Using Quantitative Analysis to Implement Autonomic IT Systems

Radu Calinescu and Marta Kwiatkowska
Computing Laboratory, University of Oxford
{Radu.Calinescu, Marta.Kwiatkowska}@comlab.ox.ac.uk

Abstract

The software underpinning today’s IT systems needs to adapt dynamically and predictably to rapid changes in system workload, environment and objectives. We describe a software framework that achieves such adaptiveness for IT systems whose components can be modelled as Markov chains. The framework comprises (i) an autonomic architecture that uses Markov-chain quantitative analysis to dynamically adjust the parameters of an IT system in line with its state, environment and objectives; and (ii) a method for developing instances of this architecture for real-world systems. Two case studies are presented that use the framework successfully for the dynamic power management of disk drives, and for the adaptive management of cluster availability within data centres, respectively.

1. Introduction

In order to support the latest advances in research, business and everyday services, software has evolved dramatically over the past decade. For the architects, developers and administrators of the software underpinning today’s ubiquitous, mobile and context-aware IT systems, this evolution has brought significant new challenges. Key among these challenges is the need for software that adapts in a predictable way to continuously changing workloads, scenarios and user objectives.

Autonomic computing represents a promising approach to achieving such adaptiveness by adding self-management capabilities to the components of an IT system [15, 21, 22, 29]. Fig. 1 depicts the high-level architecture of an autonomic computing system. Given a set of user-specified policies (i.e., system objectives), an autonomic manager monitors the system components through sensors, uses its knowledge to analyse their state and to plan changes in their configurable parameters, and implements (or “executes”) these changes through effectors.

We present a new framework comprising a software realisation of the architecture in Fig. 1, and a method for developing instances of this architecture for real-world IT systems. The novel characteristics of our framework are:

1) The knowledge within the autonomic manager consists of a continuous- or discrete-time Markov chain that models the behaviour of the managed IT components.
2) Runtime quantitative analysis of the Markov chain is employed for the analysis step in Fig. 1.
3) The autonomic manager is implemented as a web service that integrates the generic policy engine from [3] and the quantitative analysis tool PRISM [18].
4) Off-the-shelf tools are used for the computer-assisted generation of the manageability adaptor from Fig. 1 starting from the Markov chain.
5) The autonomic manager can be configured dynamically to manage any IT components whose behaviour can be specified by means of a Markov chain. To achieve this, a model of the IT components is generated from the Markov chain, and supplied to a running instance of the autonomic manager.

This framework has several significant advantages over existing approaches to autonomic system development. First, it is generic—our autonomic manager can be reconfigured for use with any legacy or future IT system whose behaviour (that is relevant to the planned autonomic application) can
be specified using a Markov chain. In contrast, current autonomic solutions use dedicated, proprietary autonomic managers for different applications (e.g., [4, 6, 19]). Second, the autonomic systems developed using the framework can implement a rich and flexible set of high-level policies that are based on an exhaustive analysis of the user-specified policies and of the managed IT components. This powerful capability is made possible by our use of runtime quantitative analysis.

2. Background

Quantitative analysis of Markov chains PRISM [18] is a probabilistic model checker/quantitative analysis tool developed by the University of Oxford’s Quantitative Analysis and Verification Group. The tool is the analysis of probabilistic models including discrete- and continuous-time Markov chains (DTMCs and CTMCs) [17] expressed in the PRISM high-level, state-based language.

Cost/reward-augmented versions of probabilistic computational tree logic (PCTL) [9] and continuous stochastic logic (CSL) [1] are used to specify the quantitative properties to analyse for DTMC and CTMC models, respectively [16]. To illustrate this process, we consider the dynamic power management (DPM) of a device with the architecture in Fig. 2. The device consists of a service provider that handles requests generated by a server requester and stored in a request queue. The service provider has several possible states corresponding to different power consumption and service rates, and its state transitions are controlled by a power manager that aims to optimise power consumption while maintaining an acceptable level of service for the device. Fig. 3 shows the PRISM representation of the CTMC model of a Fujitsu disk drive that matches the structure in Fig. 2, and which was introduced in [25].

