JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 0000; **00**:1–35
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

On-line verification in cyber physical systems: Practical bounds for meaningful temporal costs

Marcello M. Bersani[1], Marisol García-Valls[2*]

[1]Dipartamento di Elettronica, Informazione e Bioingegneria Politecnico di Milano, Milano, Italy
[1]Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid, Leganés (Madrid), Spain

SUMMARY

Cyber-physical systems (CPS) are highly dynamic and large scale systems integrated with the physical environment that they monitor and actuate on. Given the changing nature of physical environments, CPS have to adapt on-line to new situations while preserving their *correct* operation. This means that the system model may have to change or, at least, will have to be modified during its operation life, preserving correctness. *Correctness by construction* relies on using formal tools, which suffer from a considerable computational overhead. As the current system model of a CPS may adapt to the environment, the new system model must be verified before its execution to ensure that the properties are preserved. However, CPS development has mainly concentrated on the design-time aspects, existing only few contributions that address their on-line adaptation.
We design a framework for managing dynamic changes of a system based on a core entity that is an autonomic manager; we investigate the pros and cons of using formal tools within this framework to guarantee that the system properties are met at all times and across changes. We formalize the semantics of the adaptation logic of an autonomic manager (OLIVE) that performs on-line verification for a specific application, a dynamic virtualized server system. The on-line verification manager services requests from mobile clients that might require a change in both the running software components and services executed by the server. We explore the use of formal tools based on CLTLoc to express functional and non-functional properties of the system. In this scenario, we provide empirical results showing the temporal costs of our approach.

Received . . .

1. INTRODUCTION

Cyber physical systems [14] are systems with strict timing requirements, and they are at the confluence area among embedded, real-time, wireless sensor networks, and control systems. CPS are new with respect to these traditional fields due to their scale and to the uncertainty created by the surrounding environment, which can influence the system behavior both sporadically and unexpectedly. The deviation of the external systems (interacting with the CPS) from their nominal behavior or the occurrence of new situations in the environment might force the system to adopt strategies for dealing with changes in a timely manner. Adapting the functionality to a new environment allows the system to fit new needs and preserve the designed operations in contexts that are different from the original one.

*Correspondence to: mvalls@it.uc3m.es, Departament of Telematic Engineering, Universidad Carlos III de Madrid, Leganés (Madrid), Spain.

In this paper, we focus mainly on the autonomous nature of cyber-physical systems, that refers to the capacity of making correct decisions in a timely manner with the available information (that will be mostly local and also obtained from external subsystems through some communication media). Some CPS are decentralized systems where each one of their constituent subsystems may have very distinct functional requirements. This is the case of, for instance, robot squads (that are autonomous mobile nodes) performing remote monitoring and surveillance, moving around hostile environments with limited (or no) human presence. Mobile nodes must be efficient in the usage of resources by, for instance, requesting heavy operations to be performed by other powerful nodes such as servers.

Such servers are part of the cyber-physical system and must operate in a timely manner. Servers may execute a number of functions or applications that can be of different criticality levels. To ensure the temporal and spatial isolation among the various applications, more and more servers execute virtualized software platforms. For example, in [19], a distributed system based on partitioned/virtualized servers is described where the communications among them are enabled by middleware (using DDS –Data Distribution Service–) and the overall operation of the servers respects temporal deadlines. The virtualization technology has made significant improvements also in those domains requiring reliable and predictable execution. Although virtualization software challenges the predictability and reliability levels required by real-time systems [16], some solutions that start to overcome quite a few of the problems are available to (at least) provide quality of service (QoS) guarantees to soft real-time domains. To have a flexible software organization of the server, virtualization offers heterogeneous execution environments in the same physical machine. As real-time virtualizers guarantee execution isolation among virtual machines, it is currently possible to achieve the coexistence of applications with different criticality levels and requirements with respect to timeliness, reliability, or security.

This work considers client-server applications in a cyber-physical context with a specific focus on the adaptation problem of autonomic system managers. Adaptation in CPS can be related to the principles of *autonomic computing*, a term firstly used to describe *self-managing* [12] systems. An autonomic manager is a fundamental tool to implement self-managing functionalities. In the case study that we consider, the autonomic manager is included in the server software architecture to arbitrate the adaptation process at runtime. Specifically, it determines whether the request of a mobile node can be serviced or not depending on the current resource availability and possibly new requirements of the client nodes.

