOn" represent relationships between other system components. The fact that some components belong to the VLAN segment V1 is specified by the "VLANs" connections between these components and the element representing VLAN V1.

(2) In the second step of the method, the RCXML system models are translated into a partial Alloy specification that consists of sig and fact declarations associated with the elements of these models. A fragment of the Alloy declarations for the system from our case study are shown in Figure 10. Notice how the sig State {...} declaration specifies the components of servers (i.e., CPU, memory and hard disk) as well as the other relationships between the elements of the IT system. Additionally, a fact declaration is generated that defines the initial configuration of the system; the use of the prefix first indicates that all parameter values correspond to the initial state of the system. Note that in our realisation of the method, the generation of this partial Alloy specification is fully automated by means of a Java translator that takes as input the RCXML-encoded models and uses a series of XPath queries to produce the required Alloy declarations. This technique and the Java tool are described in more detail in (Kikuchi & Tsuchiya, 2010).

(3) We will split the description of this step of the method into three parts that correspond to the specification and translation into Alloy of three elements of the autonomic system: operations, constraints (or invariants) and goal policies.
<Component>
  <Hardwares>
    <Servers>
      <Server id="Server_A">
        <Configuration>
          <CPU id="CPU_A" size="12" />
          <Memory id="Memory_A" size="12" />
          <HardDisk id="HDD_A" size="16" />
          <Link id="svrA_link1" src="CPU_A" dest="Server_A" type="componentOf" />
          <Link id="svrA_link2" src="Memory_A" dest="Server_A" type="componentOf" />
          <Link id="svrA_link3" src="HDD_A" dest="Server_A" type="componentOf" />
        </Configuration>
      </Server>
      ...  
    </Servers>
    ...  
  </Hardwares>
  ...  
</Component>

Figure 8. System Configuration definition in RCXML

<Links>
  <Link id="ctl1" src="Server_A" dest="Switch_S1" type="connectedTo" />
  <Link id="ctl2" src="Server_B" dest="Switch_S1" type="connectedTo" />
  <Link id="ctl3" src="OS_image" dest="Switch_S1" type="connectedTo" />
  ...  
  <Link id="osl1" src="Host_OS_A" dest="VMM_A" type="runningOn" />
  <Link id="osl2" src="Guest_OS_A1" dest="VMM_A" type="runningOn" />
  ...  
  <Link id="acl1" src="VMM_A" dest="OS_image" type="accessTo" />
  <Link id="acl2" src="VMM_B" dest="OS_image" type="accessTo" />
  ...  
  <Link id="btl1" src="Server_A" dest="V1" type="VLANs" />
  <Link id="btl2" src="Server_B" dest="V1" type="VLANs" />
  <Link id="btl3" src="Switch_S1" dest="V1" type="VLANs" />
  <Link id="btl4" src="OS_image" dest="V1" type="VLANs" />
  ...  
</Links>

Figure 9. Relation definition between components written in RCXML

Operations  We consider that the following operations are available:

- Connection - we can establish a physical network connection between any server and network device (e.g. a switch) with an Ethernet cable.
- Access configuration - We can modify some configuration files of servers in order to allow one server or piece of software to access another one.
- VLAN configuration - We can change VLAN configurations of servers or network devices in order to make them belong to some VLAN segments.
- Migration - We can move a virtual OS from one VMM to another by using the migration function of virtual machines under the condition where both physical servers accommodating these VMM can access the same OS_image storage.
/*** System state definition ***/

sig State {
    size : (CPU->Int)+(Memory->Int)+(HardDisk->Int)+(OS->Int),
    connectedTo: (Server -> NetworkDevice) + (NetworkDevice -> Server)
        +(NetworkDevice ->NetworkDevice),
    componentOf: (CPU + Memory + HardDisk) -> Server,
    accessTo : (Program + OS + Server) -> (Program + OS + Server),
    runningOn : (Program + OS ) -> (Program + OS + Server),
    ...
}

/*** Initial state configuration ***/

fact {
    first.size = (CPU_A -> Int[12]) + (CPU_B -> Int[16]) + (CPU_C -> Int[32])
        + (Memory_A -> Int[12]) + (Memory_B -> Int[24]) + ... 
    first.componentOf = (CPU_A -> Server_A) + (Memory_A -> Server_A) + ...
    first.accessTo = (VMM_A -> OS_image) + (VMM_B -> OS_image)
    first.runningOn = (Host_OS_A -> VMM_A) + (Guest_OS_A1 -> VMM_A) + ...
    ...
}

Figure 10. System configurations translated from RCXML to Alloy

These operations are fully defined by their pre- and post-conditions, as illustrated by the Alloy specification in Figure 11. In these Alloy definitions of the four operations, s and s' represent the system state before and after the execution of the operation, respectively. For example, the Alloy predicate for connect specifies several conditions that hold when a component src is connected to a component dst:

- no connectedTo relation exists from src to dst before executing the operation;
- no connectedTo relation exists from dst to src before executing the operation;
- the connectedTo set of relations after the connect operation is the union of connectedTo before the operation and the set containing the newly established connection, i.e., \{ (src -> dst) \}.

