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
Email: {emmanuele.zambon, sandro.etalle, r.j.wieringa}@utwente.nl
[2] Technical University of Eindhoven
Eindhoven, The Netherlands
Email: s.etalle@tue.nl

SUMMARY

IT service availability is at the core of customer satisfaction and business success for today's organisations. Many medium-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.

KEY WORDS: SLA Management, Availability, Optimisation, Modelling

1. Introduction

Having a functional, cost effective and and properly managed IT infrastructure has become one of the main key success factors for all kinds of organisations. Nowadays, the 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 the so-called Service Level Agreement (SLA). (Figure 1 summarises the concept of mixed-sourced IT services regulated by SLAs.) For instance, ITIL [15] is one of the most popular frameworks providing guidelines and best practice for a correct IT service management and it describes this process in detail in [17].

In this paper we focus on IT service availability, which is at the core of customer satisfaction and business success for organisations [16], and indeed it is one of the main topics in a SLA. In fact a typical 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 two 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.

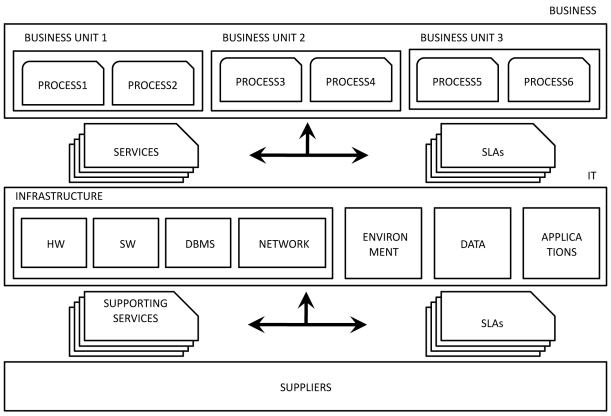


Figure 1: Mixed-sourced IT service provision regulated by SLAs.

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 the service becomes unavailable as well; other components my be organised in such a way (e.g., exploiting redundancy) that only if a number of them fails the service will 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 the customer needs and can be guaranteed by the technical infrastructure providing the service. To achieve this at the technical 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) [19] and Petri Nets [9]) 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 *minimal* 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 service catalogue of most IT outsourcing companies include different availability levels (e.g., gold, silver and bronze) with different associated prices (same service, only different availability levels, at different costs). Service providers need to minimise the cost of outsourced (sub)services while guaranteeing that their own service achieves the desired minimal availability level. Given the interactions mentioned above, this is a nontrivial 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 SLAs. This cannot be solved without the use of specific optimisation algorithms and typically IT managers choose non-optimal, conservative solutions.

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 *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 costs.

A dependency graph is an AND/OR 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 the information about components 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 the 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.

Limitation of the approach A^{2}THOS uses an AND/OR graph to represent the system, thus it is unable to explicitly represent failure recovery mechanisms such as 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 and Reliability Block Diagrams.

Organisation The rest of the paper is organised as follows. In Section 2 we present the related work in the fields of reliability and IT service composition. In Section 3 we present dependency graphs and we provide the mathematical foundation for using them to calculate service availability. In Section 4 we present the 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 performances. Finally, in Section 6 we show how we applied A^{2}THOS to a practical case of service availability planning in an industrial context.

2. Related Works

In this section we discuss related works in four relevant areas for our problem: (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.

The general approach Referring to a classic formulation [2] taken from the 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 [14]). 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): $\lambda = \frac{1}{MTBF}$. The second one specifies its Mean Time To Repair (MTTR): $\mu = \frac{1}{MTTR}$. Under this assumption the limiting availability is then obtained by 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 component. 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, etc.). To compute the availability of complex systems, Continuous Time Markov Chains (CTMC) [19], or Petri Nets [9] 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 a minimal availability in a given time frame (e.g., one 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.

System modelling Several approaches have been proposed in the literature for system reliability modelling. Fault trees (FTs) and Reliability Block Diagrams (RDBs) are the most used ones. However, we should mention that also other approaches have been proposed, e.g., Torres-Toledano and Sucar [22] use bayesian networks, and Leangsuksun et al. [13] use an 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 [9], FTs are easy to use, as they do not require very skilled modellers, and 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) [23]. Based on the minimal cut set, a combination of combinatorial techniques and CTMC or PetriNets is then used to calculate the system (limiting) availability.