The rewards...endrewards constructs in Fig. 3 define real values associated with certain CTMC states and transitions. E.g., the “power” rewards construct associates:

- a value of 0.13/referenceTimeInterval with the sp = 0 (i.e., the sleep) state of the service provider
- and a value of 7.0/referenceTimeInterval with its sleep2idle transition, where 0.13 watts is the power consumed by the disk drive

Figure 2. DPM-enabled device

Figure 3. CTMC of a Fujitsu disk drive taken from [25]; a PRISM representation was available from [24].
in the sleep state, 7.0 joules is the energy used for a sleep-
to-idle transition, and referenceTimeInterval > 0 represents
the length of the time interval over which we intend to cal-
culate the expected average power utilisation for the device.
This calculation is performed by PRISM when presented
with the “cumulative reward” property

\[ R\{ "power"\} = ? [C \leq \text{referenceTimeInterval}] \] (1)

Similarly, the quantitative analysis of the PRISM property

\[ R\{ "queueLength"\} = ? [C \leq \text{referenceTimeInterval}] \] (2)
yields the expected average queue length during the first
referenceTimeInterval seconds of operation. For a description
of the techniques used for this analysis see [18].

The properties (1)-(2) can be evaluated for fixed values
of the parameters interArrivalTime (i.e., the request inter-
arrival time) and switchToSleepProbability (i.e., the proba-
bility for the service provider to be transitioned into the low-
power sleep state when it is idle). Alternatively, a PRISM
experiment can be run to analyse these properties for a fam-
ily of CTMCs whose members correspond to different val-
ues of the two parameters—Fig. 4. The autonomic manager
introduced in this paper uses such PRISM experiments to
analyse the current state of the managed IT components,
and to plan changes in their configurable parameters that
fulfil the user-specified policies for the system (cf. Fig. 1).

Our choice of PRISM for this role is due to its proven
ability to analyse real-world systems that spawn a spectrum
of application domains ranging from communication proto-
cols and security systems to biological systems and system
dependability [17, 18]. An extensive, independent perform-
ance analysis of a range of probabilistic model checkers
[13] ranked PRISM as the best tool for the quantitative ana-
lysis of large models such as the ones encountered in the
autonomic systems targeted by our work.

**Autonomic computing policies** Several types of policies
are used in autonomic systems [30, 31]:

- **action policies** provide a low-level specification of how
  the system configuration (i.e., the set of system param-
  eters that can be modified by the autonomic manager)
  should be changed to match its state (i.e., the set of sys-
  tem parameters that can be read but not modified);
- **goal policies** specify precise constraints that the auto-
  nomic manager should meet;
- **utility policies** supply a “measure of success” that the
  autonomic system should optimise.

The use of runtime quantitative analysis within our fram-
ework enhances most the effectiveness of utility policies,
which represent the most flexible of these policy types
[5, 30]. For this reason, the remainder of the paper will
concentrate on our implementation of utility policies. Nev-

ertheless, the policy engine used by the framework supports
all policy types [3].

To illustrate the specification of utility policies, and with-
out loss of generality, we will consider policies requiring the
optimisation of multi-objective utility functions of the form:

\[ \text{utility} = \sum_{i=1}^{n} w_i \text{objective}_i, \] (3)

where the weights \( w_i \geq 0, 1 \leq i \leq n \) are used to express
the trade-off among the \( n > 0 \) system objectives. Each
objective function \( \text{objective}_i, 1 \leq i \leq n \) is an analytical
expression of the system state and configuration.

Fig. 5 depicts two such objective functions for the disk
drive system described earlier in this section, and their com-

\[ \text{combined utility} (u_1 = 100; u_2 = 11; u_3 = 9; u_4 = 120) \]

\[ \text{response-driven objective} (u_1, u_2) \]

\[ \text{power-driven objective} (u_3, u_4) \]
cal form of the utility function is:

\[
utility = w_1 \min \left(1, \max \left(0, \frac{u_1 - \text{queueLength}}{l_1 - l_1}\right)\right) + \\
\quad w_2 \min \left(1, \max \left(0, \frac{u_2 - \text{power}}{l_2 - l_2}\right)\right),
\]

where the lower bounds \(l_i\) and the upper bounds \(u_i\) from the two objective functions partition the value domains of \(\text{queueLength}\) (for \(i = 1\)) and \(\text{power}\) (for \(i = 2\)) into three regions: optimal—for values less than \(l_i\); suboptimal—for values in \([l_i, u_i]\); and zero-utility—for values larger than \(u_i\). This utility function can express a wide range of trade-offs between the power usage and the response time of the disk drive by adjusting the weights \(w_1\) and \(w_2.\)

**Policy engine** The monitor-analyse-plan-execute loop within the autonomic manager in Fig. 1 is implemented by the generic policy engine that we introduced in [2, 3], and which we now extend to use runtime quantitative analysis for the implementation of utility policies.