The rest of the operations are defined in the same manner. For instance, the pre-conditions for the migrate operation require that src and dst are Xen VMs, and that OS_image is accessible to both.

Note that we need to define explicitly not only all changes effected by each operation, but also the fact that any unchanged parameters or relationship preserve their values. This so-called frame conditions (Reiter, 1991) are required to preclude the synthesis of impossible configuration changes in the last step of the method. In our Alloy specification, the frame conditions are encoded by the frame_condition predicate in Figure 12. This predicate ensures that only parameters whose flags are set in an auxiliary variable s'.changes can be modified by any operation. To take advantage of this construct, each definition of an operation marks as changed all parameters that it rightfully modifies. For instance, a connect operation records the fact that it only changes the connectedTo parameter by including the expression s'.changes = connectedTo_c at the end of its code.
/*** Operation Definitions ***/

/* Connection operation: Connect src and dst */
pred connect[s,s':State, src,dst: Objects] {
  (not ((src->dst) in s.connectedTo)) &&
  s'.connectedTo = s.connectedTo + (src->dst) &&
  s'.changes = connectedTo_c }

/* Access config operation: make src access to dst */
pred addaccessTo[s,s':State, src,dst: Objects] {
  not ((src->dst) in s.accessTo)) &&
  s'.accessTo = s.accessTo + (src -> dst) &&
  s'.changes = accessTo_c }

/* VLAN config operation: make src join dst */
pred joinVlan[s,s':State, src,dst: Objects] {
  s'.VLANs = s.VLANs + (src -> dst) &&
  s'.changes = VLANs_c }

/* Migration operation: migrate vm from src to dst */
pred migrate[s, s':State, vm,src,dst: Objects] {
  (Xen in (src.(s.name) & dst.(s.name)) ) &&
  (OS_image in (src.(s.accessTo) & dst.(s.accessTo) ) ) &&
  ((vm->src) in s.runningOn) &&
  s'.runningOn = s.runningOn ++ (vm->dst) &&
  s'.changes = runningOn_c }

Figure 11. Operation definitions in Alloy

/*** Frame conditions ***/
pred frame_condition[s,s':State] {
  (s.connectedTo = s'.connectedTo || connectedTo_c in s'.changes) &&
  (s.componentOf = s'.componentOf || componentOf_c in s'.changes) &&
  (s.accessTo = s'.accessTo || accessTo_c in s'.changes) &&
  ...
}

Figure 12. Frame conditions

/*** State Transition Definitions ***/
fact StateTransition { all s: State, s': ord/next[s] |
  (some disj x,y: (s.Server + s.NetworkDevice) | connect[s,s',x,y]) ||
  (some disj x,y: (s.Program + s.OS + s.Server) | addaccessTo[s,s',x,y]) ||
  (some x: (s.Server + s.NetworkDevice) | some y: s.VLAN | joinVlan[s,s',x,y]) ||
  (some x: s.OS | some disj y,z: s.Program | migrate[s,s',x,y,z]) } &&
  frame_condition [s,s']

Figure 13. Definitions of state transitions with operation knowledge and frame conditions

The system state transitions (i.e., its possible configuration changes) associated with these operations can then be defined by integrating the operation knowledge encoded by the Alloy predicates in Figure 11 and the frame conditions in Figure 12. The resulting Alloy fact – depicted in Figure 13 – holds for components x, y, and z and for states s and s' if and only if the system transition from state s to state s' is valid.
Constraints
We further assume that three constraints were defined for the system: the network management expert defined an accessibility constraint and a VLAN constraint; and the VM management expert defined a capacity constraint. The three constraints, which must be complied with at all times, are presented below:

- Accessibility constraint - two network components can access each other only if they are connected via network links;
- VLAN constraint - two network components can access each other only if they belong to the same VLAN segment, or neither of them belongs to any VLAN segment;
- Capacity constraint - the total size of the guest operation systems (i.e., VMs) on a physical server must not exceed the size of any server resource.

The Alloy fact declarations equivalent to these constraints are shown in Figure 14. For example, the accessibility constraint is defined by the fact declaration mentioning that if there is an accessTo relation between some components $x$ and $y$ in a state $s$, then $y$ should be reachable from $x$ through a set of runningOn and connectedTo relations and the inverse relations of them in the state. In the Alloy code, a reflexive transitive closure of a relation and its inverse are represented using the '*' operator and '~' operator, respectively. The concatenation of different types of relation is specified using the '.' operator.