According to Flamini et al. [9], the main limitation of FTs and RBDs consists in the lack of modelling power, as they do not allow to model maintenance-related issues explicitly. To solve this problem, FTs and RDBs have been extended into Dynamic Fault Trees [6] and Dynamic Reliability Block Diagrams [5], allowing one to model maintenance-related issues.

The modelling notation we use in this paper (dependency graphs) can be seen as a condensed form of fault trees. With a single dependency graph we are able to model a forest of fault trees sharing (some of) the basic events (i.e., the failure of a component), but with different top events. A single 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 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 dependency graphs is different from the one used in FTs (for the reason we mentioned above).

Tools IBM Tivoli [12] and HP Business Availability Centre [11] 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 [21], Coral [4], Relex [18] and BlockSim [3] are tools operating with Dynamic Fault Trees. 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.

Availability in service composition In the field of IT service composition, several approaches have been proposed that consider availability as one of the QoS parameters to optimise the performances of the resulting composite IT service. Gu et al. [10] 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. [26], Yu et al. [24] and Ardagna

et al. [1] propose scheduling techniques to create a cost-optimal execution plan for composite web services which respect QoS parameters (including availability) defined in SLA contracts.

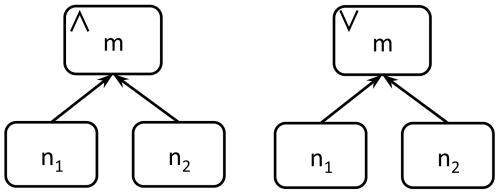
In all these works, an estimation of the availability of the composite service is made by multiplying the availability level of the components (expressed as a real number in the interval [0,1]. This is possible thanks to two simplifying assumptions. First, all the components must be available at the same time for the system to operate (i.e., the system is an AND-combination of its components and it becomes unavailable in the moment that any of its component 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. Differently 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 the 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 a *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 from the availability of n in a way that we are about to explain.

In a 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'.

On the other hand, 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* .



(a) *AND* (b) *OR*

Figure 2: Two simple dependency graphs, respectively with *AND* and *OR* nodes

If m is a node in a 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*
  - $n_1, \ldots, n_k$ are *all* unavailable at time t, in case m is an *AND* node,
  - at least one node in $n_1, \ldots, n_k$ is unavailable at time t, in case m is an *OR* node.

Formally,

Definition 3.1 (Dependency graph) *A* 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* $E \subseteq \{\langle u, v\rangle \mid u, v \in N\}$.

Given a graph $\langle N,\ E\rangle$, we call N' the set of the internal nodes of g; $N' = \{n' \text{ internal of } n \mid n \in N\}$.

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 five different servers (*`Srv1`*,* `Srv2`*,* `Srv3`*,* `Srv4`*,* `Srv5`*).* `Service1` *is implemented by* `App1` *and* `App2` *in such a way that the service goes off-line only when both applications are off-line (OR dependency).* `Service2` *is*

implemented by `App3`*, and* `App3` *depends 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 on-line.* `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 off-line. Finally, the system is protected by the firewall* `FW1`*. According to this description, we build the 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:* `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 dependency graph of our running example.*

The *state function* χ we are about to introduce models the availability of (the internal node of) a node. Given a node m, $\chi(m', t)$ is 0 iff m at time t suffers an internal failure, and is 1 otherwise. Similarly, $\chi(m, t) = 0$ indicates that the node m is unavailable at time t. As explained above, the *state function* of a node m is always 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 3.2 (State Function) *Let* $g = \langle N, E \rangle$ *be a dependency graph. We say that* χ *is a* state function *for* g *iff* $\chi : (N \cup N') \times \mathbb{R}^+ \to \{0, 1\}$*, and for each* $m \in N$ *and* $t \in \mathbb{R}^+$ *the following holds: let* $n_1, \ldots, n_k$ *be the nodes in* N m *depend on. Then*

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

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

So, given the state function of all the internal nodes in a 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 is well defined).

