INTERNATIONAL JOURNAL OF NETWORK MANAGEMENT
Int. J. Network Mgmt 2012; **22**: 104–130
Published online 19 July 2011 in Wiley Online Library (wileyonlinelibrary.com) DOI: 10.1002/nem.790

A^{2}THOS: availability analysis and optimisation in SLAs

Emmanuele Zambon[1,*†], Sandro Etalle[1,2] and Roel J. Wieringa[1]

[1]*University of Twente, Enschede, The Netherlands*
[2]*Technical University of Eindhoven, Eindhoven, The Netherlands*

SUMMARY

Information technology (IT) service availability is at the core of customer satisfaction and business success for today's organisations. Many medium- to large-size organisations outsource part of their IT services to external providers, with service-level agreements describing the agreed availability of outsourced service components. Availability management of partially outsourced IT services is a non-trivial task since classic approaches for calculating availability are not applicable, and IT managers can only rely on their expertise to fulfil it. This often leads to the adoption of non-optimal solutions. In this paper we present A^{2}THOS, a framework to calculate the availability of partially outsourced IT services in the presence of SLAs and to achieve a cost-optimal choice of availability levels for outsourced IT components while guaranteeing a target availability level for the service. Copyright © 2011 John Wiley & Sons, Ltd.

Received 16 April 2010; Revised 19 April 2011; Accepted 19 April 2011

1. INTRODUCTION

Nowadays, the information technology (IT) infrastructure of most large organisations is so complex that it is often organised in terms of *services* that are offered as part of an internal market in which different business units offer and buy IT services to and from each other. In some cases, services are acquired from an external organisation rather than from an internal business unit (outsourcing). Typically, services offered by an internal provider are customised and tailored to support the business goals of the organisation, while those offered by external providers are standardised and large scale, and therefore are less specific but potentially cheaper than those implemented internally. In some cases, internal providers outsource some sub-services to external ones, for instance when it lacks specific competencies (e.g. SAP configuration). This is a so-called mixed sourcing strategy.

Regardless of whether the service is bought internally or externally, the terms and conditions of the contract are determined in a service-level agreement (SLA). Figure 1 summarises the concept of mixed-sourced IT services regulated by SLAs.

In this paper we focus on IT service availability, which is at the core of customer satisfaction and business success for organisations [1], and indeed it is a major element in any internal or external IT SLA. In fact, a typical IT SLA includes hard clauses on the *minimal availability* of the service offered (for example, it may include that the service should not be 'down' for more than 2 hours per week, and a penalty fee for each week in which this is not satisfied).

Now, the two concerns we focus on (and at the same time the two questions to which we provide an answer within the limits of the settings of this paper) are:

1. How can a business unit check and/or guarantee that a given (offered) service will respect some given minimal availability levels?
2. As 1, while minimising costs.

*Correspondence to: Emmanuele Zambon, University of Twente, Enschede, The Netherlands.
E-mail: emmanuele.zambon@utwente.nl

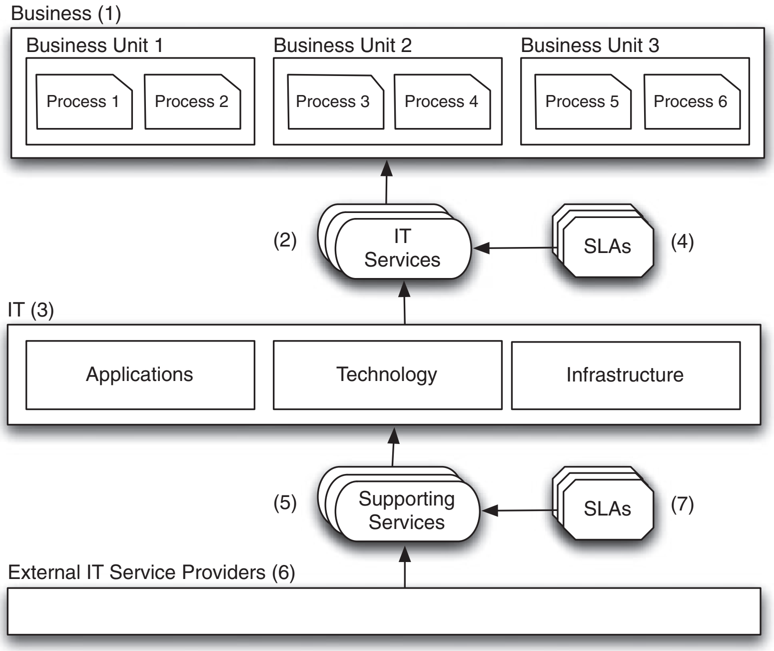

Figure 1. Mixed-sourced IT service provision regulated by SLAs. The organisation's business units (1) use IT services (2) provided by the internal IT department (3). These services are regulated by means of SLAs (4). In turn, the IT department is using supporting services (5) offered by an external provider (6) to run (part of) the IT infrastructure. Also these services are regulated by means of SLAs (7)

Let us elaborate on these two points and explain why they are not only relevant but also non-trivial problems.

An IT service is usually offered by a system consisting of several *components*. These components can interact in non-trivial ways: for instance, a component could be crucial to the service in a way that if the component is unavailable then theservice becomes unavailable as well; other components may be organised in such a way (e.g. exploiting redundancy) that only if a number of them fails will the service be affected. In addition, a component may depend in a non-trivial way on sub-services which are in turn regulated by other SLAs.

To ensure that the minimal service availability remains within the agreed margins, IT managers can take reactive (e.g. monitoring, measuring) and/or proactive measures. A key proactive measure is planning and designing service availability when services are created or changed. At the business level, planning service availability allows the service provider to set availability figures on the SLAs that both satisfy customer needs and can be guaranteed by the technical infrastructure providing the service. To achieve this at thetechnical level the service provider needs to (a) calculate the availability of the IT system providing the service(s) based on the information available on system components, and (b) make appropriate system design choices to support a specific availability level by selecting the system components based on their contribution to the availability of the system.

Reliability studies have introduced a number of (by now) standard techniques (e.g. continuous-time Markov chains (CTMC) [2] and Petri nets [3]) which allow one to compute system availability when the mean time between component failures and the mean time to repair a component is known. However, in the context of mixed-sourced IT services, this information is usually not available. Instead, SLAs between the external and the internal provider typically only include the extitminimal guaranteed availability of the component. Therefore, it is not possible to apply these standard techniques to calculate the system availability (see Section 2 for details).

Regarding the second point, the selection and negotiation of sourcing contracts is a complex problem which involves the establishment of business relations between different organisations. To select sourcing contracts in an optimal way is the subject of scientific studies in the field of economic

DOI: 10.1002/nem

game theory, such as the principal agent model (see Section 2 for further details). In this paper we focus on one contractual scenario, in which the contract between the organisation and the service provider is composed from a number of 'standard' building blocks arranged in a service catalogue. Each building block describes one single service (e.g. the management of a UNIX server) and allows the customer to choose the level of a number of quality attributes among a standardised set, with associated prices. One of these quality attributes is availability. Different availability levels (e.g. gold, silver and bronze) have different associated prices, and the customer can choose a level according to its needs. Due to economies of scale, the cost of a certain service (with a fixed configuration) decreases for the outsourcer as the purchased volume increase. Therefore, graduated prices are usually agreed so that also the price for the customer decreases when volumes increase. In order to fully benefit from this kind of sourcing contract, the customer needs to select the quality level of the different available attributes for the different building blocks in such a way that the delivered composite service, which relies on these building blocks, has the expected quality and that the cost is minimal. Considering availability, this is a non-trivial optimisation problem: one needs to determine the combination of minimal availability levels for the sub-services in such a way that the total cost is minimal, while ensuring that the resulting service achieves the availability specified in the SLA of the composite service. This cannot be solved without the use of specific optimisation algorithms and typically IT managers choose non-optimal, conservative solutions.

1.1. Contribution

We present A^{2}THOS, a framework for the analysis and optimisation of the availability of mixed-sourced IT services. The framework consists of (1) a modelling technique to represent partially outsourced IT systems, their components and the services they provide, based on AND/OR dependency graphs, (2) a procedure to calculate (a lower bound of) the system availability given the (lower bounds of) components availability, and (3) a procedure to select the optimum availability level for outsourced components in order to guarantee a desired target availability level for the service(s) and to minimise cost.

An AND/OR dependency graph is an AND/OR directed acyclic graph in which nodes represent system components and services, and edges between nodes represent the functional dependency of one node with the other. We use the graph in order to calculate a state function describing the availability of each service based on the state of the components (operational or not operational). We then use the state function and information about component availability to determine a lower bound for the availability of the service, by setting up a linear programming problem. Based on this procedure, we finally present the procedure to set up an integer programming problem which allows one to determine a cost-optimal combination of availability levels for outsourced components in order to guarantee a target service availability. We show the practical use of A^{2}THOS by implementing it in a tool which we apply to the service availability planning of an industrial case.

1.2. Limitation of the approach

A^{2}THOS uses an AND/OR dependency graph to represent IT systems; thus it is unable to explicitly represent failure recovery mechanisms such as availability of spare parts. Spare parts are used to implement warm and cold standby mechanisms. For example, to shorten the downtime caused by a server breakdown, the system administrators can keep another server ready to replace the broken one. This second server is the spare part. When it is always running (but not operating) and the workload of the broken server is automatically routed to the spare server, this mechanism is called *hot standby*. When the workload of the broken server needs to be manually routed to the spare server, this mechanism is called *warm standby*. When the spare server is not readily available, but it needs a setup phase before the workload of the broken server can be redirected to it, the mechanism is called *cold standby*. Our representation allows us to explicitly model hot standby mechanisms by using OR nodes, but it is not applicable in case of warm and cold standby mechanisms. We share this limitation with other well-known modelling techniques, such as traditional fault trees (FTs) and reliability block diagrams (RBDs).

1.3. Organisation

The rest of the paper is organised as follows. In Section 2 we present related work in the fields of economic game theory and reliability. In Section 3 we provide the mathematical foundation for calculating the minimal guaranteed serviceavailability. In Section 4 we present a procedure to find the optimal choice of availability level for outsourced components. In Section 5 we describe the tool we created to implement the A^{2}THOS framework and the benchmarks we conducted to test its scalability performance. Finally, in Section 6 we show how we applied A^{2}THOS to a practical case of service availability planning in an industrial context. Appendices A and B present the proof of one of the theorems we introduce and the representation capabilities of our model with respect to RBDs, respectively.

2. RELATED WORK

The problem we address in this paper pertains to two main research areas: economic game theory and reliability theory.

2.1. Economic game theory

Simplifying a bit from the complex reality of IT management, the general goal of IT managers within an organisation is to provide the IT services required by the organisation business units while remaining within their (limited) budget. The provided IT services need to comply with the functional and non-functional requirements of business units. On the other hand, and applying the same simplifications, the goal of business unit managers with regard to IT services is to obtain the IT services needed to run (or improve) business operations at the lowest possible price, in order to maximise profits. It is easy to see that there is a conflict on the economic aspect of these two goals. In the same way, the goal of IT managers conflicts similarly with the goals of external IT service providers. This kind of conflict can be modelled using game theory [4].

Burgess [5] proposes to use *promise theory* [6] as a framework for describing the inter-company dependency (outsourcing) and the free corporate collaboration in a market economy that occurs in the context of a service-oriented architecture. In promise theory, the actors of the service-oriented architecture are called agents and represented as graph nodes. An agent can be a person (e.g. a manager), a company or business unit (e.g. an outsourcer), a physical entity (e.g. documentation) or even a logical process. The cooperation between agents is modelled as a directed edge in the graph, labelled with a *promise*. A service from one node to another is therefore modelled as a kind of promise that can be made. In this view, SLAs can be modelled in the body of the promise as constraints, which indicate exactly what is being promised. Using agents and promises, one can model the different services provided between different entities and can identify agent properties such as functional roles (e.g. when different agents all make the same kind of promise).