Goal policies
The goal policies for the system represent the last piece of information supplied by the system experts/administrators. In our case study, there is only one goal policy. This policy requires the consolidation of the three software components of the application running on the system (i.e., Apache, Interstage and MySQL) on the same physical server, in order to minimise operation costs. This goal policy can be encoded as the Alloy fact construct in Figure 15, where the '^' operator represents the closure of relations without any reflexive relation. This fact declaration states the existence of a state $s$ in which Apache, Interstage and MySQL are all running on the same server $x$.

(4) In the final step of the method, the Alloy specification derived in the previous steps was supplied to the Alloy Analyzer tool. The experiment was carried out on a 64-bit Red Hat Enterprise Linux PC with an Intel Xeon 3GHz CPU and 2 GB memory, and Alloy Analyzer was configured to use miniSat (Niklas & Niklas, 2003) as its SAT solver engine.

In its search for a model satisfying the specification, Alloy Analyzer incrementally searches the state space within the range from 0 operations to a user-specified upper limit for the number of operations. In this manner, Alloy Analyzer finds a model representing a configuration procedure that can achieve the goal policies with the smallest number of configuration steps between 0 and the upper limit, assuming that such a model exists. If no model exists that satisfies the specification and has no more operations than the upper limit for the search, then the experiment is inconclusive: a larger model may exist, or the operations, conditions and goals specified by the domain experts may be conflicting.
/*** Declarative constraints (invariants) /***/

/* Constraint 1: If x accesses to y, x and y should be connected */
fact {all s: State | all disj x,y: (s.Program + s.OS + s.Server) |
  (x->y) in s.accessTo => y in x.*(s.runningOn).
  *(s.connectedTo + ~(s.connectedTo)). *(~(s.runningOn))}

/* Constraint 2: If x accesses to y, x's and y's server should belong to the same VLAN */
fact {all s: State | all disj x,y: (s.Program + s.OS + s.Server) |
  (x->y) in s.accessTo => (no (x+y).*(s.runningOn). (s.VLANs)) ||
  y in x.*(s.runningOn).(s.VLANs).~(s.VLANs).*(~(s.runningOn))}

/* Constraint 3: Total OSs size should be under than components' capacity */
fact {all s: State | all x: s.Server | all c: x.(~(s.componentOf)) |
  c.(s.size) >= (sum y: x.^(~(s.runningOn)) | (y & s.OS).(s.size))}

Figure 14. Declarative constraints

/*** Goal policy /***/

fact { some s: State | one x: s.Server |
  x in Apache.^(s.runningOn) &&
  x in Interstage.^(s.runningOn) &&
  x in MySQL.^(s.runningOn) }

Figure 15. Goal policy

For our case study, Alloy Analyzer took 200 seconds to synthesise a model that corresponds to the configuration-change procedure shown in Figure 16 and described below:

1. Connect Server_C with Switch_S1.
2. Incorporate Server_C into VLAN V1.
3. Establish access from Server_C to OS_image.
4. Move Guest_OS_B1 to VMM_C.
5. Move Guest_OS_A2 to VMM_C.
6. Move Guest_OS_A1 to VMM_C.

This synthesised procedure reveals several facts. First, due to the capacity constraint, the three pieces of software need to be consolidated on Server_C. However, in order to move VMs to Server_C, it should be able to access OS_image. In addition, accessing OS_image requires that Server_C and OS_image belong to the same VLAN segment and that a physical connection is established between them. Therefore, steps 1 to 3 are required to prepare the system for the VM migrations to Server_C.

Performance evaluation To evaluate the performance of our method for the implementation of goal policies, we changed the upper bound for the number of configuration steps and modified the initial configuration in the case study as described below. Then, we executed the synthesis of the procedure and evaluated the computational time and the number of clauses in the 3-SAT formula used internally by the Alloy Analyzer tool.
• **Experiment 1**: We increased the number of configuration steps to be searched from 0 to 11, for a scenario in which the system had the same architecture as in the original case study and started in the new initial state.

• **Experiment 2**: In addition to the changes from Experiment 1, we also increased the number of spare servers with the same configuration as Server\_C from 1 to 5. As each spare server is represented by six components in the Alloy specification (i.e., the server itself, hard disk, memory, CPU, virtual machine, and its host OS), the number of such components for the experiment ranged from 26 to 50 in steps of six. The number of configuration steps in a sequence was fixed at six.

![Figure 16. Procedure synthesis result](image)

Figures 17 and 18 plot the results for the two experiments. The results show that when the number of configuration steps to be synthesised or the number of components in an experiment increases, the computational time and the number of 3-SAT clauses also increase. The exception is the fluctuating computational time in Figure 17, which we found to be associated with early-terminated state searches due to the identification of instances satisfying the given conditions well before an exhaustive state search was conducted.