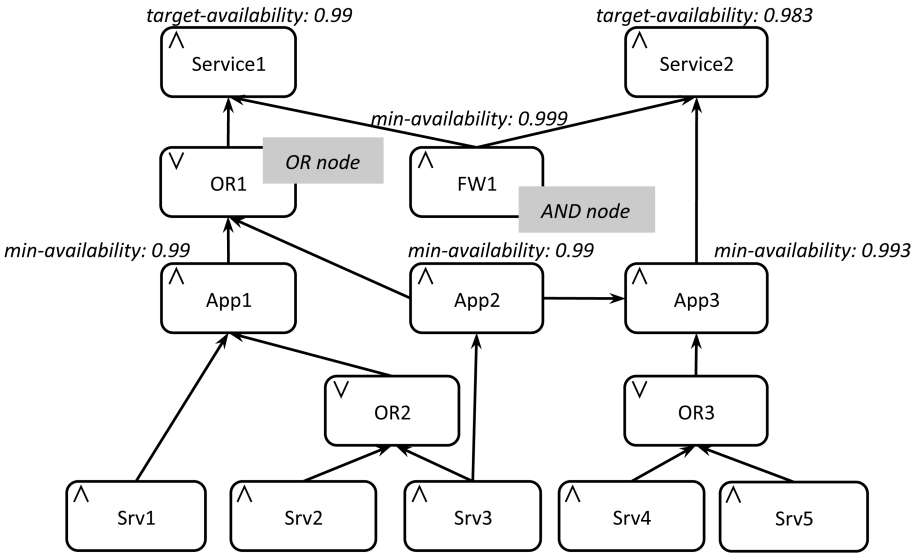


Figure 3: The dependency graph representing the system we analyse in our running example. AND nodes are represented by the ∧ symbol, OR nodes by the ∨ symbol.

According to the dependability theory [2], 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:

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

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

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). Notice 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 off-line for more than 7.2 hours in a row, while in the second case the system may be off-line for up to 87.6 hours in a row. Equation, (2) can be seen as a formalisation of the availability parameter used in SLAs.

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 the server is deployed in and the network the server is connected to 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:

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

Given an arbitrary node $m \in N$, what can we say about $av(m)$ in the same time period? In particular, can we compute a lower bound for it?

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 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 of the minimal cut sets of a fault tree, and can be obtained similarly by representing the graph as a boolean equation and the using substitution methods to reduce the equation. We will now present a more formal definition of dependency sets.

Definition 3.3. *Consider a dependency graph $g = \langle N, E \rangle$ and a node $m \in N$. The* dependency set *of m,* $\text{DEPS}_m \subseteq \mathcal{P}(\text{N'})$*, is defined inductively as follows.*

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

Running example - Part 2. *By applying the recursive Definition 3.3 to our example dependency graph, we obtain the following dependency sets for* `Service1` *and* `Service2`*:* $\text{DEPS}_{\texttt{Service1}} = \{\{$`FW1'`$\}, \{$`App1', App2'`$\}, \{$`App1, Srv3'`$\}, \{$`Srv1', App2'`$\}, \{$`Srv1', Srv3'`$\}, \{$`App2', Srv2', Srv3'`$\}, \{$`Srv2',Srv3'`$\}$. $\text{DEPS}_{\texttt{Service2}} = \{\{$`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* $\text{DEPS}_{\texttt{Service1}}$ *are unavailable at the same time,* `Service1` *is unavailable, and the same for* $\text{DEPS}_{\texttt{Service2}}$.

The dependency set of a node is always a set of sets of internal nodes, so without loss of generality, we can always write $\mathit{DEPS}_m = \{D_1, \ldots, D_k\}$, where for all $i \in [1 \,..\, k]$ $D_i = \{d_{i,1}, \ldots, d_{i,l_i}\}$, for appropriate $l_i, d_{i,1}, \ldots, d_{i,l_i}$. Note also that for all $j \in [1 \,..\, l_i]$, $\exists n \in N \mid d_{i,j} = n'$.

As for minimal cut sets in fault trees, 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, i.e., More formally, if we fix a time t then

$$\chi(m,t) = 0 \Leftrightarrow \exists D \in \mathit{DEPS}_m, \forall d \in D, \chi(d,t) = 0 \tag{4}$$

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

Theorem 3.4. *Let $g = \langle N, E\rangle$ be a 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, such that for each state function χ for g the following holds:*

$$\textit{IF } \forall n' \in N' \qquad \frac{1}{t_1 - t_0}\int_{t_0}^{t_1} \chi(n',t)dt \;\geq\; \alpha_{n'} \tag{5}$$

$$\textit{THEN} \qquad \frac{1}{t_1 - t_0}\int_{t_0}^{t_1} \chi(m,t)dt \;\geq\; \alpha_m \tag{6}$$

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 following linear programming problem.