As described in [3], the policy engine can manage IT components whose specification is supplied to the engine as part of a runtime configuration step. This specification—termed a *system model*—defines the characteristics of every IT component in the system, including its name, type (e.g., *state* or *configuration*) and value domain (e.g., *integer*, *double*, or *string*). System models are represented as XML documents that are instances of a pre-defined meta-model encoded as an XML schema, and introduced in [2, 3]. This choice was motivated by the availability of numerous off-the-shelf tools for the manipulation of XML documents and XML schemas—a characteristic largely lacking for the other technologies we considered. The policy engine is implemented as a .NET web service and takes advantage of object-oriented technology features such as polymorphism, reflection and generics in its handling of IT components whose characteristics are unknown until runtime.

### 3. Generic Method for the Development of Autonomic IT Systems

Our method for developing autonomic IT systems (Fig. 6) comprises three stages that correspond to the sets of steps to be performed by three different user roles:  

1) the manageability adaptor required to organise an existing IT system into an autonomic system is devised by the *system developer* during the *generation* stage;  
2) in the *deployment* stage, this adaptor is deployed, and the policy engine is configured by the *system administrator*;  
3) policies expressing the high-level system objectives are specified by the *end user* in the *exploitation* stage.

These stages are described below, and applied to the development of an autonomic DPM solution for the disk drive introduced in the previous section. The goal of the autonomic system developed in this running example is to ensure that the power manager from Fig. 2 adapts its decisions dynamically to changes in (a) the system workload; and (b) the user-specified utility policy for the DPM disk drive.

#### 3.1. Generation

The first stage of the method starts from a PRISM-encoded Markov chain describing the behaviour of the IT component(s) to be included in the autonomic system. This Markov chain may be available already from the development and/or verification of the IT component(s), or can be built as described in [17, 18]. For our example, we use the CTMC of a Fujitsu disk drive shown in Fig. 3.

**Step G1** In the first generation step, the Markov chain is used to derive an XML model for the configuration of the policy engine. The step is described by the mapping

\[
\text{modelTransformation} : \text{MarkovChain} \rightarrow \text{System},
\]

where \(\text{MarkovChain}\) and \(\text{System}\) represent the set of Markov chains accepted by PRISM and the set of models used to configure the policy engine from [3], respectively. This requires the identification of the parameters of the IT components from the Markov chain by means of two rules:

1) **Uninitialised const** identifiers from the PRISM-encoded Markov chain (cf. Fig. 3) represent *state* or *configuration* parameters of the modelled IT component.
2) **PRISM reward** constructs represent parameters *derived* from the values of the other component parameters.
A basic understanding of the semantics of the Markov chain is needed to distinguish between state and configuration parameters. Therefore, this transformation cannot be fully automated, but is performed in a computer-assisted fashion: the prototype model transformation tool that we implemented requires the system developer to specify the type of these parameters, and generates the other elements of the system model (including the value domain of each parameter) automatically.

As shown in Fig. 3, the disk drive from our running example has four parameters:

- \( \text{interArrivalTime} \) represents the double-valued request inter-arrival time, and is therefore a parameter associated with the system state.
- \( \text{switchToSleepProbability} \in [0, 1] \) is a configuration parameter that specifies the probability for the service provider to be transitioned into the low-power \( \text{sleep} \) state when it is \( \text{idle} \).
- \( \text{power} \) and \( \text{queueLength} \) are two \( \text{double-valued} \) derived parameters.

In addition to these parameters, the generated model is augmented automatically with a \( \text{string-valued} \) parameter \( \text{id} \) that is used to distinguish between different instances of the modelled IT component, and a \( \text{behaviouralModel} \) parameter whose \( \text{(string)} \) value is the Markov chain itself.

**Step G2** In this step, a standard XSLT engine (e.g., Saxon [27] in our prototype toolset) is used to apply a simple XSL transformation to the XML system model, and thus to automatically extract an XML schema specification for the targeted IT component(s):

\[
\text{schemaGen} : \text{System} \rightarrow \text{XmlSchema}. \quad (6)
\]

The result of this step for our running example is shown in Fig. 7. Note that for each derived component parameter with identifier \( x \), a \( \text{string-valued} \) element \( x\text{Definition} \) is automatically included in the XML complexType for the component (e.g., \( \text{powerDefinition} \) and \( \text{queueLengthDefinition} \) for the disk drive XML schema in Fig. 7). These elements are used to store the PRISM temporal logic properties that the policy engine will use to calculate the value of the derived component parameters at runtime.