![Figure 17. Number of steps and 3-SAT clauses and computational time for Experiment 1](image)
Note that the computational times to synthesise the configuration-change procedure comprises two parts: (1) the time for constructing the 3-SAT clauses from Alloy descriptions by the Alloy Analyzer; and (2) the time for the SAT solver (i.e., miniSat in our case) to find a model satisfying these 3-SAT formulas. Figure 19 shows the dependence between these computational times and the number of 3-SAT clauses in the experiments. These results show that it takes a larger amount of time for the SAT solver to find a model (represented by the solid lines) compared to the time required for the construction of the 3-SAT clauses (represented by the dotted lines) when there are many clauses. However, the times for constructing formulas and the number of 3-SAT clauses have almost linear relationships. This led us to the conclusion that 3-SAT clauses are derived in a naive way within the Alloy Analyzer. It is, however, a known fact that the number of 3-SAT clauses can be drastically reduced by customizing the CNF construction algorithm as described in (Nakamura, Naruse, Takagi, & Takagi, 2007). This improvement can be applied to reduce the computational times taken by the implementation of goal autonomic computing policies using the method presented in this section.
Related work

The configuration of complex IT systems is widely regarded as one of the key challenges in system management (Brown, Keller, & Hellerstein, 2005). As such, multiple research projects have investigated system configuration procedure planning, including Plaint (Arshad, Heimbigner, & Wolf, 2003) (Arshad, 2004) and LPG (Gerevini & Serina, 2002). Most of these techniques rely on procedural knowledge containing information about the pre- and post-conditions of each operation. They synthesize a procedure just by connecting operations in accordance with the procedural knowledge. However, various experts managing their systems have different areas of expertise about not only the procedure for configuration changes, but also the declarative constraints that should be kept in their system. Therefore, we believe that using only procedural knowledge to design configuration procedure for today’s IT systems is insufficient, and that knowledge describing discrete conditions independent from the procedural knowledge is strongly required. In order to incorporate conditions independent from procedural knowledge into a configuration planning method, several approaches such as SPICE (Eilam, Kalantar, Konstantinou, & Pacifici, 2004) (El Maghraoui, Meghrajani, Eilam, Kalantar, & Konstantinou, 2006) by IBM have been proposed. They, however, require programming to define these constraints and cannot express these constraints in a declarative way. We suppose that system management knowledge can be added, removed or modified in system management lifecycles because of various reasons such as changes in the system management policy, emissions of new components and disposition of knowledge about obsolete components. Therefore, it is very disadvantageous to embed that knowledge in the procedural program code, because it is difficult to modify the embedded knowledge scattered in a planning algorithm. For these reasons, some researchers have started to realize the importance of using discrete and declarative constraints for system management (Microsoft Research, 2008), although most of their research is in the early stage and concerns the propositions of concept or architecture, in contrast with our approach that is based on a logical and mathematical foundation.

A couple of existing projects apply Alloy to system configuration (Narain, 2005) (Warren, Sun, Krishnamohan, & Weerasinghe, 2006). These approaches aim at finding static correct configurations at a specific point of time or detecting possible constraint violations. They are therefore completely different from our approach, which synthesizes a procedure for dynamically changing a system configuration using discrete knowledge.

Implementation of utility-function policies

A utility function represents a measure of the degree to which a system satisfies its high-level objectives. Utility functions are typically defined as non-negative-valued expressions that depend on system parameters such as throughput, dependability or running costs. Utility-function autonomic computing policies specify a utility function whose value needs to be maximised through continually adjusting the configurable parameters of the system in line with changes in its workload, environment, etc.

This section describes an approach to implementing utility-function policies that is based on a formal technique termed quantitative verification. Through the implementation of these utility-function policies, autonomic IT systems provide self-optimisation functionality. We start by introducing quantitative verification next, then present our approach to implementing utility-function policies. The section concludes with a brief overview of related work.
Quantitative verification is a mathematically-based technique for establishing the correctness, performance and reliability of systems that exhibit stochastic behaviour (Kwiatkowska, 2007). Given a precise mathematical model of a real-world system, and formal specifications of quantitative properties of this system, an exhaustive analysis of these properties is performed. Example properties include the probability that a fault occurs within a specified time period, and the expected power consumption of an IT system under a given workload.

The most common system models used by quantitative verification are Markov models such as continuous- and discrete-time Markov chains (CTMCs and DTMCs), and Markov decision processes (MDPs). The approach for the implementation of utility-function policies described in this section works for autonomic systems whose components can be modelled using any of these types of Markov models. Because the case study used to illustrate the approach involves the quantitative verification of a continuous-time Markov chain (CTMC), we will introduce only this type of Markov model. A detailed description of DTMCs and MDPs in the context of quantitative verification is available from (Kwiatkowska, Norman, & Parker, 2007).