If promise theory can be used to model existing SLAs, the problem of contract selection can be addressed by means of the *principal–agent model* [7,8]. In this model, a principal wants to outsource one or more tasks to a set of agents. The principal–agent model is based on the following assumptions [9]: (1) both parties have rational behaviour and rational expectations and interact on the basis of institutions, like freedom of contract and private property; (2) the actions undertaken by the agents have external effects on the principal's profit and success; (3) the agent has discretionary freedom due to asymmetric and incomplete information with regard to the principal's activities; and (4) the agent shows opportunistic behaviour to maximise its own expected profit, instead of acting in line with the goals of the principal. Agents can fall into a variety of different types that indicate their relative costs and abilities. In the case we are considering in this paper, the principal would be the organisation (or its IT manager) and the agents would be the different IT service providers. The principal will make promises about the wages to offer to the different agents, while the agents will make promises about the 'effort', or services, they will commit on. Various mechanisms (incentives) may then be used to try to align the interests of the agent in solidarity with those of the principal, (e.g. piece rates/ commissions, profit sharing, efficiency wages, performance measurement).

DOI: 10.1002/nem

Gellings [10] analyses outsourcing relationships by applying the principal–agent model. The theoretical findings stress that within an optimal contract the marginal utilities of both parties involved have to be proportional. In practice this can be achieved by two possibilities. First, during contract negotiation, the two parties have to disclose their cost calculations. Based on this information the two parties can agree on a pricing scheme that is advantageous for both and set a price cap combined with a renegotiation option to ensure proportionality of the respective cost functions. In the second possibility, both parties can agree upon a pricing scheme that reflects proportionality of cost functions without exactly disclosing them (e.g. by means of graduated prices that reflect the realisation of economies of scale on the side of the service provider). This second possibility reflects the scenario we analyse in this paper, in which standard options are defined for outsourced services in which prices depend both on the effort (i.e. the quality level of the service) and on the margins that the outsourcer can gain because of economies of scale. In fact, the analysis presented in this paper and focused on the optimisation of IT availability costs is based on the same basic assumptions of the principal–agent model, and on one of the scenarios identified by means of the application of the principal–agent model to the context of IT outsourcing.

2.2. *Reliability theory*

Our work relates to four main topics discussed in reliability studies: (1) the general approach to calculate system availability; (2) modelling techniques to represent the system under analysis; (3) existing tools; and (4) other approaches taking into account availability to optimise IT service composition.

2.2.1. *The general approach*

Referring to a classic formulation [11] taken from reliability theory, a *repairable system* is a system which can be repaired after a failure.

In the simplest case, the system m for which availability must be determined is represented by the state function $\chi(m, t)$, which assumes value 1 if m is operating within tolerances at time t, 0 otherwise. The general way of calculating the availability of a repairable system is to assume it has an independent, exponential distribution of failure and repair time (a so-called stationary alternating renewal process [12]). However, to do so one must know at least two properties of the system: its failure rate λ; and its repair rate μ. The first property specifies how often the system will fail on average, i.e. its mean time between failure (MTBF): $\lambda = \frac{1}{\mathrm{MTBF}}$. The second one specifies its mean time to repair (MTTR): $\mu = \frac{1}{\mathrm{MTTR}}$. Under this assumption the limiting availability is then obtained using the formula $\bar{A} = \frac{\mu}{\mu + \lambda}$.

In the general case, the system can assume more than two states. Such a system is called *complex*. A complex system is a system which is made of interconnected components that as a whole exhibit one or more properties depending on the properties of the individual components. For example, a complex system can be made of two ‘simple’ components (i.e. two components that can independently be either in operative or in repairing state). The state of the system depends on the state of the two components: the system may work properly even if one component only is operative, or it may need both components to be operative. To model the state of the system, a state formula is used. Components can have more than two states (e.g. operative, planned maintenance, emergency repair). To compute the availability of complex systems, CTMC [2] or Petri nets [3] are used. To employ such techniques, one has to (1) define a state formula of the system based on the component’s state, and (2) know the transaction probability of each component from one state to the other.

In our case, the information available in the SLAs for outsourced components concerns only minimal availability in a given time frame (e.g. 1 month). Therefore, classic techniques are not applicable to this problem, as the internal states of each component and the probability of state transition (i.e. failure and repair rate) are only known by the outsourcing company.

2.2.2. *System modelling*

Several approaches have been proposed in the literature for system reliability modelling. FTs and RBDs are the most used ones. However, we should mention that other approaches have also been

DOI: 10.1002/nem

proposed; e.g. Torres-Toledano and Sucar [13] use Bayesian networks, and Leangsuksun *et al*. [14] use a UML representation (although in this second case the authors do not provide the mathematical support for reliability analysis). In FTs, a number of components (called basic events) are linked together to make up a system according to AND/OR relationships. The same behaviour is achieved in RBDs through SERIES/PARALLEL compositions. According to Flammini *et al*. [3], FTs are easy to use, as they do not require very skilled modellers, and are relatively fast to evaluate, as it is possible to use very efficient combinatorial solving techniques to obtain most of the reliability indexes.

In FTs, the system state is represented by the top event, i.e. the root of the tree. It is possible to build a Boolean equation from the FT, and to reduce it to the *minimal cut set*, i.e. the smallest set of combinations of basic events (component failures) which all need to occur for the top event to take place (system failure) [15]. Based on the minimal cut set, a combination of combinatorial techniques and CTMC or Petri nets is then used to calculate the system (limiting) availability.

According to Flamini *et al*. [3], the main limitation of FTs and RBDs consists in the lack of modelling power, as they do not allow modelling maintenance-related issues explicitly. To solve this problem, FTs and RBDs have been extended into dynamic FTs [16] and dynamic RBDs [17], allowing one to model maintenance-related issues.

The modelling notation we use in this paper (AND/OR dependency graphs) can be seen as a condensed form of FTs. With a single AND/OR dependency graph we are able to model a forest of FTs sharing (some of) the basic events (i.e. the failure of a component), but with different top events. A single AND/OR dependency graph can thus model separately the failure of all the business services which the IT system provides, and for which a specific availability level must be calculated. In fact, it is possible to (automatically) transform any AND/OR dependency graph into a forest of FTs, as well as in a set of RBD, as we show in Appendix B. We share with FTs the use of minimal cut sets, which in our notation are called dependency sets (see Section 3), but the availability calculation we apply to AND/OR dependency graphs is different from the one used in FTs (for the reason mentioned above). In Appendix B we will give a more detailed explanation of the differences between FTs and the AND/OR dependency graphs we use in A^{2}THOS.

2.2.3. Tools

IBM Tivoli [18] and HP Business Availability Center [19] are two of the most popular configuration management tools. These tools are meant to support IT managers in the configuration and maintenance of complex IT systems. Among the many features they possess, they can be used to manage SLAs, including availability levels. One can assign to each IT component the availability level imposed by SLAs, and keep track of the actual availability levels to check for SLA compliancy. However, to the best of our knowledge there is no support for the analytical calculation of the service availability.

Galileo [20], Coral [21], Relex [22] and BlockSim [23] are tools operating with dynamic FTs. Although integrating the A^{2}THOS engines in one of these tools would be useful, this was not possible: Relex and BlockSim are commercial tools, Coral is mostly a MatLab library without a GUI, and Galileo is free software, but not open source. For these reasons we developed our prototype as an independent Java/Prolog tool.

2.2.4. Availability in service composition

In the field of IT service composition, several approaches have been proposed that consider availability as one of the quality of service (QoS) parameters to optimise the performance of the resulting composite IT service. Gu *et al*. [24] propose QUEST, a framework to schedule dynamically a composite IT service while satisfying QoS requirements (e.g. response time and availability) imposed by SLAs. Zeng *et al*. [25], Yu and Lin [26] and Ardagna and Pernici [27] propose scheduling techniques to create a cost-optimal execution plan for composite web services which respect QoS parameters (including availability) defined in SLA contracts.

In all these studies, an estimation of the availability of the composite service is made by multiplying the availability level of the components (expressed as a real number in the interval [0,1]). This is possible thanks to two simplifying assumptions. First, all the components must be available at the same time for the system to operate (i.e. the system is an AND combination of its components and it becomes unavailable at the moment any of its components is unavailable). Secondly, the resulting availability is not a lower bound, i.e. there can be a run of the composite service in which the resulting availability is lower than the calculated one. Different from these approaches, A^{2}THOS is able to deal with a wider range of dependencies, namely combinations of AND and OR dependencies. In the sequel we also argue in more detail why OR dependencies are necessary to model complex IT services correctly. A^{2}THOS also allows one to calculate an absolute lower bound for the availability, which can be safely included in an SLA contract.

3. ANALYSIS OF THE MINIMAL SERVICE AVAILABILITY

We now present the theoretical foundations of A^{2}THOS. Let us first start with an intuitive explanation. We model the system using an AND/OR dependency graph, in which a node represents a component of the system that at any given time may (or may not) be available. A directed edge from node m to node n indicates that m *depends on* n, i.e. that the availability of m depends also on the availability of n in a way that we are about to explain.

In an AND/OR dependency graph, a node m can be unavailable because of an internal failure, or because (some) nodes it depends on are unavailable. To model internal failure, to each node m we associate a (virtual) *internal node* m'. We say that the internal node m' is unavailable if the node is unavailable because of an internal failure. Therefore, internal nodes are just a notation artefact with no other fundamental purpose than indicating the internal failure of a node. An availability level set to an outsourced component is associated to the internal node representing the component, while the availability level of a node (also with regard to nodes it depends on) is associated to the node itself.

To model the fact that m becomes unavailable because one or more nodes it depends on are unavailable, we then consider nodes of two types: AND and OR (see Figure 2).

If m is a node in an AND/OR dependency graph and $n_1, \ldots, n_k$ are the nodes m depends on, we say that

- m is *unavailable* at time t iff its internal node m' is unavailable at time t *or*
  - at least one node in $n_1, \ldots, n_k$ is unavailable at time t, in case m is an AND node;
  - $n_1, \ldots, n_k$ are *all* unavailable at time t, in case m is an OR node.

Formally:

Definition 1 (AND/OR dependency graph). An AND/OR dependency graph $\langle N, E\rangle$ is a directed and acyclic graph (DAG) where N is the set of nodes, and is partitioned in AND $-N$ and OR $-N$, and E is the set of edges $E \subseteq \{\langle u, v\rangle | u, v \in N\}$.

Each node $n \in N$ thus has a virtual internal node n' and, given a graph $\langle N, E\rangle$, we call N' the set of internal nodes of g; $N' = \{n'$ internal of $n | n \in N\}$.

Running example: Part 1. *In this example we analyse the availability of an IT system providing two IT services (*`Service1` *and* `Service2`*), and implemented by means of three applications (*`App1`, `App2` *and* `App3`*) running on fivedifferent servers (*`Srv1`, `Srv2`, `Srv3`, `Srv4`, `Srv5`*).* `Service1` *is implemented by* `App1` *and* `App2` *in such a way that the service goes offline only when both applications are offline (OR dependency).* `Service2` *is implemented by* `App3`*, and* `App3` *depends*

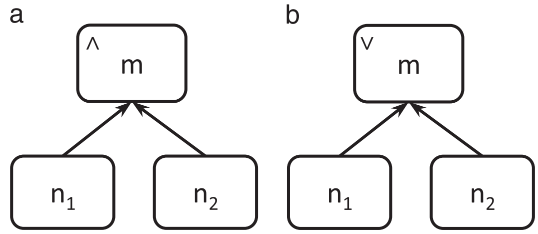

Figure 2. Two simple AND/OR dependency graphs, respectively with (a) AND and (b) OR nodes. Each of these nodes can also be indicated as internal nodes (i.e. m' is the internal node of m)

DOI: 10.1002/nem

on `App2` to work properly. `App1` is a distributed application running on `Srv1`, `Srv2` and `Srv3` in such a way that it can operate only if both `Srv1` and either `Srv2` or `Srv3` are online. `App2` runs on `Srv3`, and `App3` runs in both `Srv4` and `Srv5` with a load-balancing mechanism, such that it can continue to operate even if one of them is offline. Finally, the system is protected by the firewall `FW1`. According to this description, we build the AND/OR *dependency graph* $g = \langle N, E \rangle$ *as follows:*
AND-N = {`Service1`, `Service2`, `FW1` `App1`, `App2`, `App3`, `Srv1`, `Srv2`, `Srv3`, `Srv4`, `Srv5`}, OR-N = { `OR1`, `OR2`, `OR3` }, and E = { <`Srv1`, `App1`>, <`Srv2`, `OR2`>, <`Srv3`, `OR2`>, <`Srv3`, `App2`>, <`Srv4`, `OR3`>, <`Srv5`, `OR3`>, <`OR2`, `App1`>, <`OR3`, `App3`>, <`App1`, `OR1`>, <`App2`, `OR1`>, <`App2`, `App3`>, <`App3`, `Service2`>, <`FW1`, `Service1`>, <`FW1`, `Service2`>, <`OR1`, `Service1`> }.