**Step G3** A standard data type generator (we use the XML Schema Definition tool [20]) is employed to generate the set of data types associated with the XML schema:

\[
\text{dataTypeGen} : \text{XmlSchema} \rightarrow \mathbb{P}\text{DataType}. \quad (7)
\]

This is a set of .NET classes in our framework.

**Step G4** Finally, a simple transformation was implemented to automate the generation of manageability adaptor stubs for the components in the IT system:

\[
\text{adaptorGen} : \text{XmlSchema} \rightarrow \mathbb{P}\text{ManageabilityAdaptor}. \quad (8)
\]
state, new data types and manageability adaptor proxies are generated within the engine to enable it to interoperate with the manageability adaptors for the component types specified in the system model. This fully automated code generation process is described in [3].

**Step D2** The second development step consists of setting up the manageability adaptors built during the generation stage, and connecting them to the actual IT components to be included in the system. The first part of the operation represents a standard deployment of a web service, whereas the second part depends on the interface between the adaptor and the system components, but it typically involves configuring the two elements so that they know each other’s address. For the system used in our running example, the URL of the discrete-event simulator was provided to each other's address. For the system used in our running example, the URL of the discrete-event simulator was provided to each other's address.

### 3.3. Exploitation

**Step E1** In this step, user-specified policies that express the objectives of the system as functions of its parameters are devised and supplied to the policy engine, e.g., by using the web client that was employed to upload the system model in **Step D1**. Typically, these policies are modified over time to reflect changes in the system goals. Depending on the configuration of the policy engine, the policies are evaluated and implemented either periodically or when the engine receives notifications about changes in the values of the system parameters from the manageability adaptors.

The policy handling techniques employed by the policy engine version that did not support quantitative analysis are described in [3]. In this section, we describe for the first time how the extended version of the policy engine is handling utility policies that rely on the use of runtime quantitative analysis. This will be done for the utility policy:

\[
\text{MAXIMISE}(\text{scope}, \text{utility}, \text{configParam}, \text{min}, \text{max}, \text{step}, \text{mc})
\]

that uses the knowledge encoded in the Markov chain mc to set the value of the configuration parameter configParam for each IT component within the scope of the policy. Runtime quantitative analysis is employed to examine all configParam values that are step units apart between min and max, and to select a value that maximises a multi-objective utility function with the form in (3).

The arguments in (9) are expressions that evaluate to a set of IT components (scope), a numerical value (utility, min, max and step) and the string-valued name of a component property (configParam and mc), respectively. For instance, the policy used for our running example is

```
MAXIMISE("TYPE(.) = "diskDrive", utility, "switchToSleepProbability",
```

where the utility function to optimise is given by (4).

Note that when the choice of step is such that the Markov chains corresponding to all possible configParam values are analysed, the optimal value will be identified and used at all times. When this is not feasible (e.g., for the switchToSleepProbability disk-drive parameter in (10)), the analysis may be impacted by local maxima. Offline quantitative analysis of the underlying Markov chain is then required to confirm the absence of local maxima and/or to choose the step parameter of policy (9) such that the resulting policy is effective. For our autonomic DPM system, the latter was made possible by the analysis available from [24, 25] and led to the choice of step = 0.05.

The algorithm used to implement policy (9) within our policy engine is described by the pseudocode for the MAXIMISE function in Fig. 9. As policy (9) is local to each component in scope, the top-level foreach loop starting in line 1 implements the policy for each such component c in turn.