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

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 $\mathit{DEPS}_m = \{D_1, \ldots, D_k\}$, and for each D_i we can write $D_i = \{d_{i,i}, \ldots, d_{i,l_i}\}$. According to this notation, u_i in (7) represent the unavailability caused to m by D_i and $a_{i,j}$ represent the availability of the element $d_{i,j} \in D_i$. Given two elements $D_1, D_2 \in \mathit{DEPS}_m$, 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 (7) we call D_n the set of elements $d_{i,j}$ which all refer to the same node n. The objective function of (7) represents the availability of node m, which is expressed as 1 less the unavailability caused by each element in DEPS_n. The first k conditions impose that the internal nodes of each element $D_i \in \mathit{DEPS}_n$ are unavailable at the same time: this ensures m is disrupted because of the simultaneous failure of all the internal nodes in D_i. 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 $(1 - \alpha_{n'})$. 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.

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

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

Which gives a lower bound for the availability of `Service1` *of 0.984. Similarly, we can determine* $\alpha_{\texttt{Service2}} = 0.972$*. Figure 4 shows one possible scheduling for the failure of the components on which* `Service1` *depends on, resulting in* `Service1` *having an availability of* $\alpha_{\texttt{Service1}}$ *(0.984).*

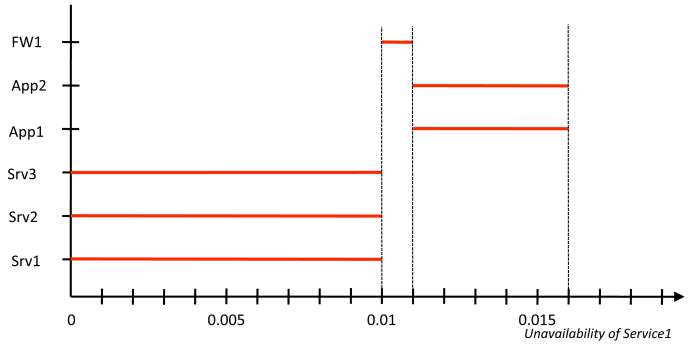


Figure 4: One possible scheduling for the failure of `FW1`, `App1`, `App2`, `Srv1`, `Srv2` and `Srv3` resulting in `Service1` having an availability of 0.984. System components are on the vertical axis and the components unavailability fraction of time ($\in [0, 1]$) is on the horizontal axis.

4. Optimisation of outsourced services

In the last section we have seen that determining the minimal availability level of a complex system is a non-trivial problem, that can be solved by reducing it to an optimisation problem. A relevant application of this result is the minimisation of the costs of outsourced subcomponents. Given that outsourcing has a cost that may depend (also) on the minimal availability guaranteed for the outsourced component, a manager typically needs to minimise the costs of the outsourcing while guaranteeing that the services provided by the system meet 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 choose among 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 a dependency graph: *target* availability nodes, *variable* availability nodes and *given* availability nodes. More formally, given a 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 traget 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 dependency graph, whenever there is an SLA (be it company internal or not) which imposes a certain level of availability for them.

Definition 4.1 (Target availability level) *Given a 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 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($\texttt{Service1}$) $= 0.99$ *and* target-availability($\texttt{Service2}$) $= 0.983$.

Variable availability nodes model the situation in which it is possible to choose the availability level of a component among different options. A typical case of *variable* availability level is when the management of system components is outsourced to another department or to another company: in these cases, the system manager may have the possibility to choose for each component a different availability level (e.g., gold, silver and bronze), with different quality level and a different associated price.

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

Definition 4.2 (Availability option) *Let $\langle N, E\rangle$ be a dependency graph, N_V be a set of variable availability nodes and N'_V be the set of the internals of the elements in N_V.*

- *An* availability assignment *for N_V is a function* aa : $N'_V \to [0, 1]$.
- *An* availability option *for N_V is a pair $\langle$*aa$, c\rangle$*, where* aa *is an availability assignment and $c :\in \mathbb{R}$ is the cost associated to it. We call O the set of all the availability options.*

Usually, a company might choose between different outsourcing options. Notice that the outsourcing option is given on a set of nodes, and not on a single node. This allows us to model bulk discounts, i.e., the fact that outsourcing – say – ten components is usually less expensive than ten times the outsourcing of a single components. (For those who are familiar with optimisation problems: this introduces a form of non-monotonicity in the outsourcing offers, where the 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.)