The implementation of adaptive software has to follow rigorous techniques, that apply both to the initial design and to the on-line adaptation of the server. There are a number of design methodologies, techniques, and technologies for this such as described in [22]. *Automatic verification* is the formal technique that we adopt in this work and that the autonomic manager leverages to support the decision process. Autonomic managers often represent the system behavior by means of a behavioral model that is updated on-line upon the system changes. Given that the server software configuration may adapt to new situations (e.g. an incoming request from a mobile node which may require a new functionality to be executed/downloaded) and that the server model is modified on-line to handle such changes, the autonomic manager can verify that the needed incremental updates in the model comply with the system specification.

Formal tools are needed to carry out the above mentioned on-line verification of the properties of the new model. This is, however, a hard task due to the inherent high overhead imposed by the solvers. This paper recognizes that there is not a *one-solution-fits-all* needs of dynamic CPS and that individual formal techniques have to be studied for each individual system (or a kind of systems) to effectively overcome their limits.

This work provides a practical illustration of the previous claim for a kind of systems, such as virtualized servers, that rely on the usage of spatially and temporally isolated partitions or virtual machines. Using partitioned systems is the de facto approach in critical software systems that are transitioning from federated architectures to integrated modular functionality. Precisely, this is widely used in airborne software systems that are based on integrated modular avionics (IMA) [24]. We present the software structure of a virtualized server and we focus on its *On-Line Verification Entity* (OLIVE). OLIVE is based on MAPE-K model [11, 12], and it is designed to operate while

DOI: 10.1002/smr

the server is in execution. OLIVE decides if a request from a mobile node can be processed by the server providing the answer in bounded time. We experiment with the usage of temporal logic in OLIVE to prove that the decision can be taken within the specified time. CPS require that such a time is determined a priori. In this paper, this is achieved through a-priori verification of the system adaptation process by solving a model-checking problem through a logical model, defined in Constraint LTL over clocks (CLTLoc) [5], that captures the semantics of the server behavior.

In this paper, we extend our previous contribution [3] in several ways. Firstly, we detail the software structure of the virtualized server, by focusing on the autonomic manager component: OLIVE. We enhance the context of the work by elaborating on the utility of this component: we describe how OLIVE is a proposal for the *fast verifier* component of the *Oma-cy* [15] architecture for distributed cyber-physical systems; and, moreover, we describe how the fast verifier component maps to the autonomic manager according to MAPE-K schema. Secondly, we elaborate on the model of the virtualized server and we provide a detailed description of its design, including also an extensive analysis of the logical model and of all its formulae. In addition, we enhance the experiments by providing the temporal cost of the execution of OLIVE and a detailed analysis of the results of the verification process.

Although a number of approaches exist with certain similarities to the concept of OLIVE (either named *resource managers* in operating systems or *autonomic managers* at user and system level), OLIVE is novel in several ways as it is designed for cyber-physical systems. Precisely, its novelties are:

- it is integrated into the Oma-cy reference middleware architecture that is targeted to distributed cyber-physical systems. Oma-cy requires the presence of fast verifier components to effectively develop distributed CPS applications. In this paper, we propose OLIVE as a solution for this;
- it uses formal models to control the evolution of a system in order to guarantee that it complies with the functional and non-functional specification;
- it allows the system to modify its model on-line;
- it integrates the temporal aspects of the system behavior, precisely the models are expressed in CLTLoc that allows to use clocks to reflect timing constraints;
- it is linked to actual model verification tools, precisely to CLTLoc verification;
- it has been validated in actual experiments and the timing cost has been obtained.

The paper is structured as follows. Section 2 describes related works. Section 3 describes the virtualized server architecture and the role of OLIVE in the context of MAPE-K adaptation strategy. Section 4 presents the formal engine executed by OLIVE; moreover, it presents the integration of formal verification logic (as OLIVE) with a reference architecture for CPS middleware. Section 5 presents the adaptive server model. Section 6 experimentally validates the approach in a realistic setting. Section 7 draws the conclusions and discusses the results. The appendixes complete the exposition with extra details and experimental data.

2. RELATED WORK.

The design and development of entities that manage the dynamic execution of a system and coordinate its transition to a different state is not a novel concept at all. Nevertheless, the requirements of the final systems have changed over time, ranging from the fairly static distributed environments of some decades ago to the highly dynamic and open concept of CPS that introduce time requirements as an essential part of their behavior. In this section, we explain different approaches to enable dynamic (on-line) adaptation using different techniques: autonomic computing, middleware, and adaptive resource management.