First, lines 2–9 of the algorithm synthesise the three parameters required to run the command-line version of the PRISM tool for component c: the Markov chain to analyse (markov); the quantitative properties to evaluate (propertyList); and the value ranges for the PRISM parameters in the Markov chain (constList). In line 2, reflection is used to obtain and invoke the Get method for the parameter of c whose name is provided as the mc argument of MAXIMISE. The result is stored as a string in markov. The foreach loop in lines 4–8 uses the TYPE operator to find out the type of c, looks up this component type in the SYSTEMMODEL supplied to the policy engine in the deployment step D1, and examines each of the component parameters. Component parameters of type ‘derived’ correspond

\[\text{MAXIMISE}(\text{scope}, \text{utility}, \text{configParam}, \text{min}, \text{max}, \text{step}, \text{mc})
\]

1 foreach c ∈ scope do
2     markov = INVOKE(c, GETMETHOD(\text{PARAMETER}(c, \text{p.ID} = "Definition"))).\{ \}
3     propertyList = ".", constList = "."
4    foreach p ∈ SYSTEMMODEL(\text{TYPE(c).parameters}) do
5       if p.type = "derived" then
6           propertyList += INVOKE(c, GETMETHOD(\text{PARAMETER}(c, \text{p.ID} = "Definition"))).\{}\{ + ,
7           else if p.ID ≠ configParam & p.ID ≠ mc then
8               constList += p.ID + + .
9           end if
10       end foreach
11       result ∈ prismResults such that
12           EVAL(utility, c|result) = \max\{\text{prismResults}.EVAL(utility, c|result)\}
13           configParam = \{\text{result(configParam)}\}
```

Figure 9. Implementation of the utility policy (9) using runtime quantitative analysis

\[0, 1, 0.05, "behaviouralModel"\]
to reward constructs in the PRISM-encoded Markov chain (cf. Fig. 3), and the definitions of their associated temporal logic properties are concatenated into \texttt{propertyList} (lines 5–6). The values of all other component parameters (except \texttt{configParam} and \texttt{mc}) are used to synthesise the comma-separated list of PRISM parameter values \texttt{constList} in lines 7–8. The PRISM encoding of the \texttt{configParam} values to be analysed (i.e., \texttt{"configParam=min:max:step"}) is then appended to the resulting \texttt{constList} in line 9.

The PRISM experiment is run in line 10, and the results of this runtime quantitative analysis are parsed into an \( \left( \left\lfloor \frac{\text{max} - \text{min}}{\text{step}} \right\rfloor + 1 \right) \)-element array \texttt{prismResults}. For each analysed \texttt{configParam} value \( x_i = \text{min} + i \cdot \text{step} \), \( 0 \leq i \leq \left\lfloor \frac{\text{max} - \text{min}}{\text{step}} \right\rfloor \), \texttt{prismResult}[i] is a dictionary that maps \texttt{configParam} to \( x_i \), and the name of each ‘derived’ parameter of \( c \) (e.g., "power" and "queueLength" for a \texttt{diskDrive} component) to the value that the parameter would take if \texttt{configParam} = \( x_i \).

In line 11, the \texttt{utility} function is evaluated for the considered values of \texttt{configParam}. For each dictionary \( x \in \texttt{prismResult} \), the parameters of \( c \) that \( x \) maps to values are set to these values, and \texttt{utility} is calculated for the resulting object \( c | x \), using an auxiliary function \texttt{EVAL}. One of the \texttt{prismResult} elements that maximise utility is chosen as the analysis result, and used in line 12 to set the value of the \texttt{configParam} field of \( c \). On exit from \texttt{MAXIMISE}, the new \texttt{configParam} values for the IT components in \texttt{scope} will be enforced using the appropriate manageability adaptors, as described in [3].

The outermost loop starting in line 1 is performed \#\texttt{scope} times. Lines 2–3 take constant time, and so does the \texttt{foreach} loop in lines 4–8 because an IT component has a constant number of parameters to examine and handle. Line 9 takes constant time, while running PRISM to analyse a temporal logic property takes \( t_{\text{PRISM}} > 0 \) time units. A detailed analysis of \( t_{\text{PRISM}} \) is available in [17]; here, we will just state that \( t_{\text{PRISM}} \) is linear in the size of the analysed property, and quadratic in the state size of the analysed Markov chain. Evaluating \texttt{utility} for a single \texttt{configParam} value in line 11 takes constant time, hence choosing a value that maximises \texttt{utility} will take \( O(\frac{\text{max} - \text{min}}{\text{step}}) \) time. Finally, setting the value of \texttt{configParam} in line 12 takes constant time. The overall complexity of the algorithm is therefore \( O(\#\texttt{scope} \cdot t_{\text{PRISM}} + \frac{\text{max} - \text{min}}{\text{step}}) \)—linear in the number of IT components to which autonomic capabilities are added, in the time taken to analyse the Markov chain using PRISM, and in the number of analysed \texttt{configParam} values.

Step E2 The policy engine detects automatically all manageable IT components that have been registered with its component discovery service, and whose types are specified in the system model used for its configuration.

4. Case Studies

4.1. Dynamic Power Management