Table I: An example of availability level options price catalog

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

Running example - Part 5. *An outsourcing company has provided an offer for managing the servers in our example system. The offer includes different availability level options for the management of Windows servers and UNIX servers. Table I summarises the price catalogue for this offer. 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* $N_V = \{\texttt{Srv1}, \texttt{Srv2}, \texttt{Srv3}, \texttt{Srv4}, \texttt{Srv5}\}$. *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* $\langle \text{aa}_1, 4800\rangle$ *where* $\text{aa}_1(\texttt{Srv1}') = \text{aa}_1(\texttt{Srv2}') = \text{aa}_1(\texttt{Srv3}') = \text{aa}_1(\texttt{Srv4}') = \text{aa}_1(\texttt{Srv5}') = 0.99$, *and* $\text{c} = 3 \cdot 1000 + 2 \cdot 900 = 4800$.

To guarantee that the 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 model components for which the minimal availability is known and not variable.

Definition 4.3 (Given availability level) *Given a 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 give are* `FW1`, `App1`, `App2` *and* `App3`. *Therefore, according to Figure 3 we have that* given-availability$(\texttt{FW1}) = 0.999$, given-availability$(\texttt{App1}) = 0.99$, given-availability$(\texttt{App2}) = 0.99$ *and* given-availability$(\texttt{App3}) = 0.993$.

We now can give a more formal definition of the problem at hand. Let $\langle N, E\rangle$ be a dependency graph, $N = N_T \dot\cup N_V \dot\cup N_G$, a function *target-availabiliy* 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).

As a result, we obtain the optimal assignment of variable availability levels as the solution to the linear programming problem with variables in a finite domain given by (8).

$$\begin{cases} \text{choose } o = \langle aa, c\rangle \in O \text{ to} \\ \text{minimize } c \\ \text{subject to:} \\ \textit{mimimum-availability}(t_1, o) \geq \textit{target-availability}(t_1) \\ \vdots \\ \textit{minimal-availability}(t_P, o) \geq \textit{target-availability}(t_P) \end{cases} \tag{8}$$

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:*

$$\begin{cases} \textit{choose } o = \langle \text{aa}, c\rangle \in O \textit{ to} \\ \textit{minimize } c \\ \textit{subject to:} \\ \text{minimal-availability}(\texttt{Service1}, o) \geq 0.99 \\ \text{miniaml-availability}(\texttt{Service2}, o) \geq 0.983 \end{cases}$$

Which gives us a (optimal) solution with cost 6300Euro *when* $\alpha_{\texttt{Srv1}'} = 0.998$, $\alpha_{\texttt{Srv2}} = 0.99$, $\alpha_{\texttt{Srv3}} = 0.998$, $\alpha_{\texttt{Srv4}} = 0.99$ *and* $\alpha_{\texttt{Srv5}} = 0.998$.

5. Implementation and benchmarks

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 [7]

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.

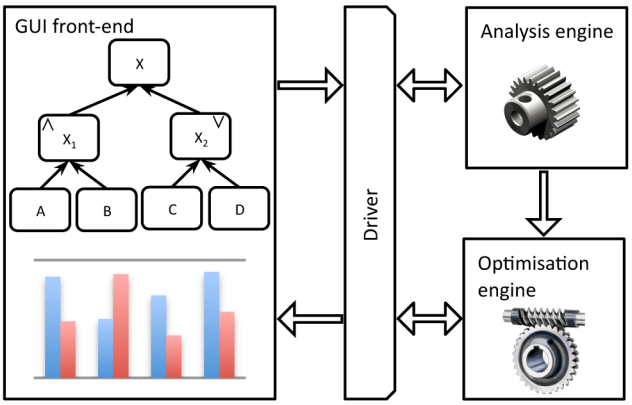


Figure 5: A^{2}THOS architecture

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 dependency graph by dragging and dropping nodes and edges, to annotate each node with its availability figure(s) or availability level options and to view the analysis and optimisation results. The analysis engine solves the availability analysis 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 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.

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 [20], 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 with no doubt be improved: our goal is to demonstrate how this can be done, and not that of providing 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 by increasing 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 II. 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 a dependency graph of 250 nodes and 600 constraints on average in less than a minute.

Table II: 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

Secondly, we benchmarked the optimisation algorithm. Our prototype implementation works by exhaustive 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 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 of 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 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.