A CTMC is a tuple \( \langle S, s_0, R, L \rangle \), where \( S \) represents a finite set of states, \( s_0 \in S \) is the initial state, \( R : S \times S \rightarrow R_{\geq 0} \) is a transition rate matrix, and \( L : S \rightarrow 2^{AP} \) is a labelling function that associates a subset of the atomic propositions \( AP \) with each CTMC state. Given two states \( s_1, s_2 \in S \), the probability of the transition between \( s_1 \) and \( s_2 \) being enabled within \( t > 0 \) time units is \( 1 - e^{-R(s_1,s_2)t} \). The use of exponentially distributed delays between state transitions makes CTMCs effective at modelling real-world system characteristics such as component failures and repairs, and request inter-arrival and service rates.

The quantitative properties to be verified are expressed in probabilistic temporal logics such as Continuous Stochastic Logic (CSL) (Aziz et al., 2000; Baier, Haverkort, Hermanns, & Katoen, 2003) for CTMC models (Table 4). For example, given a state \( s \in S \) and an atomic proposition \( a \in AP \), \( s \models P_{\leq t}[true \cup [0,t]a] \) specifies that the probability of satisfying the atomic proposition \( a \) after at most \( t > 0 \) time units after starting in state \( s \) is greater than or equal to \( p \).

Probabilistic model checkers are software tools that can be used to establish quantitative system properties that can be expressed in CSL or in a similar probabilistic temporal logic. The probabilistic model checker PRISM (Kwiatkowska, Norman, & Parker, 2005b) that our approach uses to implement utility-function policies can handle probabilistic models including CTMCs, DTMCs and MDPs. These models are specified in the high-level PRISM modelling language. They consist of sets of modules that correspond to the components of the system, and whose parallel composition represents the model of the overall system. Each module is defined as a set of guarded commands that describe the possible changes in the values of a set of a variables associated with the module. A generic command from a CTMC model has the form

\[
\text{[label]} \text{ guard -> rate : action.}
\]

This command specifies that the transitions in the value of the module variables given by \text{action} is enabled with the given \text{rate} whenever the Boolean expression \text{guard} is true \text{ and} all transitions with the same \text{label} from other modules are also enabled.
Table 4: CSL operators; the first two operators are used to build path formulas, and the last couple of operators build state formulas. A CSL formula is a state formula.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>$X_\phi$</td>
<td>State formula $\phi$ is true in the next state</td>
</tr>
<tr>
<td>(Time-bounded) Until</td>
<td>$\phi_1 U^i \phi_2$</td>
<td>State formula $\phi_2$ is true at some moment in the time interval $I \subset R_{\geq 0}$ and $\phi_1$ is true at every preceding moment.</td>
</tr>
<tr>
<td>Probabilistic operator</td>
<td>$P_{\text{eq}}(\psi)$</td>
<td>Path formula $\psi$ is true with probability $\triangleright p$, where $\triangleright \in {&lt;, &gt;, \leq, \geq}$ and $p \in [0, 1]$.</td>
</tr>
<tr>
<td>Steady-state operator</td>
<td>$S_{\text{eq}}(\phi)$</td>
<td>In the long run, state formula $\phi$ is true with probability $\triangleright p$, where $\triangleright \in {&lt;, &gt;, \leq, \geq}$ and $p \in [0, 1]$.</td>
</tr>
</tbody>
</table>

Implementation of utility-function policies using quantitative verification

Description of the approach

An autonomic IT system comprises parameters that are monitored by its MAPE loop and parameters whose values are adjusted by the MAPE loop in response to changes in the values of the monitored parameters. We shall call the first set of parameters state parameters, and the set of parameters that are modified by the MAPE loop configuration parameters. The state parameter changes induced by adjustments in the configuration parameter values represent the behaviour of the system, and a model that describes this relationship between the configuration and state parameters of a system is termed a behavioural model or operational model.

Given an autonomic IT system with the state parameters $s_i \in S_i$, $1 \leq i \leq n$ and the configuration parameters $c_i \in C_i$, $1 \leq i \leq m$, our approach can be used to implement utility-function policies based on a utility function of the form

$$
utility : S_1 \times S_2 \times \cdots \times S_n \times C_1 \times C_2 \times \cdots \times C_m \rightarrow R
$$

if a PRISM CTMC, DTMC or MDP operational model for the system is available. PRISM operational models for a wide range of application domains are available from (Kwiatkowska et al., 2005b; Kwiatkowska, Norman, & Parker, 2005a; Prism, 2010), and can be used directly or as templates for building operational models for new types of IT systems.