We start our presentation of case studies with the experimental results for the autonomic DPM system from Sect. 3. A discrete-event simulator was implemented to evaluate policy (10) for a simulated disk drive handling requests with exponentially distributed inter-arrival time. Each experiment was for a one-hour period in simulated time. The mean request inter-arrival time was varied in the range 200ms to 2000ms in each experiment, and the parameters \( w_i, l_i, u_i, 1 \leq i \leq 2 \) from (4) were varied across experiments. This allowed us to assess the ability of the system to adapt to changes in its workload and objectives, respectively. Additionally, two standard DPM methods were also applied to each of the experimental data sets in order to contrast the autonomic DPM method with existing DPM techniques. The two standard DPM methods selected were ([25]): the \textit{timeout method}, which moves the disk drive into the sleep state after a period of idleness \( t \) and “awakens” it immediately after a request has arrived; and the \textit{N method}, which moves the disk drive into the sleep state as soon as it becomes idle, and “awakens” it after \( N \) requests accumulate in its queue.

The experimental results are summarised in Fig. 10. The graphs contrast the average actual utility achieved by our autonomic disk drive with the average actual utilities achieved by the timestamp and \textit{N} methods. For the timestamp method, different experiments were run for each considered disk drive workload, and for each 500ms-apart value of \( t \) in \([0, 10s]\). The results shown in Fig. 10 correspond to the choice of \( t \) yielding the highest average actual utility, so that the autonomic method is compared with the best possible instance of the timestamp method. Likewise, \textit{N}-method experiments were run for all \( 0 \leq N \leq 10 \), and the best result is represented in the graphs. The actual utilities are shown as a percentage of the maximum value of the utility function (4) (i.e., \( w_1 + w_2 \)), for multiple values of the \( w_1/w_2 \) ratio between the weights of the power-related and performance-related terms of the utility function. The chosen \( w_1/w_2 \) ratios cover a broad range of power/performance trade-offs, (i.e., \( w_1/w_2 \in \{0.1, 0.2, \ldots, 1.0, 2.0, \ldots, 10.0\} \)), and are represented on the horizontal axes using logarithmic scale.

Each graph corresponds to a different combination of the \( l_i, u_i \) parameters of the utility function (4), \( 1 \leq i \leq 2 \). For the top row of three graphs, the performance-related utility parameters are fixed at relatively demanding values (i.e., \( l_1 = 8 \) and \( u_1 = 9 \)), and the power-related requirements are relaxed from left to right (cf. Eq. (4) and Fig. 5). The graphs in the bottom row correspond to much more relaxed performance requirements (i.e., \( l_1 = 18 \) and \( u_1 = 20 \)), and to the same power-related parameters \( l_2, u_2 \) as the top row.
Note that a straightforward way to enable users with limited IT expertise to select an effective policy instance for their needs is to present them with a graphical user interface (GUI) comprising a “power/performance trade-off” slider GUI control that adjusts the $w_1/w_2$ ratio, and a radio-button GUI control for selecting a family of policy instances such as the ones corresponding to the six graphs in Fig. 10.

The experimental results show that the autonomic disk drive achieves better utility than the timeout and N methods across a broad range of performance/power trade-offs. For all experiments, the autonomic DPM method betters the other methods by a wider margin for small to medium values of $w_1/w_2$, i.e., when the power-related term of (4) has a major contribution to the overall utility. In these areas of the graphs, the autonomic DPM method adjusts the configuration of the disk drive to its workload in ways that are not accessible to the timeout and N heuristics for any considered value of their parameters $t$ and $N$, respectively.

For large values of $w_1/w_2$, the three DPM methods achieve similar actual utility. The explanation is that for these $w_1/w_2$ ratios the power-related term of the utility function (4) is dwarfed by the performance-related term, and both standard methods can maximise the performance-related term for certain values of their parameters. These values are $t = 10$s for the timeout method (i.e., an idle disk drive wakes up as soon as a request enters its queue). Hence, DPM is of limited use when the utility function is nearly independent of the power usage; in this case, the utility value can be optimised by simply maintaining the disk drive active at all times.

The policy evaluations were done at regular, 10-second time intervals. For our simple CTMC, the execution of the algorithm in Fig. 9 took sub-second time on a 3GHz dual-core AMD-processor desktop running Windows XP. The average CPU overhead was in the range 1.5%-2.5%, and techniques for decreasing it are discussed in Sect. 6.

As a final remark, note that when applied to hardware IT components such as disk drives, our method provides functionality similar to that of automatic control applications. What distinguishes our method from these applications is the ability to reconfigure the policy engine at its core for use across application domains, and its model-driven approach to IT component management using autonomic policies to express the high-level system objectives.