Table III: 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

Our results indicates 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 III 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 independent in one hour and a half. However trend is – as expected – exponential, and with 30 independent nodes we exceed six 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 it is 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 6 independent nodes).

In the unlikely case that one would need to apply the algorithm to a too large 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* [8]) 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.

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. can A^{2}THOS be applied to a practical case, i.e., does it not require information not available in practice?
2. does the prototype we implemented scale up to conditions of practice?
3. does A^{2}THOS yield information useful 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, like 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 Quality of Service (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 Service Level Agreements (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 standard contract regulating the application and server management service provisioning. The standard 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 of each building block are also standardised. 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 catalog 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, a too low value could prevent the agreement to be 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, a too high 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 require 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 need 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 is already 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 a 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 paper about a risk assessment technique, which we tested on the same system [25]. We used our tool to represent the 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 distributed between 2 datacenters and connected simultaneously to 2 different network segments by means of 2 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 one minute for Oxygen.

Table IV 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 the existing minimal monthly availability level reported in the internal SLAs in the third column. 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

Table IV: 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

building blocks.

The system we analysed in the second sub-case is called Hydrogen and provides similar functionalities as Oxygen, but for the Company external contractors. Hydrogen has been 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 dependency graph. To carry out this step we follow the same procedure we adopted for Oxygen. The resulting graph is made of 26 nodes and 33 edges. Secondly, we annotated the nodes with given availability. These nodes represent the datacenters and the network segments. We acquired this information from the IT department personnel, which keeps 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 one 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 cost from the existing ones. The resulting optimal allocation in this case would be $\sim$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, and will negotiate with the Outsourcer the introduction of a new availability level of 0.990 for servers.

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

Can the method be 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 *we* can use the method. Our application has not revealed obstacles to usage by *other* people, but further evidence would be needed to substantiate

this claim positively. It is also interesting to discuss which other contexts A^{2}THOS can be applied to. 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 availability level of each system component and (3) the different availability levels which can be chosen for outsourced components and the associated cost. To use A^{2}THOS in other contexts, these three pieces of information must be available. 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 which 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 it 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 some IT service providers do not provide this feature, 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.

Does the prototype we implemented scale up to conditions of practice? 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 a dependency graph for these systems. As we already argued above, the information required is available, but building an accurate dependency graph for an IT manager can be time consuming. As the size of the system grows, the difficulty of choosing an (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 take decisions about the availability of outsourced system components for each new system 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 this two observations, we can argue that the performances of our prototype are sufficient in many practical situations, even with an IT system up to three times larger than the ones we considered.

Does A^{2}THOS yield information useful for its intended users? The feedback we had from the IT management of the Company suggests a positive answer to this question. In particular, they found the information useful for: (1) taking informed decision about the planning of the availability of their IT services, (2) improve the quality of the IT services provided to the business units of the Company and (3) justify 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.

7. Conclusions

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 *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 costs. The engine we have used is a proof-of-concept, and its speed can certainly be improved. This is however outside the scope of this paper.

We have analysed the SLAs of a few organisations and we concluded that in case of outsourcing, these SLAs were sub-optimal: the final availability levels were achievable with less expensive means (in some cases, even better availability levels were achievable at a lower cost). This is not surprising as the optimisation of the SLAs is a non-trivial problem which in practice is "solved" by educated guess by the chief IT officer. We have shown in this paper that this problem can also be tackled effectively using our modelling framework. Our benchmarks show that – even though the underlying problem is exponential – A^{2}THOS can tackle IT systems which are three times larger than the ones we could find at a multinational company. Given that the optimisation of the SLAs can save (immediately) company money while leaving the global service level unchanged, we believe that out framework could be profitably applied in practice.

Acknowledgment

The authors would like to thank Pieter Hartel for his valuable comments.