To model the OR dependencies correctly we added three virtual nodes to act as logical gates: `OR1`, `OR2` and `OR3`. These nodes do not correspond to any existing component of the system, and therefore they cannot fail by themselves. Similarly, also the two nodes representing services (`Service1` and `Service2`) correspond to system functionalities which cannot fail by themselves. Figure 3 shows the AND/OR dependency graph of our running example.

In classic reliability theory, the internal state of a node is described by a random variable which assumes value 1, corresponding to the functioning state, and 0, corresponding to the failed state. Accordingly, the state of a system made of multiple components is described by a vector of random variables, each describing the state of one component. It is also possible to describe the state of a (sub) system with values 0 or 1 as a function of the states of the (sub)system components.

We represent these concepts by means of the *state function* χ. Given a node m, $\chi(m', t)$ is 0 iff m at time t suffers an internal failure (represented by the virtual internal node m'), and 1 otherwise. Similarly, $\chi(m, t) = 0$ indicates that node m is unavailable at time t (either because of an internal failure or because a node m depends on has failed). As explained above, the state function of a node m is a function of the state of its internal node and the state functions of the nodes it depends on. This is formalised in the next definition.

Definition 2 (State function). Let $g = \langle N, E \rangle$ be an AND/OR dependency graph and N' be the set of (virtual) internal nodes corresponding to N. We say that χ is a *state function* for g iff $\chi : \left(N \cup N'\right) \times \mathbb{R}^+ \to \{0, 1\}$, and for each $m \in N$ and $t \in R^+$ the following holds: let $n_1, \ldots, n_k$ be the nodes in N m depends on. Then

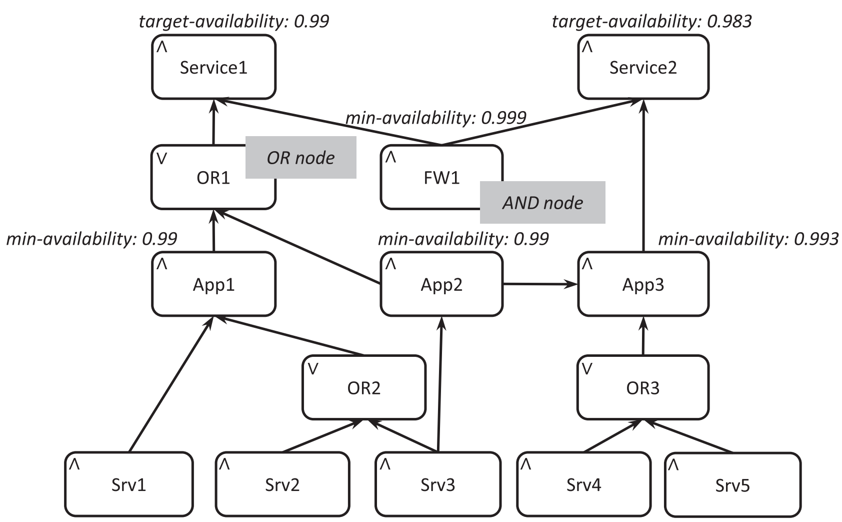

Figure 3. The AND/OR dependency graph representing the system we analyse in our running example. AND nodes are represented by the ∧ symbol and OR nodes by the ∨ symbol

DOI: 10.1002/nem

$$\chi(m,t) = \begin{cases} \chi(m',t)\cdot\chi(n_1,t),\ldots,\chi(n_k,t), & \text{if } m \text{ is an AND node} \\ \chi(m',t)\cdot\max(\chi(n_1,t),\ldots,\chi(n_k,t)), & \text{if } m \text{ is an OR node} \end{cases} \tag{1}$$

Using this function, we can represent the part of a time interval $[t_0, t_1]$ in which a given node is available as the infinite set $\{t \in [t_0, t_1] | \chi(m', t) = 1\}$.

Therefore, given the state function of all the internal nodes in an AND/OR dependency graph, one can iteratively compute the state function of all the nodes in the graph (here the fact that the graph is acyclic guarantees that the above function iswell defined). The only condition we pose on the χ function is that it should be Lebesgue measurable, which guarantees that given two real numbers r and s we have that $\int_r^s \chi(m,t)\mathrm{d}t$ exists. It should also be noted that $\chi(m, t)$ is a random variable, as a function of random variables.

According to the dependability theory [11], the *interval availability* of a node m is the fraction of a given interval of time that m operates within tolerances. Supposing the given interval of time is $[t_0, t_1]$, the formula of interval availability is given by

$$\bar{A}(m, t_0, t_1) = \frac{1}{t_1 - t_0}\int_{t_0}^{t_1} \chi(m,t)\mathrm{d}t \tag{2}$$

The *limiting interval availability*, or steady-state availability, is the expected fraction of time in the long run that the system operates within tolerances ($lim_{t\to\infty} \bar{A}(m, t_0, t)$).

The formulas for availability we just described, however, are too general to perform numerical calculations. $\chi(m, t)$, being a random variable, will be governed by a distribution function. Therefore, in reliability theory, it is common practice to assume a stationary alternating renewal process, i.e. a system with independent, exponential distributions of failure and repair times. Under this assumption, the formula for the limiting interval availability is given by $\frac{\mu}{\lambda + \mu}$, where λ is the failure rate and μ is the repair rate. λ and μ can be estimated for a real system based on the MTBF and MTTR, and this allows one to compute the limiting interval availability for a real system (under the given assumptions).

In SLAs, the agreed minimal availability is always indicated as fraction of uptime *in a given time frame* (e.g. 0.98 uptime per month). Note that the presence of the time frame is crucial: for instance, guaranteeing 0.99 uptime per month is more difficult than guaranteeing 0.99 uptime per year. In the first case the system may not be offline for more than 7.2 hours in a row, while in the second case the system may be offline for up to 87.6 hours in a row. Equation (2) can be seen as a formalisation of the availability parameter used in SLAs. Therefore, to be able to calculate the system interval availability, the component interval availability must have been expressed in the same *time interval* (e.g. 1 month). Since the minimal availability for outsourced components is given in SLAs, under the assumption of a state function as defined in (1), we do not need to set any stochastic assumption of the form of χ, as the quantity:

$$\bar{A}(m', t_0, t_1) = \frac{1}{t_1 - t_0}\int_{t_0}^{t_1} \chi(m',t)\mathrm{d}t$$

is a given input and never needs to be calculated.

The availability of a component is also given under the assumption that any other component it depends on is always available. For example, for server management, the SLA ensures a given availability level, provided that the data centre in which the server is deployed and the network to which the server is connected are operating within tolerances.

Now, the technical question we are going to address in the rest of this section, the answer of which will form the basis of our approach, is the following.

Let us fix a reference time interval $[t_0, t_1]$ and suppose that we know a lower bound for the availability of the internal nodes of the nodes in a graph, i.e. for each $n' \in N'$, we know an $\alpha_{n'}$ such that the state function χ satisfies the following equation:

$$(\mathrm{av}(n') =) \frac{1}{t_1 - t_0}\int_{t_0}^{t_1} \chi(n',t)\mathrm{d}t \geq \alpha_{n'} \tag{3}$$

where $\mathrm{av}(n')$ is the fraction of the time interval $[t_0, t_1]$ in which n' was operating within tolerance. Given an arbitrary node $m \in N$, what can we say about $\mathrm{av}(m)$ in the same time period? In particular, can we compute a lower bound for it?

DOI: 10.1002/nem

Note the importance for the IT manager of considering a lower bound of the availability of a node (i.e. an IT service). The availability of the node depends on the availability of nodes it depends on, and only a minimum guaranteed figure is given for them. Based on assumption (4) of the principal agent model (see Section 2), the agent (i.e. the service provider) shows opportunistic behaviour to maximise its own profit, regardless of the goals of the principal. Under this assumption, the principal (IT manager) cannot expect that the unavailability of the outsourced nodes will be distributed in a way that guarantees his availability-related goals with respect to the business units of the organisation. For example, if the service provider has an economic advantage in planning the maintenance of outsourced components in such a way that the IT service of a customer will violate one of the organisation (internal) SLAs it will do that, provided that it is not violating oneof the SLAs agreed with the customer. Therefore, the IT manager will be interested in considering the 'worst case' scenario when planning the availability of outsourced components.

3.1. Dependency sets

To answer the question above, we have to introduce the concept of dependency set. The dependency set of a node m is the set of the smallest sets of internal nodes in the AND/OR dependency graph which, if all unavailable at the same time, will cause the failure of m. The elements of a dependency set have the same property as the minimal cut sets of a FT, and can be obtained similarly by representing the graph as a Boolean equation and using substitution methods to reduce the equation. We will now present a more formal definition of dependency sets.

Definition 3. Consider an AND/OR dependency graph $g=\langle N,E\rangle$ and a node $m\in N$. The *dependency set* of m, $\mathrm{DEPS}_m\subseteq\wp\left(N'\right)$, is defined inductively as follows:

- If m is a leaf node, then $\mathrm{DEPS}_m=\{\{m'\}\}$.
- If m has children $n_1..n_k$; let $\mathrm{DEPS}_{n_1},\ldots,\mathrm{DEPS}_{n_k}$ be the dependency set of $n_1,\ldots,n_k$ and assume (without losing generality) that for every i, $\mathrm{DEPS}_{n_i}=\left\{D_{i,1},\ldots,D_{i,l_i}\right\}$, then:
  - if $m\in\mathrm{AND}-N$ then
$$\mathrm{DEPS}_m=\{\{m'\}\}\cup\cup_{i\in[1..k]}\mathrm{DEPS}_{n_i};$$
  - if $m\in\mathrm{OR}-N$ then
$$\mathrm{DEPS}_m=\{\{m'\}\}\cup\left\{D_{1,j_1}\cup\ldots\cup D_{k,j_k}\middle|D_{i,j_i}\in\mathrm{DEPS}_{n_i}\right\}.$$

Running example: Part 2. *By applying Definition 3 to our example* AND/OR *dependency graph, we obtain the following dependency sets for* `Service1` *and* `Service2`*:* $\mathrm{DEPS}_{Service1}$*={{FW1′}, {App1′, App2′}, {App1, Srv3′}, {Srv1′, App2′}, {Srv1′, Srv3′}, {App2′, Srv2′, Srv3′}, {Srv2′, Srv3′}.* $\mathrm{DEPS}_{Service2}$*={{FW1′}, {App2′}, {App3′}, {Srv3′}, {Srv4′, Srv5′}}. For the sake of presentation we did not include in the dependency sets the internal nodes that cannot fail by themselves (i.e. Service1′, Service2′, OR1′, OR2′ and OR3′). It is easy to see that when the nodes of any of the elements of* $\mathrm{DEPS}_{Service1}$ *are unavailable at the same time,* `Service1` *is unavailable, and the same for* $\mathrm{DEPS}_{Service2}$.

The dependency set of a node is always a set of sets of internal nodes, so without loss of generality we can always write $\mathrm{DEPS}_m=\{D_1,\ldots,D_k\}$.

As for minimal cut sets in FTs, a relevant property of DEPS_m is that, if the internal m' of m is available at a given time t, then m is not available only if there exists DEPS_m such that at least all the internals of the nodes contained in one element D of DEPS_m are all unavailable. More formally, if we fix a time t_0, then

$$\chi(m,t_0)=0\Leftrightarrow\exists D\in\mathrm{DEPS}_m,\forall d\in D,\chi(d,t_0)=0 \quad (4)$$

For the sake of presentation we skip the (straightforward) demonstration of this property.