4.2. Cluster Availability Management

The second case study involves the adaptive control of cluster availability within a data centre. The objective of this application is to control the number of servers allocated to the $N \geq 1$ clusters of a data centre in order to maximise...
the utility function

\[
\text{utility} = \sum_{i=1}^{N} w_i \cdot \text{GOAL}(\text{availability}_i \geq l_i) - w_0 \sum_{i=1}^{N} \text{servers}_i
\]  

subject to

\[
\sum_{i=1}^{N} \text{servers}_i \leq \text{Total_servers and required}_i \leq \text{servers}_i, \quad (12)
\]

where \(w_i > 0\), \(\text{availability}_i \in [0, 1]\), \(l_i \in [0, 1]\), \(\text{required}_i \geq 1\) and \(\text{servers}_i \geq 1\) represent the priority, expected availability, target availability, number of required servers, and number of (allocated) servers for cluster \(i\), \(1 \leq i \leq N\), respectively. The parameters \(w_i\) and \(l_i\) have the same role as in (4), but we used the operator \(\text{GOAL}(\text{availability}_i \geq l_i)\) instead of \(\min \left(1, \max \left(0, \frac{u_i - \text{availability}_i}{u_i - l_i}\right)\right)\) like in the objective functions from (4) in order to express the degenerate case when \(u_i = l_i\). The \(\text{GOAL}\) operator yields 1 when its argument is true and 0 otherwise, \(\text{Total_servers} \geq 1\) is the total number of servers in the data centre, and \(0 < w_0 \ll 1\) is a constant.\(^5\) The expected availability of cluster \(i\), \(\text{availability}_i\), was defined as the expected fraction of a one-year time period during which at least \(\text{required}_i\) servers are usable (i.e., they are operational, connected to an operational switch, etc.), assuming that the cluster configuration does not change.

As the utility function (11) is defined across all clusters within a data centre, the associated policy is a global policy. Its implementation is achieved using a variant of the algorithm in Fig. 9 that combines the per-component results of the runtime quantitative analysis instead of deciding the value of \(\text{servers}_i\), \(1 \leq i \leq N\) on a per cluster basis.

Like in the previous case study, we applied the method introduced in Sect. 3 starting from an existing Markov chain representation of the targeted IT component, i.e., the CTMC of a dependable cluster of workstations introduced in [10]. This CTMC takes into account the failure and repair rates of all components from our targeted cluster architecture—shown in Fig. 11. Consequently, the policy engine can use PRISM experiments to calculate the expected cluster availabilities for the data-centre configurations that satisfy (12), and to decide the number of servers that each cluster should get so that the value of the utility function (11) is maximised. Given the complexity of the CTMC behavioural model, we configured the policy engine to employ the notification mechanism of the cluster manageability adaptor, and recalculate the server allocations only when the system state changed. With all possible data-centre configurations analysed by the PRISM experiments, this calculation took up to 30 seconds with the policy engine running on the same desktop PC as before. This response time is acceptable for our use case because, based on our previous experience with policy-based data centre management [4], half a minute represents a small delay compared to the time required to provision a server when it is allocated to a new cluster.\(^6\)

The \textit{step} parameter of policy (9) was set to 1, so that all possible values of the configuration parameter \text{servers}, were considered during the runtime quantitative analysis, and the

\(^5\)The second term of (11) ensures that when multiple configurations maximise the first sum, the configuration using the fewest servers is preferred.

\(^6\)Sect. 6 suggests techniques for working around the time taken by runtime quantitative analysis when such delays are not acceptable.
chosen configuration was guaranteed to be optimal. For this reason, we present only a set of typical experimental results for this case study (Fig. 12) rather than contrasting these results with those corresponding to other availability management techniques. Maximising the three-cluster data centre utility involves allocating sufficient servers to achieve the target availability of the highest priority cluster 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 SILVER and BRONZE to realise their target availabilities, and only the required numbers of servers are allocated to these clusters. The second term of the utility function (11) ensures that servers are not allocated to clusters unless this can improve the system utility. Keeping more servers unassigned in this way is advantageous, as the overhead to provision a spare server for inclusion into a cluster is significantly lower than the overhead to move it from one cluster to another.

5. Related Work

A number of other projects have addressed the model-based development of autonomic systems. In [14], the authors define an autonomic architecture meta-model that extends IBM’s autonomic computing blueprint [12], and use a model-driven process to partly automate the generation of instances of this meta-model. Each instance is a special-purpose organic computing system that can handle the use cases defined by the model employed for its generation. Our framework eliminates the need for the 19-activity generation process described in [14] by using a generic policy engine that can be dynamically reconfigured to handle any use cases in which the behaviour of the system components can be modelled by means of a Markov chain.