One class of utility functions that is particularly flexible and straightforward to use with our approach is the class of multi-objective utility functions defined analytically as linear combinations of several system objectives:

$$
utility(s_1, s_2, \ldots, s_n, c_1, c_2, \ldots, c_m) = \sum_{i=1}^{r} w_i \text{objective}_i,
$$

where the weights $w_i \in R$, $1 \leq i \leq R$, are used to specify the trade-offs between the $r \geq 1$ system objectives. Each objective function $\text{objective}_i$, $1 \leq i \leq R$, is an analytical expression of system parameters and formally specified quantitative properties to be analysed using PRISM.
To implement a utility-function policy based on the utility given by eq. (2), our realisation of the MAPE autonomic computing loop employs PRISM to analyse the quantitative properties used to define the system objectives $objective_i$, $1 \leq i \leq R$. This analysis is performed each time when a change in the monitored system parameters triggers the execution of the MAPE loop, and consists of establishing the value of these quantitative properties for a set of possible values for the configurable system parameters $c_i$, $1 \leq i \leq m$. For systems whose configuration parameters can take a finite set of discrete values, all possible configurations can be analysed in this way, and new, optimal values for the configuration parameters are obtained as

$$(c'_1, c'_2, ..., c'_m) = \arg\max_{(x_1, x_2, ..., x_m) \in C_1 \times C_2 \times \cdots \times C_m} utility(s_1, s_2, ..., s_n, x_1, x_2, ..., x_m)$$

This is the scenario encountered in the case study presented later in this section. For systems that comprise configuration parameters that can take a continuous range of values, the same analysis is performed for a finite set of possible configurations $C_1 \times C_2 \times \cdots \times C_m$. While this precludes the calculation of the optimal configuration parameters, an effective suboptimal solution can be obtained for many real-world autonomic systems if the analysed set of configurations is chosen carefully – this scenario is presented and validated in our related work (Calinescu & Kwiatkowska, 2009b).

Several characteristics make the probabilistic model checker PRISM particularly suitable for the implementation of utility-function policies as described above. First, PRISM supports the concept of experiments, i.e., the analysis of quantitative properties for a range of values for the model parameters. Therefore, a single PRISM operation is sufficient to analyse a quantitative property associated with a system objective from (2) for all configurations examined during an execution of the MAPE loop. Furthermore, PRISM provides a command-line interface that the MAPE loop uses to run the quantitative verification automatically, as a background process. Finally, an extensive, independent performance analysis of a broad selection of probabilistic model checkers (Jansen et al., 2008) ranked PRISM as the best tool for the quantitative analysis of large models such as the ones encountered in the work described in this section.

**Case study**

To illustrate the approach to implementing utility-function policies described so far, we will use a case study that involves the adaptive allocation of servers within a data centre. Given a data centre comprising a set of clusters, the objective of this application is to dynamically adjust the allocation of the data-centre servers to clusters in line with changes in (a) the data-centre and cluster parameters; and (b) the user-specified data-centre objectives. The parameters of this system are

- $N \geq 1$, the number of clusters in the data-centre;
- $totalServers \geq 1$, the number of servers in the data-centre;
- $priority_i > 0$, the priority of cluster $i$, $1 \leq i \leq N$;
- $requiredServers_i \geq 1$, the number of servers that cluster $i$ needs in order to handle its workload, $1 \leq i \leq N$;
- $allocatedServers_i \geq 0$, the number of servers allocated to cluster $i$, $1 \leq i \leq N$;
- $targetAvailability_i \in [0,1]$, the user-specified steady-state probability that cluster $i$ has at least $requiredServers_i$ operational servers, $1 \leq i \leq N$. 

Note that the configuration parameters of the system are \( \text{allocatedServers}_i \geq 0 \), \( 1 \leq i \leq N \). All other parameters described above are state parameters. Also known, and assumed to be constant, are the cluster topology and the failure and repair rates of the cluster components (i.e., servers, switches and backbone).

We will use an existing PRISM Markov chain that models the behaviour of a cluster with these characteristics. This is a continuous-time Markov chain taken from (Prism, 2010) and shown in Figure 20. The original purpose of this CTMC was to perform off-line analysis of quantitative properties of the cluster.

The utility function for our case study is a linear combination of \( N + 1 \) system objectives. The first \( N \) of these objectives use a CSL formula to express the requirement
that each of the $N$ clusters achieves its target availability, and the last objective encodes the requirement that as few data-centre servers as possible are allocated to the $N$ clusters at any point in time:

$$\text{utility} = \sum_{i=1}^{N} \text{priority}_i \cdot (S_{\text{targetAvailability}}[\text{operationalServers}_i \geq \text{requiredServers}_i] ? 1: 0) - \varepsilon \sum_{i=1}^{N} \text{allocatedServers}_i$$

where $\text{operationalServers}_i$, $1 \leq i \leq N$, represents the number of operational servers in the $i$-th cluster. Notice that the weights used for the first $N$ system objectives are the priorities of the $N$ clusters. The weight for the last objective is a small positive constant $\varepsilon \ll 1$; this makes the MAPE loop select the system configuration with the fewest allocated servers when several configurations exist that yield identical utility with respect to the first $N$ system objectives.