Most recent work on development of cyber-physical systems focuses on the pure distributed nature of such systems with the goal of providing reliable software infrastructures that ensure communication timeliness. An example of ensuring communication timeliness is by using virtualized servers [19] with schedulability of the communications to limit the time assigned to

DOI: 10.1002/smr

interactions. Other improvements for communication are also described in [20] that satisfy the temporal properties (guaranteed service time) by using the power of the underlying multicore; or [21] that modifies existing middleware platforms to more flexibly support larger numbers of communicating clients. Cyber-physical systems, however, require extensive improvement on the support to the evolution of the system model and on-line verification techniques.

As acknowledged by some authors [10], the architectural model-driven approach is one of the most comprehensive and widely accepted strategies to develop complex systems such as CPS that change over time. In a model driven approach, some form of model of the entire managed system is created by the autonomic manager, usually called *architectural model* that expresses its behavior, requirements, and goals. Changes are firstly planned and applied to the model to show the resulting state of the adaptation. If the new state of the system is acceptable, the plan can then be applied to the managed system. The architectural model can be used to verify that the system integrity is preserved when applying an adaptation. There are few contributions on this side mostly due to two factors. Firstly, the cost of the verification process may be too high to be suitable for most application domains. Secondly, the number of restrictions to be considered during the interval between the identification of the adaptation need and the adaptation transition itself can be high and complex to model. Some approaches such as [17] verify the temporal domain exclusively using simple utilization based schedulability analysis over distributed service based applications. Other approaches [27] provide design contracts explicitly including the timing aspects of the components behavior, but these only focus on the design of controllers for CPS.

MAPE-K loop [11] is among the most widely adopted schemes for adaptive systems. It stands for *monitor*, *analyze*, *plan*, *execute*, and *knowledge*, that identifies the set of phases for defining the evolution of a system's architectural model, embedding self-managing properties. Software reconfiguration schemes over MAPE-K for CPS using Petri nets have been initially explored in [18] as a means to evaluate the temporal behavior of a model depending on on-line input data. The differentiation between knowledge and planning in MAPE-K is considered fuzzy by some authors [10] since the information that constitutes the knowledge about the system may come from very distinct sources, e.g., logs of daily operation or human experts, among others. There are different methods to represent knowledge in autonomic systems [13]: (1) *utility* that is an indication of benefit for a system as a measure of a number of parameters; (2) *reinforcement learning* used for fine tunning adaptation policies or establishing new ones by observing the results of the management actions; it does not require a model of the system; (3) *probabilistic techniques* that are used to, e.g., select between algorithms to find the best solution. The *monitoring* part of MAPE-K collects data about the system behavior, that is needed to generate an informed adaptation decision. Monitoring is handled in [23] as heartbeats, an interface-based solution for applications to actively monitor and signal their progress levels with respect to user-defined performance goals. The gathered information can be exposed to an autonomic manager that will decide the subsequent adaptation actions.

Moreover, frameworks as [25] and [12] build self-adaptive distributed systems based on the concept of multi-agents from artificial intelligence; they use utility functions to establish the desired goals for controlling the interaction among autonomic elements that gather domain knowledge to perform reinforcement learning. This yielded the Unity framework that was evaluated in a data center scenario and applied in commercial products such as IBM WebSphere. The approach of [28] designs an autonomic element, termed as the combination of an autonomic manager and a managed element. The autonomic element adapts by applying learning adaptation policies that modify the behavior of the system by fine tunning the running algorithms. It relies on monitoring to collect data about the system behavior, and to fine tune the running algorithms in some way.

In large scale CPS deployments, nodes can achieve higher autonomy by including only the key software elements at deployment time together with a supporting intelligence that enables them to modify their software capacity/functionality once in operation. Nodes behaving in this manner can function as server nodes of other mobile CPS nodes that communicate with the servers using wireless networks. The benefits of virtualization technology are well known, and it directly supports the dynamic update of functionality in hostile deployments ([2], [31]); virtualization technology can

DOI: 10.1002/smr

be used in combination with hierarchical scheduling to assign fixed temporal partitions to virtual machines ensuring execution isolation. Servers will use virtualization technology to dynamically provide the required customized application to the client nodes, i.e, virtualization technology allows the dynamic update of virtual machines images containing specific functionality/services requested by the mobile clients. Currently, with the new technologies for image migration[†], it is possible to provide more efficient network latencies by bringing the computation close to the client.