A. Proof of Theorem 3.4

Proof Assuming without loss in generality that $DEPS_m = \{D_1, \ldots, D_k\}$ and $D_i = \{d_{i,1}, \ldots, d_{i,l_i}\}$ we see that:

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

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

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

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

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

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

Based on this, (10) becomes

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

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

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

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

$$\int_{\overline{X}^{\tau}} f dt = \int_{\tau} f dt - \int_{X} f dt$$

thus:

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

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

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

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

$$\int_{\overline{\tau(D_i)}^{\tau}} 1dt$$

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

$$\int_{\overline{\tau(D_i)}^{\tau}} 1dt = \int_{\overline{\tau(d_{i,1}) \cup \ldots \cup \tau(d_{i,l_i})}^{\tau}} 1dt \tag{16}$$

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

$$\int_{\overline{\tau(D_i)}^{\tau}} 1dt = \int_{\overline{\tau(d_{i,1})}^{\tau} \cap \ldots \cap \overline{\tau(d_{i,l_i})}^{\tau}} 1dt \tag{17}$$

Recall that, if A and B are measurable,

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

thus:

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

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:

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

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 equality sign. Therefore (15) becomes

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

and we can use (19) to determine $av(m)$. Since all D_is are pairwise disjoint, if $d_{i,j} = n'$, then $\frac{1}{t_1-t_0}\int_{\tau(d_{i,j})} dt = \frac{1}{t_1-t_0}\int_{t_0}^{t_1} \chi(n', t)dt$. Therefore, if we set $\forall n' \in N'\ av(n') = \alpha_{n'}$, we determine α_m from (19) by substituting $\frac{1}{t_1-t_0}\int_{\tau(d_{i,j})} dt$ 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. $DEPS_m = \{\{m'\}, \{n'_1, n'_2\}\}$ 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 $1 - \alpha_{m'}$ but not at the same time of 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 it exists a state function for which, when $avn' == \alpha_{n'}$, then $av(m) = \alpha_m$.

In the general case (b), the elements D_i of $DEPS_m$ are not pairwise disjoint, i.e., $\exists i, j \mid D_i \cap D_j \neq \emptyset$. 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

$$\frac{1}{t_1 - t_0} \int_{\tau_{d_{i,j}}} 1dt$$

and u_i the (unknown) quantity

$$1 - \frac{1}{t_1 - t_0} \int_{\overline{\tau(d_{i,1})}^{\tau} \cap \ldots \cap \overline{\tau(d_{i,l_i})}^{\tau}} 1dt$$

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 $D(n') = \{d_{i,j} \mid d_{i,j} = n'\}$), then we know that

$$\begin{array}{rcl} av(n') & = & \frac{1}{t_1 - t_0} \int_{t_0}^{t_1} \chi(n', t) dt \\ & = & 1 - \sum_{d_{i,j} \in D(n')} 1 - \frac{1}{t_1 - t_0} \int_{\tau(d_{i,j})} 1dt \end{array} \tag{21}$$

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:

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

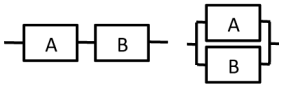
A solution to this problem can be found by using the simplex algorithm. Such a solution indicates both the lower bound for the availability of m, i.e., the minimal value of (15) With this we then proved our theorem.

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 (dependency graphs) is expressive enough. A very popular and widely used approach to represent complex systems for reliability and availability analysis is using Reliability Block Diagrams (RBDs). In this section we show that our representation is at least as expressive as an RBD.

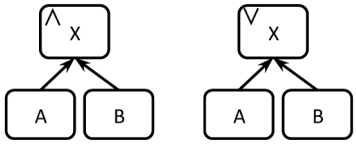
An RDB is a graphic representation of the complex system where every component is represented by a block (rectangle) and it is connected to other components, in series or parallel form. A serial connection between two blocks

(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 are 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 a dependency graph.



(a) Series (b) Parallel

Figure 6: Reliability Block Diagram



(a) Series (b) Parallel

Figure 7: Dependency graph

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

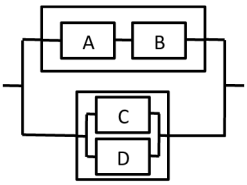


Figure 8: Reliability Block Diagram parallel composition

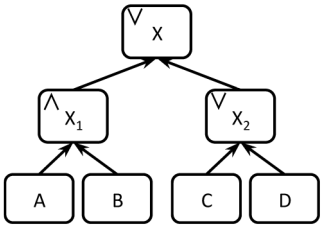


Figure 9: Dependency graph parallel composition

Regarding the composition operation, in Figure 8 we show the parallel composition of two components, where each of them is a serial composition two sub-components, in parallel with two other sub-components. Figure 9 shows the same system represented as a 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 that we can model any combination of grouping and series/parallel compositions in the RBD, with a combination of *AND* and *OR* nodes.