Several research projects propose the use of Model-Driven Architecture (MDA) techniques to develop autonomic computing policies and autonomic systems starting from high-level behavioural models of the system or of its components [7, 23, 26]. Two of these approaches [7, 23] are targeted at bespoke systems whose components already exhibit sophisticated autonomic behaviour, and therefore cannot be readily extended to handle generic legacy components. In contrast, our generic framework can accommodate any type of IT component for which a probabilistic model is available. The preliminary work described in [26] is closer to our approach in that it advocates the importance of using MDA techniques in the development of generic autonomic systems; however, the authors do not substantiate their proposal with any concrete solution, but rather qualify it as an open challenge. Our framework represents an important first step towards addressing this challenge.

In [11], the authors take a view similar to ours by introducing a paradigm termed model-driven autonomic computing, and explaining that the model-based validation of self-management decisions represents a more reliable and flexible approach than the use of pre-set policies. A hierarchical model of NASA’s Autonomous Nano-Technology Swarm missions is successfully used in [11] to achieve the self-managing functionality that these missions depend on, and thus to illustrate the benefits of the approach. Our work complements the results in [11] with a new model-based approach to developing autonomic functionality, and proposes a generic method that uses existing tools and standards for the implementation of autonomic systems.

6. Conclusion

Boosted by the development of powerful quantitative analysis tools [13, 16], the use of probabilistic model checking for the static analysis of the quantitative properties of real-world systems has become increasingly popular in recent years [17]. In this paper, we extended the use of Markov chains and quantitative analysis to the development and runtime operation of autonomic IT systems.

Starting from a PRISM-encoded Markov chain available from the verification of a legacy IT system or built as described in [17, 18], the generic method for the development of autonomic functionality introduced in the paper uses automated and computer-assisted techniques to devise a self-managing version of the original system. The three stages of the method can be carried out by a system developer with experience in quantitative analysis using PRISM (the generation stage); by a system administrator with minimal experience in web service deployment (deployment); and by a technically competent user (exploitation).

The paper describes for the first time an extended version of the policy engine from [3] that uses runtime quantitative analysis to implement powerful user-specified policies that would have been extremely complicated to express and support otherwise. The runtime use of quantitative analysis involves the automated synthesis of the parameters for the execution of a PRISM experiment [18], including the generation of appropriate values and value sets for the undefined constants in the PRISM Markov chain, and the assembly of the PRISM probabilistic temporal logic properties to be evaluated. The results of the quantitative analysis are then automatically interpreted and used to select an appropriate new configuration for the system, ensuring that it continually adapts to changes in its environment and in the user-specified policies.

Runtime quantitative analysis can incur significant overheads in terms of both response time and resource (e.g., CPU and memory) utilisation. The use of a subscription-notification mechanism as described in Sect. 4 is one way to reduce these overheads by avoiding the unnecessary periodical re-evaluation of policies. Several other options that
we are planning to investigate include: the use of caching and pre-evaluation techniques to bypass the analysis step during policy evaluation; performing the quantitative analysis within the process space of the policy engine; and the use of a hybrid approach in which a less demanding PRISM experiment is carried out to produce a close-to-optimal configuration for the autonomic system and a faster technique is then used to refine this configuration.

Other areas of future work are the development of additional applications to assess the effectiveness of our framework within new application domains; and the design of policies supporting an extended set of probabilistic system properties, along the lines of the results reported in [8]. Finally, a major challenge for the future is the development of online machine learning techniques for the synthesis and/or refinement of a Markov-chain model for an IT system, e.g., starting from an approximate behavioural model provided by its administrator.

Acknowledgement This work was partly supported by the UK Engineering and Physical Sciences Research Council grant EP/F001096/1.

References