The recent work *OMA-cy* [15] proposes an *Overarching middleware architecture for cyber-physical systems*. It is based on the classical *middleware* definition where three main layers are described: *infrastructure*, *distribution*, and *domain specific*. *OMA-cy* provides a modified *domain specific* layer that includes a *Fast Verifier* component as one of the specific *CPS functions*. Although it is acknowledged the need and mandatory location for an on-line verification logic inside CPS middleware that provides time efficient (and possibly bounded) adaptation, Oma-cy does not elaborate further into any specific logic for the *Fast Verifier* component. OLIVE autonomic manager is a proposal for such a logic.

3. DYNAMIC EXECUTION SUPPORT ARCHITECTURE

3.1. Virtualized server: architecture and adaptation strategy

The proposed architecture (Figure 1) has a central component named OLIVE (*On-LIne VErification manager*) that follows the principles of MAPE-K loop. OLIVE is an autonomic manager that handles the adaptation process derived from client requests by internally maintaining an updated model of the application logic. The adaptation requires the autonomic manager to build a *tentative new model* of the system and verify it on-line to determine if it conforms to the modified specification. If it does conform, the tentative model replaces the current one and becomes the actual current system model.

The architecture of the virtualized server node requires that the autonomic manager resides in a priviledged zone of the software stack as it arbitrates the execution of the system. OLIVE is located in the *virtual machine monitor* (the virtualization layer, where the decision process runs) and in the native operating system (*the host OS*, where the access to the system resources is priviledged). At that position, it has access to the buffer of incoming requests handled by the network protocol stack so that it can immediately run the decision process to determine the feasibility of an incoming request.

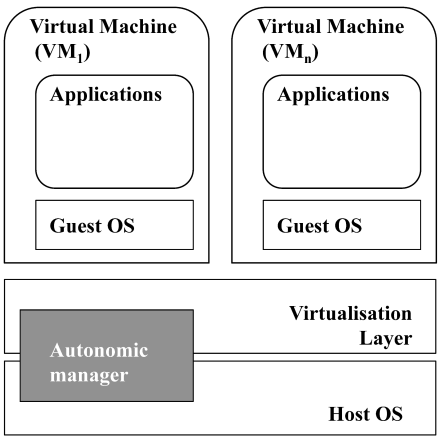


Figure 1. *Software design of an adaptive virtualized server that is integrated with OLIVE*

[†]Examples are IBM Informix, Windows Server, etc.

DOI: 10.1002/smr

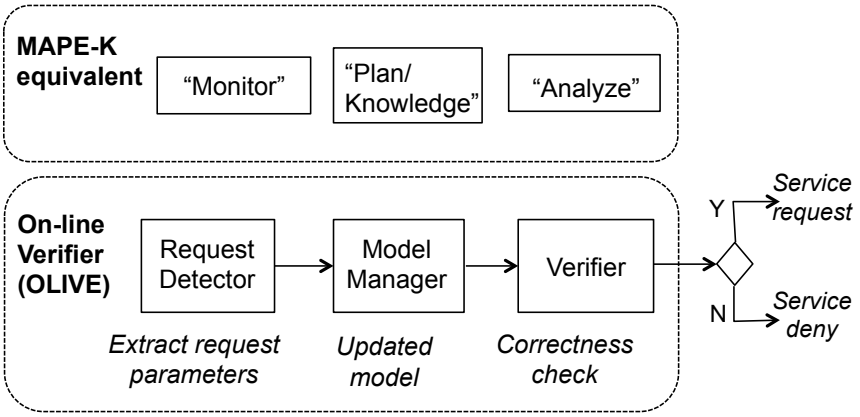


Figure 2. *Mapping of OLIVE components to MAPE-K entities.*

A specific realization of the MAPE-K phases into the OLIVE manager is proposed with the goal of achieving a time-bounded decision process as required by the real-time properties of CPS. Figure 2 shows the mapping between OLIVE and MAPE-K. The names of the units are shown in italics. The *Execute* module is not present as it is outside of the scope of this paper; a number of proposals for Execute algorithms are available in the literature, consisting of protocols for transitioning to the verified system model. Some proposed works are based on enforcing the execution of a selected model for either low-level software reconfigurations based on services [17], components [7], or mode change techniques [26].

The operation logic of the autonomic manager is the following. Client requests are detected through OLIVE's *Monitor* module. Upon a request, OLIVE runs the *Analyze* module to determine if the request can be served or not according to the tentative system model; the system model is contained in the *Knowledge* module. If the request can be served, the *Plan* module applies a strategy for the change and it is enforced through the *Execute* module. The *Analyze Module* of OLIVE has a submodule named on-line *Verifier* entity that has the objective of verifying the tentative future model during execution. The *Verifier* entity encapsulates the specific logic for verification. Therefore, if different formal techniques were to be used, only the *Verifier* and the *Knowledge* module will have to be changed; the rest of the architecture remains the same. The on-line verification logic should execute in *bounded time* in order to fully meet the requirements of CPS with respect to timely execution.