REFERENCES

1. D. Ardagna and B. Pernici. Global and Local QoS Guarantee in Web Service Selection. In *Business Process Management Workshops*, pages 32–46, 2005.
2. R.E. Barlow and F. Proschan. *Mathematical Theory of Reliability*. SIAM: Society for Industrial and Applied Mathematics Philadelphia, 1996.
3. BlockSim: System Reliability and Maintainability Analysis Software Tool. http://www.reliasoft.com/BlockSim/.
4. H. Boudali, P. Crouzen, and M.I.A. Stoelinga. A compositional semantics for Dynamic Fault Trees in terms of Interactive Markov Chains. In *Proc. of the 5th International Symposium on Automated Technology for Verification and Analysis*, pages 441–456. LNCS, 2007.
5. S. Distefano and L. Xing. A New Approach to Modeling the System Reliability: Dynamic Reliability Block Diagrams. In *RAMS '06: Annual Reliability and Maintainability Symposium*, pages 189–195, Washington, DC, USA, 2006. IEEE Computer Society.
6. J.B. Dugan, S.J. Bavuso, and M.A. Boyd. Dynamic Fault-Tree Models for Fault-Tolerant Computer Systems. *IEEE Transactions on Reliability*, 41(3):363–377, September 1992.
7. The ECLiPSe Constraint Programming System. http://87.230.22.228/.
8. Elsevier, editor. *Handbook of Constraint Programming*. F. Rossi, P. van Beek and T. Walsh, August 2006.
9. F. Flammini, N. Mazzocca, M. Iacono, and S. Marrone. Using Repairable Fault Trees for the evaluation of design choiches for critical repairable systems. In *HASE '05: Proc. Ninth IEEE International Symposium on High-Assurance Systems Engineering*, pages 163–172, Washington, DC, USA, 2005. IEEE Computer Society.
10. X. Gu, K. Nahrstedt, R.N. Chang, and C. Ward. QoS-Assured Service Composition in Managed Service Overlay Networks. In *International Conference on Distributed Computing Systems*, page 194, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
11. HP. HP Business Availability Center. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-15-25_4000_100__, August 2009.
12. IBM. IBM Tivoli. http://www.ibm.com/software/tivoli/, August 2009.
13. C. Leangsuksun, H. Song, and L. Shen. Reliability Modeling Using UML. In *Software Engineering Research and Practice*, pages 259–262, 2003.
14. H. Maciejewskia and D. Caban. Estimation of repairable system availability within fixed time horizon. *Reliability Engineering & System Safety*, 93(1):100–106, January 2006.
15. Office of Government Commerce (OGC). *Introduction to the ITIL Service Lifecycle*. TSO (The Stationery Office), 2007.
16. Office of Government Commerce (OGC). *ITIL Version 3 Service Design*. TSO (The Stationery Office), 2007.
17. Office of Government Commerce (OGC). *ITIL Version 3 Service Strategy*. TSO (The Stationery Office), 2007.
18. Relex Software Corporation. http://www.relex.com.
19. S. Ross. *Introduction to Probability Models, Seventh Edition*. Harcourt Academic Press, 1989.
20. S. Smale. On the average number of steps of the simplex method of linear programming. *Mathematical Programming*, 27(3):241–262, October 1983.
21. K.J. Sullivan, J.B. Dugan, and D. Coppit. The Galileo fault tree analysis tool. In *Proc. of Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 232–235. IEEE Computer Society, 1999.
22. J.G. Torres-Toledano and L.E. Sucar. Bayesian Networks for Reliability Analysis of Complex Systems. In *IBERAMIA '98: Proceedings of the 6th Ibero-American Conference on AI*, pages 195–206, London, UK, 1998. Springer-Verlag.
23. W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. Fault Tree Handbook. Technical report, US Nuclear Regulatory Commission NUREG-0492, 1981.
24. T. Yu and K-J. Lin. Service Selection Algorithms for Web Services with End-to-End QoS Constraints. In *IEEE International Conference on E-Commerce Technology*, pages 129–136, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
25. E. Zambon, S. Etalle, R.J. Wieringa, and P.H. Hartel. Architecture-based Qualitative Risk Analysis for Availability of IT Infrastructures. Technical Report TR-CTIT-09-35 ISSN 1381-3625, Centre for Telematics and Information Technology, University of Twente, Enschede, 2009. http://eprints.eemcs.utwente.nl/15983/.
26. L. Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1291834.