As an example let us consider the two toy cases described in Figure 2. In case (a), $\mathrm{DEPS}_m=\{\{m'\},\{n'_1\},\{n'_2\}\}$; so if $\chi(m,t)=0$ and $\chi(m',t)=1$, then either $\chi(n'_1,t)=0$ or $\chi(n'_2,t)=0$.

DOI: 10.1002/nem

In case (b), $\text{DEPS}_m = \{\{m'\}, \{n'_1, n'_2\}\}$; so if $\chi(m,t)=0$ and $\chi(m',t)=1$, then both $\chi(n'_1,t)=0$ and $\chi(n'_2,t)=0$.

The following theorem states that if we know a lower bound for the availability of the internal nodes of an AND/OR dependency graph then we can effectively compute an optimal lower bound of the availability of each node $m \in N$ in the graph. In the theorem we will also explain the meaning of an optimal availability lower bound.

In relation to the concrete problem we are discussing in this paper, this theorem states that there is a way for the manager to check that a partially outsourced IT service will comply with the internal SLAs, even in the case that the outsourced components will be unavailable in the worst possible combination (but still within their respective SLAs).

Theorem 1: Let $g = \langle N, E \rangle$ be an AND/OR dependency graph, $[t_0, t_1]$ be a time interval, and for each $n' \in N'$ let $\alpha_{n'}$ be a real value $\alpha_{n'} \in [0,1]$. Then, for each $m \in N$ we can compute α_m optimal, such that for each state function χ for g the following holds:

$$\text{if } \forall n' \in N' \frac{1}{t_1 - t_0} \int_{t_0}^{t_1} \chi(n', t)\mathrm{d}t \geq \alpha_{n'} \tag{5}$$

then

$$\frac{1}{t_1 - t_0} \int_{t_0}^{t_1} x(m, t)\mathrm{d}t \geq \alpha_m \tag{6}$$

We say that α_m is optimal if we can find a χ for g such that (5) holds and in (6) equality holds.

We provide the proof for this theorem in Appendix A. As a result of the proof, we obtain a method to calculate the lower bound α_m of the availability of any node m in the graph. The method consists in solving the linear programming problem of (7).

Intuitively, the availability of m is minimal when m is sequentially disrupted by the simultaneous failure of all the internal nodes of each element of DEPS_m. Without loss in generality, we can write $\text{DEPS}_m = \{D_1, \ldots, D_k\}$, and for each D_i we can write $D_i = \{d_{i,i}, \ldots, d_{i,l_i}\}$ and call u_i the unavailability caused to m by D_i. According to this notation, the availability of m is given by 1 less the sum of the unavailability caused by each D_i (represented by u_i). Therefore, to minimise the availability of m we write the following optimisation problem:

$$\alpha_m = \begin{cases} \text{minimize } 1 - u_1 - \cdots - u_k \\ \text{subject to} \\ u_1 = (1 - a_{1,1}) = \cdots = (1 - a_{1,l_1}) \\ \vdots \\ u_k = (1 - a_{k,1}) = \cdots = (1 - a_{k,l_k}) \\ \forall n \in N, \sum_{d_{i,j} \in D_n} 1 - a_{i,j} \geq 1 - \alpha_{n'} \\ a_{1,1}, \ldots a_{1,l_1}, \ldots, a_{k,1}, \ldots, a_{k,l_k} \geq 0 \end{cases} \tag{7}$$

The objective function of (7) represents the availability of node m, which needs to be minimised.

The first k conditions impose that the internal nodes of each element $D_i \in \text{DEPS}_n$ are unavailable at the same time: this ensures m is disrupted because of the simultaneous failure of all the internal nodes in D_i. To this end, we call $a_{i,j}$ the availability of the element $d_{i,j} \in D_i$ and we impose that, given a D_i, all $a_{i,j}$ are equal.

The subsequent condition imposes for each node $n \in N$ that the availability of its internal node n' is not less than $\alpha_{n'}$: in this way we ensure that, even if an internal node n' is contained in more than one element of DEPS_n, the unavailability caused by its failure will not exceed its upper bound $(1-\alpha_{n'})$. In fact, given two elements $D_1, D_2 \in \text{DEPS}_m$, these two elements might not be pairwise disjoint (i.e. $D_1 \cap D_2 \neq \emptyset$) because some elements in D_1 and D_2 refer to the same node. In this condition, we call D_n the set of elements $d_{i,j}$ which all refer to the same node n.

The last condition ensures no negative value can be used to represent availability. A solution to (7) can be found by using the simplex algorithm.

From now on, we call α_m determined from (7) the *minimal aggregated availably level* of m.

DOI: 10.1002/nem

Running example: Part 3. *According to the dependency sets we previously determined and to (7), the linear programming problem which determines* α_{Service1} *is*

$$\begin{cases} \text{minimize } 1-u_1-u_2-u_3-u_4-u_5-u_6-u_7 \text{ subject to} \\ u_1=(1-a_{\text{FW1,1}}) \\ u_2=(1-a_{\text{App1,2}})=(1-a_{\text{App2,2}}) \\ u_3=(1-a_{\text{App1,3}})=(1-a_{\text{Srv3,3}}) \\ u_4=(1-a_{\text{Srv1,4}})=(1-a_{\text{App2,4}}) \\ u_5=(1-a_{\text{Srv1,5}})=(1-a_{\text{Srv3,5}}) \\ u_6=(1-a_{\text{App2,6}})=(1-a_{\text{Srv2,6}})=(1-a_{\text{Srv3,6}}) \\ u_7=(1-a_{\text{Srv2,7}})=(1-a_{\text{Srv3,7}}) \\ 1-a_{\text{FW1,1}}\geq 1-\alpha_{\text{FW1}'}=0.001 \\ (1-a_{\text{App1,2}})+(1-a_{\text{App1,3}})\geq 1-\alpha_{\text{App1}'}=0.01 \\ (1-a_{\text{App2,2}})+(1-a_{\text{App2,4}})+(1-a_{\text{App2,6}})\geq 1-\alpha_{\text{App2}'}=0.005 \\ (1-a_{\text{Srv1,4}})+(1-a_{\text{Srv1,5}})\geq 1-\alpha\text{Srv1}'=0.001 \\ (1-a_{\text{Srv2,6}})+(1-a_{\text{Srv2,7}})\geq 1-\alpha_{\text{Srv2}'}=0.001 \\ (1-a_{\text{Srv3,3}})+(1-a_{\text{Srv3,5}})+(1-a_{\text{Srv3,6}})+(1-a_{\text{Srv3,7}})\geq 1-\alpha_{\text{Srv3}'}=0.01 \\ a_{\text{FW1,1}},a_{\text{App1,2}},a_{\text{App1,3}},a_{\text{App2,2}},a_{\text{App2,4}},a_{\text{App2,6}},a_{\text{Srv1,4}}, \\ a_{\text{Srv1,5}},a_{\text{Srv2,6}},a_{\text{Srv2,7}},a_{\text{Srv3,3}},a_{\text{Srv3,5}},a_{\text{Srv3,6}},a_{\text{Srv3,7}}\geq 0 \end{cases}$$

which gives a lower bound for the availability of `Service1` *of 0.984. Similarly, we can determine* $\alpha_{\text{Service2}}=0.972$. *From the IT manager's point of view, the results of this example suggests that there could be a certain scheduling of the unavailability of the outsourced components* `Service1` *depends on (in a given time frame), which would result in an availability level of 98.4% for* `Service1`, *without the outsourcer having violated any of the availability SLAs agreed with the IT manager for the outsourced components. Figure 4 shows one possible scheduling for the unavailability of the components on which* `Service1` *depends on, resulting in* `Service1` *having an availability of* α_{Service1} *(0.984).*

4. OPTIMISATION OF OUTSOURCED SERVICES

In the outsourcing scenario that we are considering in this paper, we have seen that in order to fully benefit from this kind of sourcing contract the customer needs to select the quality level of the different available attributes (including availability) for the different building blocks that the

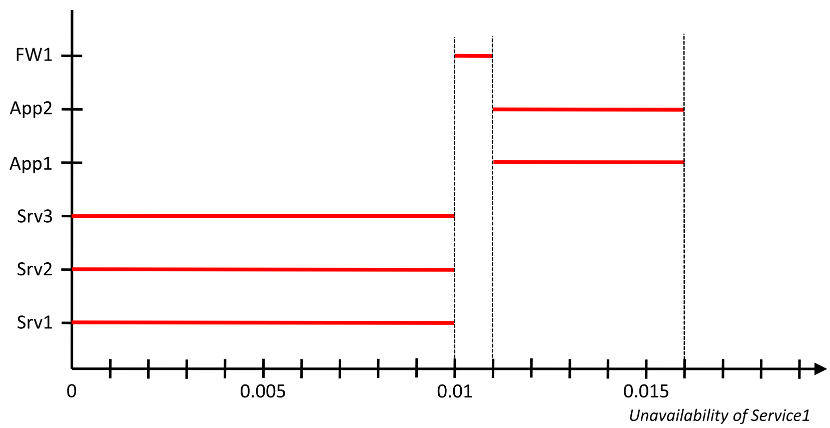

Figure 4. One possible scheduling for the failure of `FW1`, `App1`, `App2`, `Srv1`, `Srv2` and `Srv3` resulting in `Service1` having an availability of 0.984. System components are on the vertical axis and the components unavailability fraction of time ($\in [0, 1]$) is on the horizontal axis

contract provides in such a way that the delivered service has the expected quality and that the cost is minimal.

In the previous section we have seen that determining the minimal availability level of a complex system is a non-trivial problem, which can be solved by reducing it to an optimisation problem. A relevant application of this result is the minimisation of the cost for outsourced subcomponents. Given that outsourcing has a cost that may (also) depend on the minimal availability guaranteed for the outsourced component, a manager needs to minimise the outsourcing cost while guaranteeing that the services provided by the system meet—among other non-functional requirements—the target availability.

The situation is the reverse from the one in the previous section: instead of *calculating* the service minimal availability *given* the minimal availability of the various system components, one wishes to calculate what is the least expensive combination of components given the target minimal availability of the services. Thus availability level optimisation consists in determining the assignment of an availability level to the components of the system for which it is possible to chooseamong different availability levels, so that

1. a *minimal aggregated availability level* is ensured for the services provided by the system;
2. the cost of the assignment is minimal.

To this end we distinguish among three types of nodes in an AND/OR dependency graph: *target* availability nodes, *variable* availability nodes and *given* availability nodes. More formally, given an AND/OR dependency graph $\langle N, E\rangle$, $N = N_T \cup N_V \cup N_G$, where N_T, N_V and N_G are the pairwise disjoint sets of target, variable and given availability nodes.

Target availability nodes are the nodes modelling the services provided by the system. The target expresses the minimal availability level which the system is or should be able to guarantee regarding a given functionality (service). Typically, we define a target availability level on the service nodes of the AND/OR dependency graph, whenever there is an SLA (be it company internal or not) which imposes a certain level of availability for them.

Definition 4 (Target availability level). Given an AND/OR dependency graph $\langle N, E\rangle$, and $N_T \subset N$ the set of target availability nodes, the target availability level of a node is a mapping *target availability* $: N_T \to [0, 1]$.

Running example: Part 4. *Our example system provides two main functionalities, described in the AND/OR dependency graph by the* `Service1` *and* `Service2` *nodes. The functionality described by* `Service1` *is more mission-critical than the one described by Service2, and an SLA set on the system ensures a minimal availability level of 0.99 for* `Service1` *and of 0.983 for* `Service2`*. Accordingly,* target availability*(Service1)* $= 0.99$ *and* target availability*(Service2)* $= 0.983$.

Variable availability nodes model the situation in which it is possible to choose the availability level of a component among different options (e.g. gold, silver and bronze) with different associated price. Variable availability nodesare the components outsourced under the contract scenario assumed in this paper.

We model the domain of a variable availability level by means of a set of *availability options*.

Definition 5 (Availability option). Let $\langle N, E\rangle$ be an AND/OR dependency graph, N_V a set of variable availability nodes and $N_V^{'}$ the set of the internals of the nodes in N_V. An *availability option* for N_V is a function aa : $N_V^{'} \to [0, 1]$. We call O the set of all the availability options.