The optimal configuration for the data centre is determined by the MAPE loop each time when there is a change in one of the state parameters of the system. Each such iteration of the MAPE loop involves performing a series of PRISM experiments to analyse the CSL-encoded property in eq. (3) for the $N$ data-centre clusters. In performing these experiments, the parameters of the CTMC model that correspond to state parameters of the system (i.e., $\text{totalServers}$, $\text{requiredServers}_i$ and $\text{targetAvailability}_i$, $1 \leq i \leq N$) are fixed so as to match the parameter values obtained from monitoring the system and from the user-specified target availabilities. In contrast, the configuration parameters are assigned each possible value, namely $\text{allocatedServers}_i = \text{requiredServers}_i, \text{requiredServers}_i + 1, \ldots, \text{totalServers}$, $1 \leq i \leq N$. Figure 21 depicts the result of the PRISM experiments carried out for a data centre with $N = 3$ clusters and $\text{totalServers} = 25$ when $\text{requiredServers}_1 = 5$, $\text{requiredServers}_2 = 7$, $\text{requiredServers}_3 = 10$. These experiments can be used, for instance, to establish the number of servers that should be allocated to each cluster in order to achieve two-nines or three-nines availability for that cluster.

Carrying out background PRISM experiments such as those illustrated in Figure 21 enables the MAPE loop to identify the optimal system utility and the configuration that yields it. This configuration is then used to ensure that the autonomic data centre achieves maximum utility in its new state.

We implemented an autonomic data-centre simulator, and we used it to simulate a data centre comprising $N = 3$ clusters with different priorities: a “GOLD” cluster, a “SILVER” cluster and a “BRONZE” cluster. Figure 22 shows a typical set of experimental results that were obtained over a four-week period in simulated time. During this time period, the clusters were subjected to variable workloads by varying the values of the $\text{requiredServers}_i$ state parameters, $1 \leq i \leq 3$. Each such change triggered the execution of the utility-function policy (3), and thus the self-adjustment of the configuration parameters $\text{allocatedServers}_i$, $1 \leq i \leq 3$, based on the results of the quantitative analysis performed using PRISM experiments. With all possible data-centre configurations analysed by the PRISM experiments, the selection of the optimal configuration took up to 14 seconds with the experiments carried out on a 3.14GHz Intel Core-2 Duo CPU desktop PC. This response time is acceptable for this case study since under a quarter of a minute represents a small delay compared to the time required to provision a server when it is allocated to a new cluster. In another case study presented in (Calinescu &
Kwiatkowska, 2009b), we demonstrate that sub-second response time can be achieved for a real-world autonomic system.

![Figure 21](image)

Figure 21. PRISM experiments that analyse the expected availability of clusters with 5, 7 and 10 required servers, respectively

As indicated by the experimental results in Figure 22, implementing the utility-function policy ensures that sufficient servers to achieve the target availability are allocated to the highest priority cluster (i.e., “GOLD”) at all times. During time intervals when all three clusters require large numbers of servers, this impacts the ability of the lower priority clusters (i.e., “BRONZE” and, sometimes, “SILVER”) to realise their target availabilities. Whenever this happens, the two clusters of lower priority are allocated only their required numbers of servers, which represent the minimum values for the $\text{allocatedServers}_2$ and $\text{allocatedServers}_3$ configuration parameters that are considered by the MAPE loop.

**Related work**

The use of runtime quantitative verification for the implementation of utility-function policies in autonomic IT systems was originally proposed in our preliminary work in (Calinescu & Kwiatkowska, 2009b, 2009a; Calinescu, 2009a). We are not aware of any other approaches to using techniques from the formal methods domain to support self-adaptation in autonomic systems.

Utility-function policies for autonomic computing were first advocated in (Walsh, Tesauro, Kephart, & Das, 2004), and have subsequently been used in a significant number of autonomic computing applications, including (White et al., 2004; Kephart & Walsh, 2004; Das et al., 2006; Kusic, Kephart, Hanson, Kandasamy, & Jiang, 2008; Calinescu, 2009a; Werkman, Schoonhoven, Jonge, & Matthijsen, 2010). The implementation of the utility-function policies in these applications often involves solving an integer or mixed-integer programming problem, e.g., (White et al., 2004; Das et al., 2006). Despite the availability of effective solvers for such problems, the solutions they produce cannot guarantee optimality. Using quantitative verification techniques overcomes this limitation.
In other applications, the autonomic systems are characterised by non-linear dependencies between their parameters. The implementation of utility-function policies on these systems can no longer be reduced to solving an integer/mixed-integer programming problem, and the typical approach in this case is to use heuristics such as (Kusic et al., 2008). This represents another scenario in which the approach described in this section supports the effective implementation of utility-function policies.