According to the assumed scenario, IT managers can select the value of a number of quality attributes from a list defined in the contract. Examples of such quality parameters include incident response time, availability of the support (e.g. 24/7 or 8/5), data backup rates and availability.

The options available to an IT manager that has to choose the value of quality parameters for a certain building block are in principle as big as the combinations of all possible parameter values (e.g. an option could be choosing an UNIX server with incident response time within 4 hours, support available 24/7, daily data backup and 0.995 availability). With *availability options*, we are only considering the combination of availability levels for components. To enumerate the availability options for a single component, the IT manager must first select the other quality parameters. We believe that other optimisation techniques may be used for the selection of the other quality parameters, similarly to what we are about to present in this section.

The outsourcing contract includes a price catalogue associated with each building block. The price catalogue contains the price associated with different quality parameter values. We assume that the cost (for the customer) associated with a certain availability option can be extracted from the price catalogue by summing the price of all the selected quality parameters (including the price of the availability level). Owing to economies of scale, the outsourcing company may also apply graduated unit prices if the number of purchased items exceeds a certain value. The cost of availability options should take this into account. (For those who are familiar with optimisation problems: this introduces a form of non-monotonicity in the outsourcing offers, where outsourcing more services could be potentially less expensive than outsourcing a smaller number of services. This non-monotonic aspect of the problem makes it more difficult to find the optimal solution.)

Based on these considerations, we define the cost of an availability option as a function that maps the availability option to the domain of real numbers (i.e. money). The definition is kept general to allow the implementation of (more) complex pricing schemes.

Definition 6 (Availability option cost). Let $\langle N, E\rangle$ be an AND/OR dependency graph, N_V a set of variable availability nodes, $N_V^{'}$ the set of the internals of the nodes in N_V and O the set of possible availability options for N_V. The cost of the availability option is a function $cost : O \rightarrow \mathbb{R}^+$.

Running example: Part 5. *The contract between the outsourcing company and the customer defines different availability level options for the management of Windows servers and UNIX servers. Table 1 summarises the associated price catalogue. In our example* `Srv1`, `Srv2` *and* `Srv3` *are Windows servers with variable availability, and both* `Srv4` *and* `Srv5` *are UNIX servers with variable availability. The set of variable availability nodes is* $N_V = \{Srv1, Srv2, Srv3, Srv4, Srv5\}$*. According to the price catalogue, there are* `number of availability levels` *at the power of* `number of variable availability nodes` ($3^5 = 243$) *possible combinations for the minimal availability level of the elements in* $N_V^{'}$, i.e. $|O| = 243$. *One of these combinations is* o_1, *where* $o_1(Srv1') = o_1(Srv2') = o_1(Srv3') = o_1(Srv4') = o_1(Srv5') = 0.99$. *The cost function is extracted from Table 1 (in this simplified example, availability is the only quality parameter to choose), and* $\text{cost}(o_1) = (3 \cdot 1000 + 2 \cdot 900) = 4800$.

To guarantee that condition 1 of our problem is met, we extend the definition of minimal aggregated availability to be applied also to nodes with a variable availability. According to this, given a node $t \in N_T$ with target availability, we call *minimal availability*(t, o) the minimal aggregated availability of t when for all nodes n with variable availability $\alpha_{n'}$ is determined by o.

Finally, a node with *given* availability models components for which the minimal availability is known and not variable.

Definition 7 (Given availability level). Given an AND/OR dependency graph $\langle N, E\rangle$, and $N_G \subset N$ the set of target availability nodes, the given availability level of a node is a mapping *given availability*: $N_T \rightarrow [0, 1]$.

Running example: Part 6. *In our example the components whose minimal availability is given are* `FW1`, `App1`, `App2` *and* `App3`*. Therefore, according to Figure 3 we have that* given availability

Table 1. An example of availability level options price catalogue

Availability level	Minimal quantity	Windows price	UNIX price
0.99	1 server	1000 euro	900 euro
0.99	6 servers	900 euro	800 euro
0.995	1 server	1300 euro	1200 euro
0.995	11 servers	1200 euro	1100 euro
0.998	1 server	1500 euro	1400 euro
0.998	6 servers	1400 euro	1300 euro

DOI: 10.1002/nem

(FW1) = 0.999, given availability*(App1)* = 0.99, given availability*(App2)* = 0.99 *and* given availability*(App3)* = 0.993.

At the beginning of this section we have defined our problem as determining the assignment of an availability level to each outsourced component of the system so that the services offered by the system meet a certain target availability while minimising the cost. Considering the three types of node that we have just defined, we can reformulate the problem as: given an IT system made of *target availability* nodes, *given availability* nodes and *variable availability* nodes and certain availability levels required from the services offered by the system, choose an availability option for *variable availability* nodes such that the resulting availability of *target availability* nodes matches the required level and the outsourcing cost is minimum.

More formally, let $\langle N, E\rangle$ be an AND/OR dependency graph, $N = N_T \dot{\cup} N_V \dot{\cup} N_G$, a function *target availability* on $N_T^{'}$, a function *given availability* on $N_G^{'}$, a set O of availability options for N_V and a function *minimal availability* for $N_T \times O$. Find the option $o \in O$ with minimal cost such that $\forall t \in N_T$, *minimal availability*$(t, o) \geq$ *target availability*(t).

Based on this formal description we formulate in (8) a linear programming problem with variables in a finite domain. The solution to this problem is the availability option that minimises the outsourcing cost.

$$\begin{cases} \text{minimize } cost(o) \\ \text{subject to:} \\ o \in O \\ minimal\ \ availability(t_1, o) \geq target\ \ availability(t_1) \\ \vdots \\ minimal\ \ availability(t_P, o) \geq target\ \ availability(t_P) \end{cases} \tag{8}$$

Running example: Part 7. *Recall that the nodes with variable availability are* `Srv1`*,* `Srv2`*,* `Srv3`*,* `Srv4` *and* `Srv5`*. The set of availability options O is made of 243 elements. We want to ensure that the lower bound of the monthly availability is 0.99 for* `Service1` *and 0.983 for* `Service2`*. Consequently, the optimisation problem is as follows:*

$$\begin{cases} \text{minimize } cost(o) \\ \text{subject to:} \\ o \in O \\ minimal\ \ availability(\text{Service1}, o) \geq 0.99 \\ minimal\ \ availability(\text{Service2}, o) \geq 0.983 \end{cases}$$

which gives us an (optimal) solution with cost 6300 euro *when* $\alpha'_{Srv1} = 0.998$, $\alpha_{Srv2} = 0.99$, $\alpha_{Srv3} = 0.998$, $\alpha_{Srv4} = 0.99$ *and* $\alpha_{Srv5} = 0.998$.

5. IMPLEMENTATION AND BENCHMARKS

5.1. Implementation

We have implemented a prototype of A^{2}THOS to run our lab experiments and to support case studies. The prototype is written in Java and prolog in about 10 000 lines of code. We chose to use the ECLiPSe [28] prolog platform since it provides a flexible yet powerful set of constraint solvers which we need to deal with the linear programming problems of A^{2}THOS. The available solvers include *fd*, a solver for finite domain integer problems, *ic*, a solver for hybrid integer/real-interval problems and *eplex*, an interface to an (external) simplex solver library.

Figure 5 shows the software architecture of our prototype. It consists of four interacting components: the GUI front-end, the driver, the analysis and the optimisation engines. The GUI front-end manages the interaction with the final user. It is implemented as a standalone Java application and it allows the user to quickly create the AND/OR dependency graph by dragging and dropping nodes and edges, to annotate each node with its availability figure(s) or availability level options and to viewthe analysis and optimisation results. The analysis engine solves the availability analysis

DOI: 10.1002/nem

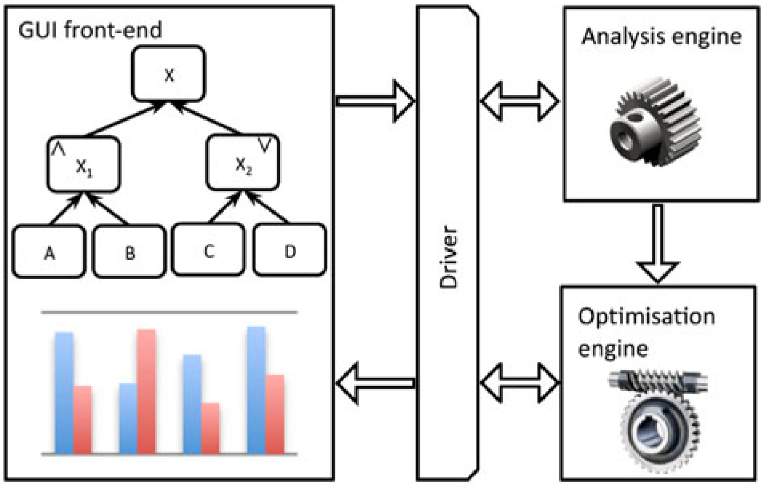

Figure 5. A^{2}THOS architecture

problem, described in Section 3. It is implemented in prolog by using the *eplex* (simplex algorithm) solver of the ECLiPSe platform. The optimisation engine solves the availability optimisation problem, which we describe in Section 4. It is also implemented in prolog by using the *fd* solver of the ECLiPSe platform. Finally, the driver is written in Java and manages the interaction of the Java components with the prolog ones. It uses the JavaECLiPSe interface to build a prolog optimisation problem from the AND/OR dependency graph and the other availability-related information inserted by the user. It then translates the results given by the engines in a format that can be presented to the user by the Java GUI front-end.

5.2. *Benchmarks*

To be of practical use, our prototype needs to deliver a solution to the linear programming problems in a reasonable time. Unfortunately, the simplex algorithm has a worst-case exponential complexity [29], and solving by brute-force linear programming problems with variables in a finite domain has an exponential complexity in the number of variables and their domain size. This means that the implementation does not scale, and therefore we have to benchmark whether it can tackle the size of a real-world IT system. In the sequel we show that it does so; nevertheless we want to stress that our implementation is just a proof of concept and its speed can no doubt be improved: our goal is to demonstrate how this can be done, and not to provide a fast implementation.

We benchmarked the performance of our prototype by running it on inputs with growing size. We run our test on a machine with an Intel Pentium 4 CPU running at 3.6 GHz and with 2 GB RAM.

First, we benchmarked the availability analysis. Here, the complexity of the simplex algorithm is determined by the number of variables and constraints of the linear programming problem it solves. Therefore, we generate inputs for the analysis engine byincreasing the number of nodes and by adjusting the node types and edges to obtain a growing number of constraints (and associated variables) for the linear programming problem of (7). We set the maximum number of nodes for our tests to 250 and the maximum number of constraints to 600. In our experience these numbers correspond to a fairly large IT system. To increase randomness we also repeat several times (five) the test for a certain number of nodes and constraints, and we then calculate the average computation time. The results are shown in Table 2. Our tests indicate that given a fixed number of constraints the computational time is basically linear in the number of nodes, and that our prototype is able to handle an AND/OR dependency graph of 250 nodes and 600 constraints on average in less than a minute.

Secondly, we benchmarked the optimisation algorithm. Our prototype implementation works by exhaustively searching the space of all available options and choosing the best one. The algorithm is thus optimal (it finds the best solution, every time), but its complexity is exponential in the number of variables (which in this case corresponds to the number of nodes with variable availability). Again, the

Table 2. Performance of the simplex algorithm for availability analysis

Nodes	Constraints	Time (s)
15	10	0.00001
15	20	0.002
60	10	0.001
60	20	0.005
60	60	0.02
120	10	0.004
120	20	0.01
120	100	0.09
120	150	0.22
120	250	0.8
120	300	1.3
120	600	20.2
250	10	0.009
250	20	0.011
250	100	0.22
250	150	0.5
250	250	1.6
250	300	2.6
250	600	41.1

fact that the algorithm is exponential means that we cannot expect it to scale up indefinitely, and it is therefore important to assess via benchmarks how big a problem it is able to tackle.

We carried out these benchmarks as follows. We create a simple program which takes as input the desired number of nodes with availability level options, the average number and the average size of the dependency sets and generates a random AND/OR dependency graph with random availability level options which match the given parameters. We set three possible availability levels for nodes with variable availability, since that is the most common configuration in outsourcing scenarios (gold, silver and bronze). We then solve the problem with our optimisation engine and note the execution time. We use an increasing number of nodes with variable availability (up to 50) and we specify different average number and average sizes of the dependency sets. We repeat several times (five) the test for each configuration in order to increase randomness.

Our results indicate that the computational time is mostly influenced by the number of independent nodes. By independent node here we mean a node that appears in only one element of a dependency set. We report in Table 3 the results of our tests with 50 nodes. As the number of independent nodes increases, the computational time increases as well. We are able to solve a problem with 50 nodes, among which 25 are independent in $1\frac{1}{2}$ hours. However, the trend is—as expected—exponential, and with 30 independent nodes we exceed 6 hours of computation. This is due to the fact that the solver has to explore all the possible combinations of values for the variables associated to independent nodes, while the domain of the other variables is limited by the problem constraints.

Our benchmark indicates that the crucial factor influencing the computation time is the number of independent nodes (outsourced components) which contribute independently to the system availability (AND dependency), and that the algorithm as itis now is always able to handle situations with up to 25 such nodes. In practice, this number is sufficient to model a single medium/large IT system, as we will show in the next section. It is worth noting that one can break a huge IT system into independent subsystems and apply the algorithm to them one by one. In this light, 25 outsourced components represent a limit which is basically never exceeded (in our industrial test case, which was carried out at a multinational company, we had a maximum of six independent nodes).

In the unlikely case that one would need to apply the algorithm to too large a system (e.g. exceeding the 40 AND independent), one could still refer to the optimisation problem we have reported in Section 4, but then use a non-exhaustive algorithm to find a solution to it. Non-exhaustive algorithms (e.g. those based on *local search* [30]) have the disadvantage that they do not guarantee finding the optimal solution (they usually find a local optimum, which is not guaranteed to be a global optimum as well), but could probably easily scale to hundreds of independent nodes.

Table 3. Performance of the availability optimisation algorithm with 50 variable availability nodes

Independent nodes	Time (s)
10	≤ 0.01
15	0.01
20	197.20
25	5418.50
30	≥ 21600.00

6. METHODOLOGY: PRACTICAL USE OF A^{2}THOS

In this section we present a case study we carried out on the IT infrastructure of a large multinational company. With this case study we want to address three important questions regarding A^{2}THOS:

1. Does the approach scale up to conditions of practical utility?
2. Can A^{2}THOS be (economically) applied to a practical case; i.e. does it require information not available in practice?
3. How useful is the information provided by A^{2}THOS for its intended users?

Let us now present the context in which we carried out the case study. The multinational company (from now on we call it the Company) has a global presence in over 50 countries and counts between 100 000 and 200 000 employees. Our case study was conducted at the site of IT facilities for the Company's European branch.

The Company IT department supports the business of hundreds of other departments by offering thousands of applications accessed by approximately 100 000 employee workstations and by many hundreds of business partners. IT services are planned, designed,developed and managed by the IT infrastructure department located at the Company's headquarters. These services (e.g. e-mail or ERP systems) are part of the IT infrastructure which is used by all the different Company's branches all over Europe.

For efficiency reasons, as in most other large organisations, business units exchange services by means of an 'enterprise internal market'. One business unit pays another one for the use of a given service and the service provider unit finances its activities by means of these funds. Within this 'internal market', the quality of the provided services is regulated by means of SLAs. Among the other QoS parameters, SLAs include the *minimal ensured availability* of the offered services.

IT services are designed internally by the IT department and then partly outsourced for implementation and management to another company. We call this company the Outsourcer. The Outsourcer is a market-leading international IT services provider. The outsourced tasks include application and server management, help-desk and problem solving. Although the servers running the IT services are owned by the Company and physically kept within its data centres, the Outsourcer manages the OS and the software running on them. The Outsourcer has signed contracts with the Company which include SLAs regarding both the security of the information managed by the outsourcing company and the availability of the outsourced services.

The Company and the Outsourcer have established a global contract regulating the application and server management service provisioning. The contract is made of several *building blocks*, e.g. UNIX server management, Windows server management or Oracle database management. Every time the IT department of the Company needs to deploy a new IT service, a new request is issued to the Outsourcer to provide the building blocks needed by the service. The QoS parameters and their possible values foreach building block are also predefined in the contract. Regarding availability, for each building block the Company can choose among different guaranteed minimal availability levels. The price for the provisioning of each building block with specific QoS parameters is part of the price catalogue of the Outsourcer.

One of the problems the IT infrastructure managers of the Company have to deal with is how to determine the minimal availability level of new IT services. In fact, this availability level is meant to be used to set up the Company internal SLAs between the service provider (the IT department) and the service users (the other departments of the Company). It is important that the IT infrastructure manager is as precise as possible in determining the minimal availability level to be agreed with the internal users. In fact, too low a value could prevent agreement being reached, as the service users may not be willing to pay for a service which does not fit their needs in terms of availability. On the other hand, too high a value may impact the budget of the IT department, as for each time that the SLA is not respected the department has to pay a penalty. Ultimately, if the SLAs are violated too many times the service users may decide to terminate the service delivery contract before the IT department has compensated the initial service establishment costs. The reverse problem faced by the IT infrastructure manager is: if the service user has a specific requirement for the availability of the service, which QoS levels should be agreed with the Outsourcer for the outsourced building blocks such that the resulting internal service availability level meets the user requirements?

As we said, traditional approaches to the availability analysis are not quite applicable to this context. In fact, traditional availability analysis of complex systems requires the analyst to know for each system component the mean time to failure (MTTF) and mean time to recovery (MTBR) parameters. The personnel of the IT department can measure (or estimate) these parameters for the portion of the IT system which is under its direct control. However, it cannot do this for the parts (a large majority) that are managed by the Outsourcer. The only information the IT manager can rely on for outsourced components is the guaranteed minimal availability level agreed with the Outsourcer. Therefore, the IT infrastructure manager currently estimates the service availability levels based on simple heuristics (e.g. if he needs 0.99 availability for the service, he will choose at least 0.99 availability for each building block).

In our case study we addressed this problem using the A^{2}THOS framework. We structured the case study in two sub-cases. In the first sub-case we carried out the availability analysis of an IT system which has already been in place for some years. In the second sub-case we carried out both the availability analysis and the optimisation for a new IT system which is about to be deployed. Our results have been used by the IT manager both to set the internal SLAs for the new service and to choose the proper availability level of the building blocks of the system.

In the first sub-case, the IT system we analysed is the authorisation and authentication system of the Company, called Oxygen. To carry out the availability analysis we first needed to represent Oxygen as an AND/OR dependency graph. We extracted the information from the network diagram, the functional specification document, and the security architecture and design document. The procedure we followed is very similar to the one we described in our previous publication [31]. We used our tool to represent the AND/OR dependency graph of the system and to annotate nodes with their minimal availability level. The resulting graph consists of 65 nodes and 112 edges. Among the nodes are 13 IT services, 32 applications, 14 servers equally distributedbetween two data centres and connected simultaneously to two different network segments by means of two different firewalls.

The second step of this sub-case is to determine the minimal monthly availability for the nodes in the graph. We extracted this information from the SLA documentation attached to the standard contract signed between the Company and the Outsourcer. Finally, we extracted the current minimal monthly availability of the IT services supported by Oxygen from the Company internal SLAs documentation.

We used the analysis engine of our tool to carry out the availability analysis: the whole algorithm completed in less than 1 minute for Oxygen.

Table 4 reports the results of our analysis. We report in the first column the (anonymised) service, in the second column the minimal monthly availability level of each service calculated with our tool, and in the third column the existing minimal monthly availability level reported in the internal SLAs. Compared to the estimates made by the Company IT manager, we observe that the internal SLA specifies for Service1, Service4, Service5 and Service10 a minimal availability level which could not be guaranteed even in the case when the Outsourcer respects all its SLAs with the Company. This is a possible risk for the IT manager for the reasons we discussed above. On the other hand, we also see that the minimal monthly availability level we calculated for Service6, Service7, Service8, Service10, Service11, Service12 and Service13 is higher than the one specified in SLAs. This is also a criticality for the IT manager, as he is spending more money than needed to guarantee the availability level of the outsourced Oxygen building blocks.

Table 4. Results of the availability analysis on Oxygen

Service	Calculated α	Existing α
Service1	0.96	0.99
Service2	0.98	0.98
Service3	0.98	0.98
Service4	0.96	0.98
Service5	0.97	0.99
Service6	0.99	0.98
Service7	0.99	0.98
Service8	0.99	0.98
Service9	0.99	0.98
Service10	0.96	0.98
Service11	0.99	0.98
Service12	0.99	0.98
Service13	0.99	0.98

The system we analysed in the second sub-case is called Hydrogen and provides similar functionalities to Oxygen, but for the Company external contractors. Hydrogen was designed after Oxygen and is now in the final development phase. In this phase, the internal SLAs with the Hydrogen service users are already set, and the Company IT manager has to issue a request to the Outsourcer for the building blocks to deploy Hydrogen. He also has to specify in the request the desired availability level for each building block. Therefore, in this second phase of our case study we use the availability level optimisation of the A^{2}THOS framework.

The first step of this sub-case is the same as in the previous case: building the AND/OR dependency graph. To carry out this step we follow the same procedure we adopted for Oxygen (and described in more detail in our previous publication [31]). The resulting graph is made of 26 nodes and 33 edges. Secondly, we annotated the nodes with given availability. These nodes represent the data centres and the network segments. We acquired this information from the IT department personnel, which keep track of the monthly availability performances of their main IT infrastructure components. We set the given availability as the lowest monthly availability value observed in the monitoring data. Finally, we extracted the availability options for the eight variable availability nodes (servers). We obtained the required information from the building block description documents and the price catalogue provided by the Outsourcer to the Company. According to these documents, the Outsourcer offers three availability levels for the six Unix servers (0.995, 0.998 and 0.999) and two levels for the two Windows ones (0.995, 0.998). The resulting number of availability options is 733.

We used our tool to obtain the optimal configuration of availability levels for the servers of Hydrogen: the whole algorithm completed in less than 1 minute for Hydrogen. If the Company IT managers adopted the same strategy chosen for Oxygen (i.e. to choose the lowest availability level for all the outsourced components), they would have spent as little as possible, but two services of Hydrogen would have had a minimal availability lower than the one set in the internal SLAs. The optimal combination computed with our tool ensures that the minimal availability is compliant with the SLAs for all the services, with a cost which is only 2% greater. We also considered the effect of adding a further availability level (0.990) to the price catalogue of the outsourced components, extrapolating the associated price from the existing ones. The resulting optimal allocation in this case would be ~30% lower. The IT managers will take their decisions based on these results. In more detail, they will choose the optimal allocation we computed for Hydrogen. The Company may also start a renegotiation of the contract with the Outsourcer to discuss the introduction of a new availability level of 0.990 for servers. Based on interviews with personnel of the Company, we know the Company expects that the Outsourcer may agree on introducing a lower availability level for a lower price only if providing a lower availability would reduce its (internal) service delivery costs. In other words, if by decreasing the availability level the Outsourcer has an economic advantage, i.e. because of smaller maintenance costs, it will be able to agree on the introduction of the new availability level. On the other hand, in case there would be no (or too little) advantage, the Outsourcer would never agree on such a renegotiation.

Let us now address the questions we listed at the beginning of this section.