Figure 22. Autonomic data-centre simulation results for a four-week period in simulated time

Conclusion

Formal methods and their supporting tools have come a long way in recent years. Examples of significant advances in the area include the advent of quantitative verification (Kwiatkowska, 2007) and of “lightweight” formal methods (Jackson, 2006), as well
as the release of effective, open-source model checkers (Holzmann, 2003; Cimatti et al., 1999; Kwiatkowska et al., 2005a) and model finders (Alloy, 2010). The strong commitment to formal methods by the academic research community, and the growing interest that commercial organisations are taking in contributing to this research are indicators that the field will continue to develop rapidly in the near future.

Based on our recent work exploring the application of formal methods to self-managing IT systems (Calinescu & Kwiatkowska, 2009a; Calinescu, 2009a; Kikuchi et al., 2007; Calinescu & Kwiatkowska, 2009b; Kikuchi & Tsuchiya, 2010), we firmly believe that autonomic computing will become one of the main beneficiaries of these developments. Indeed, with autonomic IT systems increasingly expected to provide high levels of predictability and dependability, there is no doubt that using formal methods to support their operation would have a positive outcome. The key question is how to achieve this integration of formal methods into the development, verification and operation of autonomic IT systems. Our chapter looked at three possible approaches to achieving this integration, namely conflict detection in autonomic computing policies, implementation of goal policies, and implementation of utility-function policies. Sample case studies were presented to illustrate these approaches, and to demonstrate the feasibility of using techniques from various areas of formal methods within autonomic IT systems.

While the sample successful case studies presented in the chapter are based on real-world IT systems, it is clear that these applications of formal methods cannot readily benefit any type or size of autonomic IT system. The most significant challenge encountered in this work has been and will remain the state-space explosion (E. M. Clarke et al., 2000) – the often exponential increase in the number of model states with the increase of the system size, and a known limitation of model checking and model synthesis alike. Significant effort will be required to overcome this limitation and improve the scalability of the two techniques, especially when they are employed within a running autonomic system. One opportunity for achieving a substantial improvement that we would like to suggest is based on the concept of incremental analysis of updated models (Langville & Meyer, 2006). When formal methods are applied to autonomic systems, the same system properties are analysed repeatedly for different variants of a system model – these variants correspond to the changes in the system parameters over time. Incremental analysis aims to take advantage of the fact that the differences between the model variants analysed in successive analysis steps are often insignificant, and to speed the process by deriving the analysis results for most steps from the results obtained in previous analysis steps.

Additional challenges in using formal methods in autonomic IT systems include the need to build the system models employed by these methods, and to derive the temporal logic properties or the first-order logic formulas encoding the system constrains, goals and/or utility. Carrying out either of these tasks requires significant expertise. The opportunities for addressing these challenges include the reuse of models or model templates from existing case studies (e.g., (Kwiatkowska et al., 2005a)) and the specification of the properties to analyse in a high-level language (Grunskie, 2008), respectively.

Another opportunity to extend the applicability of formal methods to autonomic computing involves the use of formal techniques for the end-to-end development of autonomic IT systems. Two development platforms that fit this description have been proposed recently (Calinescu & Kwiatkowska, 2009a; Vassev & Hinchey, 2009a). The tool for the development of self-* systems presented in (Calinescu & Kwiatkowska, 2009a) employs a combination of formal software development techniques including model transformation, model-driven code generation and dynamic software reconfigura-
tion (Calinescu, 2009a, 2009b) to add autonomic capabilities to systems whose components can be modelled as Markov chains. Given a PRISM-encoded Markov model of the form described in the previous section, the tool automates or aids (a) the development of the artifacts necessary to build the self-* system; and (b) their integration into a fully-operational self-* solution. The alternative development platform described in (Vassev & Hinchey, 2009a) is a framework that supports autonomic-system specification, validation and code generation. This platform comprises a formal notation and tool support for the definition and validation of autonomic system specifications, and includes generators that translate valid specifications into Java code that implements these specifications (Vassev & Mokhov, 2009; Vassev & Hinchey, 2009b).

Both platforms mentioned above speed up the development of autonomic systems compared to implementing equivalent systems manually. At the same time, they allow developers with limited expertise in formal methods to take advantage of the benefits offered by these techniques. We therefore envisage that this emerging trend to integrate formal methods within domain-independent platforms for the development of autonomic systems will grow to become one of the key areas of autonomic computing research in the future.

Acknowledgements

This work was partly supported by the UK Engineering and Physical Sciences Research Council grants EP/F001096/1 and EP/H042644/1, and by the FP7 European project CONNECT (FP7 231167). Shinji Kikuchi would like to thank to Dr. Tsuneo Katsuyama, Motomitsu Adachi and Satoshi Tsuchiya at Fujitsu Laboratories Limited for their support of this research and for their many insightful comments.

References


Calinescu, R. (2009a). General-purpose autonomic computing. In M. Denko et al. (Eds.), Autonomic computing and networking (pp. 3-30). Springer.


Lens (2010). Lens Computer Science Research Centre. SAT4J. (http://www.sat4j.org/)