Does the approach scale up to conditions of practical utility? In this case study we applied A^{2}THOS to two distributed systems which are used by a large multinational company. The size of these systems is comparable to the size of the other systems the Company is using. The first scalability issue regards the time needed to build an AND/OR dependency graph for these systems. As we already argued above, the information required is available, but building an accurate AND/OR dependency graph for an IT manager can be time consuming. As the size of the system grows, the difficulty of choosing a (close to) optimal combination of availability levels for outsourced components grows more rapidly than the difficulty of building the graph. This suggests that it may be worth using A^{2}THOS for large systems, whose optimum component availability level combination is hard to find. Secondly, the case study confirms that our prototype can tackle large IT systems. We motivate this statement by the following two observations. First, the IT manager(s) of the Company need to decide about the availability of outsourced system components only when a new system is introduced in the IT infrastructure. In other words, the unit for the decision of the IT manager is limited to one system at a time. This is not surprising, if we consider that an organisation's IT infrastructure is incrementally built following the needs of the organisations: every time a new system is added to the infrastructure, only the availability issues of that system are taken into account. Secondly, the size (in terms of number of components) of the IT systems we analysed in our case study is comparable to the size of the other systems in the Company's IT infrastructure. We expect to find the same system size in other (large) organisations as well. According to these two observations, we can arguethat the performances of our prototype is sufficient in many practical situations, even with an IT system up to three times larger than the ones we considered.

Can A^{2}THOS *be (economically) applied to a practical case*? The fact that we were able to successfully carry out the case study presented above supports a positive answer to this question. However, it only shows that (a) *we* can use the method, and (b) that there exists one context in whichthe problem can be tackled by adopting our approach and in which the information required to adopt it was available.

Regarding (a), our application has not revealed obstacles to usage by *other* people, and further evidence would be needed to substantiate this claim positively.

Regarding (b), we have already discussed above the scalability of our approach for large organisations. We also believe that the context and problem we address in this paper are common to many medium/large-scale organisations. The problem is thus if the information required is available at those organisations. The information required to use A^{2}THOS can be summarised in (1) the components of the IT system under analysis and their functional dependencies, (2) the minimal availabilitylevel of each system component and (3) the different availability levels which can be chosen for outsourced components and the associated price. We learnt from this case study that most of the information regarding (1) can be extracted from the system functional and design documentation. The lacking parts can be easily integrated by interviewing the technical personnel who designed or implemented the system. Information regarding (2) is normally present in the SLA documentation for outsourced components. For components which are managed internally, this information can be extracted by measuring the component's performance over time (as was done by the Company in this case). It is also possible to calculate availability levels analytically, by using standard reliability techniques, as we mentioned in Section 2. Information regarding (3) is only available if the IT service provider allows its customers to choose among different availability options. Although not all outsourcing contracts with IT service providers may be configured in such a way, we learnt during our case study that the Outsourcer is applying this strategy to all of its customers. Therefore there are indications that say that A^{2}THOS is also applicable (at least) to these customers.

How useful is the information provided by A^{2}THOS *for its intended users*? The feedback we received from the IT management of the Company suggests a positive answer to this question. In particular, they found the information useful for: (1) making informed decisions about the planning of the availability of their IT services; (2) improving the quality of the IT services provided to the business units of the Company; and (3) justifying to the upper management the outsourcing costs in a more precise way. This provides support for our claim of potential usefulness of A^{2}THOS for its intended contexts of application.

DOI: 10.1002/nem

7. CONCLUDING REMARKS

In this paper we present A^{2}THOS, a framework for the analysis and optimisation of the availability of mixed-sourced IT services. The framework consists of (1) a modelling technique to represent partially outsourced IT systems, their components and the services they provide, based on AND/OR dependency graphs, (2) a procedure to calculate (a lower bound of) the system availability given the (lower bounds of) components availability, and (3) a procedure to select the optimum availability level for outsourced components in order to guarantee a desired target availability level for the service (s) and to minimise cost.

We analyse a scenario in which a contract between customer and service provider is made of a number of 'standard' building blocks arranged in a service catalogue for which the customer can choose the desired quality level. Based on the principles of economic game theory, assuming that the service provider acts rationally in its own interest, the customer has maximum economic advantage when making a cost-optimal choice of quality levels.

Among the different quality levels, the optimisation of availability levels is a non-trivial problem which can be tackled effectively using our modelling framework. Our benchmarks show that—even though the underlying problem is exponential—A^{2}THOS cantackle IT systems which are three times larger than the ones we could find at a multinational company.

APPENDIX A: PROOF OF THEOREM 1

Assuming without loss in generality that $\mathrm{DEPS}_m = \{D_1, \ldots, D_k\}$ and $D_i = \{d_{i,1}, \ldots, d_{i,l_i}\}$ we see that

$$\chi(m,t) = \prod_{i\in[1\,..\,k]} \left(\max_{j\in[1\,..\,l_i]} \chi(d_{i,j}, t)\right) \tag{9}$$

and calling $\chi(D_i, t) = \max_{j\in[1\,..\,l_i]} \chi(d_{i,j}, t)$ we obtain

$$\mathrm{av}(m) = \frac{1}{t_1 - t_0} \int_{t_0}^{t_1} \chi(m,t)\mathrm{d}t \tag{10}$$

$$= \frac{1}{t_1 - t_0} \prod_{i\in[1..k]} \chi(D_i, t)\mathrm{d}t \tag{11}$$

Let $\tau(a) = \{t \in [t_0, t_1] \mid \chi(a,t) = 1\}$ be the subset of the interval $[t_0, t_1]$ in which a functions correctly, since $\chi : N \times [t_0, t_1] \times \{0, 1\}$ and χ is measurable (for instance, in our context we can assume that, once we fix a node a, the graph of $\chi(a,t)$ switches from 0 to 1 a finite number of times) we have that

$$\int_{t_0}^{t_1} \chi(a,t)\cdot\chi(b,t)\mathrm{d}t = \int_{\tau(a)} \chi(b,t)\mathrm{d}t = \int_{\tau(a)\cap\tau(b)} 1\mathrm{d}t$$

Based on this, (10) becomes

$$\mathrm{av}(m) = \frac{1}{t_1 - t_0} \int_{\tau(D_1)\cap\ldots\cap\tau(D_k)} 1\mathrm{d}t \tag{12}$$

By set theory, if τ is a set and $A, B \subseteq \tau$, we have that $A \cap B = \overline{\overline{A}^\tau \cup \overline{B}^\tau}^\tau$, where $\overline{A}^\tau$ is the complement of A w.r.t. τ. Then, let $\tau = [t_0, t_1]$, we have that

$$\mathrm{av}(m) = \frac{1}{t_1 - t_0} \int_{\overline{\overline{\tau(D_1)}^\tau \cup\ldots\cup \overline{\tau(D_k)}^\tau}^\tau} 1\mathrm{d}t \tag{13}$$

Recall that, if $X \subseteq \tau$ and both are measurable,

$$\int_{\overline{X}^{\tau}} f\mathrm{d}t = \int_{\tau} f\mathrm{d}t - \int_{X} f\mathrm{d}t$$

Thus:

$$\mathrm{av}(m) = \frac{1}{t_1 - t_0}\left[\int_{\tau} 1\mathrm{d}t - \int_{\overline{\tau(D_1)}^{\tau} \cup \ldots \cup \overline{\tau(D_k)}^{\tau}} 1\mathrm{d}t\right] \tag{14}$$

Recall that if A and B are measurable and f with positive values, we have that $\int_{A\cup B} f\mathrm{d}t \leq \int_A f\mathrm{d}t + \int_B f\mathrm{d}t$, where the equality holds if A and B have empty intersection. Therefore we have that

$$\mathrm{av}(m) \geq 1 - \frac{1}{t_1 - t_0}\left[\int_{\overline{\tau(D_1)}^{\tau}} 1\mathrm{d}t + \ldots + \int_{\overline{\tau(D_k)}^{\tau}} 1\mathrm{d}t\right] \tag{15}$$

where the inequality becomes an equality if the D_is are pairwise disjoint. Intuitively,

$$\int_{\overline{\tau(D_i)}^{\tau}} 1\mathrm{d}t$$

is the unavailability caused by the nodes in D_i in the time interval $\tau = [t_0, t_1]$. In fact, let $D_i = \left\{d_{i,1}, \ldots, d_{i,l_i}\right\}$, we have that

$$\int_{\overline{\tau(D_i)}^{\tau}} 1\mathrm{d}t = \int_{\overline{\tau\left(d_{i,1}\right) \cup \ldots \cup \tau\left(d_{i,l_i}\right)}^{\tau}} 1\mathrm{d}t \tag{16}$$

By set theory, if $A, B \subseteq \tau$ we have that $\overline{A \cup B}^{\tau} = \overline{A}^{\tau} \cap \overline{B}^{\tau}$; then

$$\int_{\overline{\tau(D_i)}^{\tau}} 1\mathrm{d}t = \int_{\overline{\tau\left(d_{i,1}\right)}^{\tau} \cap \ldots \cap \overline{\tau\left(d_{i,l_i}\right)}^{\tau}} 1\mathrm{d}t \tag{17}$$

Recall that, if A and B are measurable,

$$\int_{A \cap B} 1\mathrm{d}t \leq \min\left(\int_A 1\mathrm{d}t, \int_B 1\mathrm{d}t\right)$$

Thus:

$$\int_{\overline{\tau(D_i)}^{\tau}} 1\mathrm{d}t \leq \min_{j\in[1..l_i]}\left((t_1 - t_0) - \int_{\tau\left(d_{i,j}\right)} 1\mathrm{d}t\right) \tag{18}$$

We can substitute in (18) the inequality with an equality as we are interested in the upper bound of the unavailability caused by the elements in D_i. Substituting (18) into (15) we obtain

$$\begin{aligned}\mathrm{av}(m) \geq\ & 1 \\ & - \min_{i\in[1\,..\,l_1]}\left(1 - \frac{1}{t_1 - t_0}\int_{\tau\left(d_{1,i}\right)} 1\mathrm{d}t\right) \\ & - \ldots \\ & - \min_{i\in[1\,..\,l_k]}\left(1 - \frac{1}{t_1 - t_0}\int_{\tau\left(d_{k,i}\right)} 1\mathrm{d}t\right)\end{aligned} \tag{19}$$

Let us now distinguish two cases: (a) D_is are pairwise disjoint, i.e. $\forall\, i, j\ D_i \cap D_j = \emptyset$, and (b) they are not pairwise disjoint. In (a) the inequality sign in (15) becomes an equalitysign. Therefore (15) becomes

$$\mathrm{av}(m) = 1 - \frac{1}{t_1 - t_0}\left[\int_{\overline{\tau(D_1)}^{\tau}} 1\mathrm{d}t + \ldots + \int_{\overline{\tau(D_k)}^{\tau}} 1\mathrm{d}t\right] \tag{20}$$

DOI: 10.1002/nem

and we can use (19) to determine av(m). Since all D_is are pairwise disjoint, if $d_{i,j}=n'$, then $\frac{1}{t_1-t_0}\int_{\tau(d_{i,j})}\mathrm{d}t=\frac{1}{t_1-t_0}\int_{t_0}^{t_1}\chi(n',t)\mathrm{d}t$. Therefore, if we set $\forall n'\in N'$ $\mathrm{av}(n')=\alpha_{n'}$, we determine α_m from (9) by substituting $\frac{1}{t_1-t_0}\int_{\tau(d_{i,j})}\mathrm{d}t$ with $\alpha_{n'}$ when appropriate.

In fact, in this case the various D_is are independent from each other and to calculate the minimal availability (given the constraints on the availability of the internal nodes) we can restrict the search to those schedulings in which the various D_is are unavailable in non-overlapping time frames and all the elements of any D_i are unavailable at exactly the same time. For example, consider the case of Figure 2(b). $\mathrm{DEPS}_m=\left\{\{m'\},\left\{n_1',n_2'\right\}\right\}$ and, according to (19), $\alpha_m=1-(1-\alpha_m)-\min((1-\alpha_{n_1}),(1-\alpha_{n_2}))$. The availability of m reaches α_m when (1) m' is unavailable for $1-a_{m'}$ but not at the same time for n_1 and n_2, and (2) n_1 and n_2 are unavailable at the same time for $\min((1-\alpha_{n_1}),(1-\alpha_{n_2}))$. This also shows that there exists a state function forwhich, when av$n'==\alpha_{n'}$, then $\mathrm{av}(m)=\alpha_m$.

In the general case (b), the elements D_i of DEPS_m are not pairwise disjoint, i.e. $\exists\, i,j \,|\, D_i\cap D_j\neq\emptyset$. By using (19) in this case we would obtain a value of α_m which is not minimal. To determine the availability lower bound α_m we then set up a linear programming problem. For the sake of presentation, we call $a_{i,j}$ the (unknown) quantity

$$a_{i,j}=\frac{1}{t_1-t_0}\int_{\tau_{d_{i,j}}}1\mathrm{d}t$$

and u_i the (unknown) quantity

$$u_i=1-\frac{1}{t_1-t_0}\int_{\overline{\tau(d_{i,1})}^{\tau}\cap\ldots\cap\overline{\tau(d_{i,l_i})}^{\tau}}1\mathrm{d}t$$

By substituting $a_{i,j}$ and u_i in (15) where possible, we obtain the objective function we need to minimise to find the lower bound we aim at. The first k constraints are derived from (17) and ensure that the nodes belonging to a certain D_i can be unavailable all at the same time for u_i.

Given the definition of dependency set, if we call $D(n')=\{d_{i,j}\,|\,d_{i,j}=n'\}$), then we know that

$$\begin{aligned}\mathrm{av}(n')&=\frac{1}{t_1-t_0}\int_{t_0}^{t_1}\chi(n',t)\mathrm{d}t\\&=1-\sum\nolimits_{d_{i,j}\in D(n')}1-\frac{1}{t_1-t_0}\int_{\tau(d_{i,j})}1\mathrm{d}t\end{aligned}\tag{21}$$

From (21) we derive a set of constraints which ensure that the lower bound of the availability caused by each internal node n' in all the elements of DEPS_m is the known value $\alpha_{n'}$.

As a result we get the following linear programming problem:

$$\alpha_m=\begin{cases}\text{minimize } 1-u_1-\cdots-u_k\\\text{subject to}\\u_1=\left(1-a_{1,1}\right)=\cdots=\left(1-a_{1,l_1}\right)\\\vdots\\u_k=\left(1-a_{k,1}\right)=\cdots=\left(1-a_{k,l_k}\right)\\\forall n\in N,\sum\nolimits_{d_{i,j}\in D_n}1-a_{i,j}\geq 1-\alpha_{n'}\\a_{1,1},\ldots a_{1,l_1},\ldots,a_{k,1},\ldots,a_{k,l_k}\geq 0\end{cases}\tag{22}$$

A solution to this problem can be found by using the simplex algorithm. Such a solution indicates both the lower bound for the availability of m, i.e. the minimal value of (15). With this we then proved our theorem.

DOI: 10.1002/nem

APPENDIX B: REPRESENTATION CAPABILITIES

In order to apply our approach to the real world one could wonder if the technique we adopt to represent the complex system (AND/OR dependency graphs) is expressive enough. A very popular and widely used approach to represent complex systems for reliability and availability analysis is using RBDs. In this section we show that our representation is at least as expressive as an RBD.

An RBD is a graphic representation of the complex system where every component is represented by a block (rectangle) and it is connected to other components, in series or parallel form. A serial connection between two blocks (see Figure 6a) means that the system (composed by the two blocks) is operational when both blocks are operational. A parallel connection (see Figure 6b) between two blocks means that the system is operational when at least one of the two blocks is operational. The whole system is then modelled as a combination of series and parallel blocks. A group of interconnected components can be represented as a single macro-component. In turn, macro-components can be connected to other components (e.g. see Figure 8) and grouped again. Hence, to prove that our representation is as expressive as an RBD, we need to show how each one of the three main operations on RBDs can be equally expressed as an AND/OR dependency graph.

If we consider the serial system of Figure 6(a), made of only two components, the corresponding AND/OR dependency graph is given in Figure 7(a). To represent the system we use an AND node X, which depends on nodes A and B. Similarly, the parallel system of Figure 6(b) is equivalent to the AND/OR dependency graph in Figure 7(b), where the OR node X depends on nodes A and B.

Regarding the composition operation, in Figure 8 we show the parallel composition of two components, where each of them is a serial composition of two sub-components, in parallel with two other sub-components. Figure 9 shows the same system represented as an AND/OR dependency graph. We model as an AND node X_1 the serial composition of sub-components A and B, and as an OR node X_2 the parallel composition of sub-components C and D. Finally, we add an OR node X which represents the parallel composition of the two above-mentioned components. It is easy to see

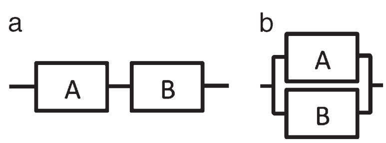

Figure 6. RBD: (a) series; (b) parallel

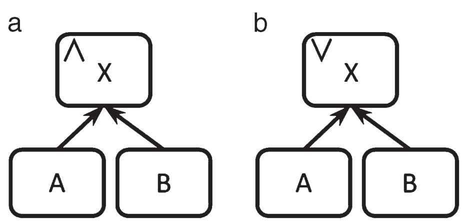

Figure 7. AND/OR dependency graph: (a) series; (b) parallel

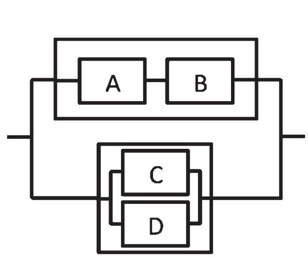

Figure 8. RBD parallel composition

DOI: 10.1002/nem

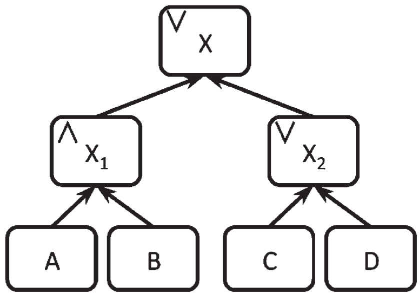

Figure 9. AND/OR dependency graph parallel composition

that we can model any combination of grouping and series/parallel compositions in the RBD, with a combination of AND and OR nodes.

ACKNOWLEDGEMENTS

The authors would like to thank Pieter Hartel and an anonymous reviewer for his valuable comments.

REFERENCES

1. Office of Government Commerce (OGC). *ITIL Version 3 Service Design*. TSO: London, 2007.
2. Ross S. *Introduction to Probability Models* (7th edn). Academic Press: New York, 1989.
3. Flammini F, Mazzocca N, Iacono M, Marrone S. Using repairable fault trees for the evaluation of design choices for critical repairable systems. In *HASE '05: Proceedings of the 9th IEEE International Symposium on High-Assurance Systems Engineering*, IEEE Computer Society: Washington, DC, 2005; 163–172.
4. Mayerson RB. *Game Theory: Analysis of Conflict*. Harvard University Press: Cambridge, MA, 1991.
5. Burgess M. System administration and micro-economic modelling. In *Handbook of Network and System Administration*, Bergstra J, Burgess M (eds). Elsevier: Amsterdam, 2007; 729–773.
6. Burgess M. An approach to understanding policy based on autonomy and voluntary cooperation. In *Proceedings of 16th IFIP/IEEE Distributed Systems Operations and Management (DSOM 2005)*. LNCS 3775. Springer: Berlin, 2005; 97–108.
7. Eisenhardt KM. Agency theory: an assessment and review. *Academy of Management Review* 1989; **14**(1): 57–74.
8. Brousseauand E, Glachant J-M (eds). *The Economics of Contracts: Theory and Applications*. Cambridge University Press: Cambridge, UK, 2002.
9. Keil P. Principal agent theory and its application to analyze outsourcing of software development. In *Proceedings of the 7th International Workshop on Economics-Driven Software Engineering Research*. ACM Press: New York, 2005; 1–5.
10. Gellings C. Outsourcing relationships: designing 'optimal' contracts: a principal–agent-theoretic approach. In *Proceedings of Americas Conference on Information Systems (AMCIS '06)*, 2006; 40–49.
11. Barlow RE, Proschan F. *Mathematical Theory of Reliability*. Society for Industrial and Applied Mathematics: Philadelphia, PA, 1996.
12. Maciejewskia H, Caban D. Estimation of repairable system availability within fixed time horizon. *Reliability Engineering and System Safety* 2006; **93**(1): 100–106.
13. Torres-Toledano JG, Sucar LE. Bayesian networks for reliability analysis of complex systems. In *IBERAMIA '98: Proceedings of the 6th Ibero-American Conference on AI*. Springer: London, 1998; 195–206.
14. Leangsuksun C, Song H, Shen L. Reliability modeling using UML. In *Software Engineering Research and Practice*, 2003; 259–262.
15. Vesely WE, Goldberg FF, Roberts NH, Haasl DF. Fault Tree Handbook. Technical report, US Nuclear Regulatory Commission NUREG-0492, 1981.
16. Dugan JB, Bavuso SJ, Boyd MA. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability* 1992; **41**(3): 363–377.
17. Distefano S, Xing L. A new approach to modeling the system reliability: dynamic reliability block diagrams. In *RAMS '06: Annual Reliability and Maintainability Symposium*, IEEE Computer Society: Washington, DC, 2006; 189–195.
18. IBM Tivoli. Available: http://www.ibm.com/software/tivoli/ [30 May 2011].
19. HP Business Availability Center. Available: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-15-25_4000_100__ [30 May 2011].
20. Sullivan KJ, Dugan JB, Coppit D. The Galileo fault tree analysis tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, IEEE Computer Society: Washington, DC, 1999; 232–235.
21. Boudali H, Crouzen P, Stoelinga MIA. A compositional semantics for dynamic fault trees in terms of interactive Markov chains. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis*. LNCS 4762. Springer: Berlin, 2007; 441–456.

22. Relex Software Corporation. Available: http://www.relex.com [30 May 2011].
23. BlockSim: system reliability and maintainability analysis software tool. http://www.reliasoft.com/BlockSim/ [30 May 2011].
24. Gu X, Nahrstedt K, Chang RN, Ward C. QoS-assured service composition in managed service overlay networks. In *International Conference on Distributed Computing Systems*. IEEE Computer Society: Los Alamitos, CA, 2003; 194.
25. Zeng L, Benatallah B, Ngu AHH, Dumas M, Kalagnanam J, Chang H. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering* 2004; **30**(5): 311–327.
26. Yu T, Lin K-J. Service selection algorithms for web services with end-to-end QoS constraints. In *IEEE International Conference on E-Commerce Technology*. IEEE Computer Society: Los Alamitos, CA, 2004; 129–136.
27. Ardagna D, Pernici B. Global and local QoS guarantee in web service selection. In *Business Process Management Workshops*, 2005; 32–46.
28. The ECLiPSe constraint programming system. Available: http://87.230.22.228/ [30 May 2011].
29. Smale S. On the average number of steps of the simplex method of linear programming. *Mathematical Programming* 1983; **27**(3): 241–262.
30. Rossi F, van Beek P, Walsh T. *Handbook of Constraint Programming*. Elsevier: Amsterdam, 2006.
31. Zambon E, Etalle S, Wieringa RJ, Hartel PH. Model-based qualitative risk assessment for availability of IT infrastructures. In *SOSYM: Software and System Modelling*, 2010; 1–28.

AUTHORS' BIOGRAPHIES

Emmanuele Zambon received his PhD at the University of Twente in 2011, he is now a researcher at the University of Twente. His two main research interests are on Information Risk Management and Intrusion Detection Systems. He graduated in 2005 in Computer Science at the Cá Foscari University of Venice with a thesis on Intrusion Detection Systems. During and after his studies he has been employed in the Information Risk Management group of KPMG and in Telecom Italia as a consultant, doing ethical hacking, network vulnerability/ assessment, and software development/performance tuning.

Sandro Etalle is professor of Embedded System Security at the Technical University of Eindhoven where he leads the chair of computer security, and part time professor at the University of Twente. In 2002, he started working on the protection of confidential data, a line of study which lead in 2004 to the definition of the first logic for accountability. His research interest include Trust Management, Policy Compliance, Intrusion Detection and Risk Management.

Roel J. Wieringa is professor of Information Systems at the University of Twente. His research interests lie in methods for software system requirements engineering and architecture design, and in the application of these methods to distribute information systems, varying from groupware to work-flow and e-commerce systems. His research includes the development of advanced tools to support design methods, including tools to simulate and explore possible system designs. He has published books and over 50 papers on software design methods.

DOI: 10.1002/nem