The two main activities of OLIVE in relation to MAPE-K loop are further detailed below.

Creation of a tentative future model. The *Request Detector* component captures requests from clients which could enforce new requirements. If so, the current software configuration of the server has to be modified causing an incremental model modification. The *Model Manager* modifies the current system model which is possibly enriched with the new properties as expressed in the request. The resulting model is called the *tentative future model*. This phase corresponds, in part, to the *Plan* phase of MAPE-K loop.

Execution-time verification *of the tentative future model*. The tentative future model undergoes validation by means of the *Verifier* entity. The verification result depends on the fulfillment of the specified utility criteria, e.g., the new restrictions introduced (or eliminated) by the request, called *incompatibilities*. The resulting verification time has a direct impact on the suitability for CPS domains given their inherent temporal requirements. This phase differs from the *design-time verification* phase in that the execution-time phase must be time-bounded. This depends on the specific tools and mechanisms employed; some may provide bounded-time results if a small set of changes are given in the future tentative model; others may yield to unacceptably large times. This phase maps, in part, to the *Plan* and *Knowledge* phases of MAPE-K loop. The refinement of the *creation of a tentative future model* with respect to MAPE-K is that a specific satisfiability check

DOI: 10.1002/smr

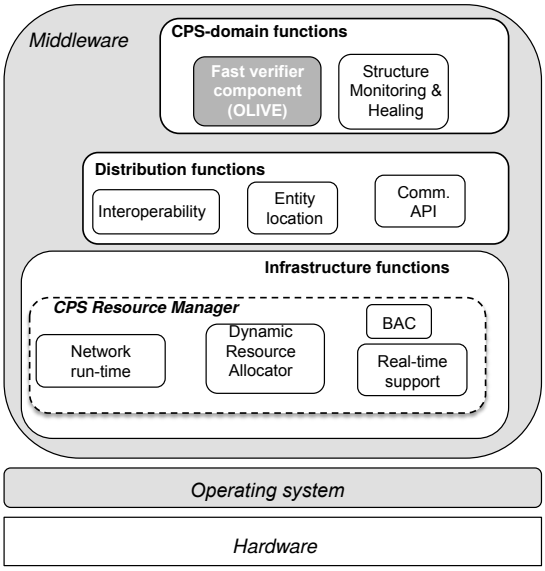


Figure 3. *The placement of the autonomic manager OLIVE inside the Fast Verifier component of Oma-cy middleware architecture*

is used. The tentative model may render unsatisfiable (no further action is taken) or satisfiable (the tentative model is applied and becomes the current model).

3.2. Integration of OLIVE into a CPS middleware

In a distributed environment such as a networked virtualized server, the role of communication middleware is essential. Middleware (in its original meaning [30]) refers to the software layer that enables the communication of remote processes and hides the complexity and heterogeneity of the underlying nodes' hardware and networking details. Oma-cy [15] digs into this concept and modifies it to suit the inherent properties of cyber-physical systems: rigorous design, timeliness, dynamic behavior and adaptation. In summary, the *domain specific* layer of middleware is transformed into a *CPS domain layer* that defines the needed components to suit the needs of cyber-physical systems in a distributed/networked domain. With respect to other domains, the fundamental difference of this layer is that it requires the presence of a *Fast Verifier* component. The role of the fast verifier is to embed the logic that supports the generation of modified models and the on-line verification of these models to ensure correctness at all times. In Oma-cy, the *CPS-domain functions* provide the basic support to the execution of CPS such as managing the structure of the system (e.g., number of nodes in the system and their interconnections), monitoring of the nodes performance and healing to recover from faults (e.g. nodes leave unexpectedly, server capacity is close to saturation, etc.).

CPS middleware require the integration of formal logic to check the future models generated during execution as an attempt of the system logic to adapt to the changing environment. Therefore, components such as OLIVE (in this case, based on CLTLoc) will be embedded inside this *Fast Verifier* component as shown in Figure 3.

4. FORMAL GROUND OF OLIVE

This section describes the formal modeling and validation engine of OLIVE. The formal tools used internally by OLIVE for rigorous model representation and validation relies on *satisfiability checking* of temporal logic formulae [29]. In the considered scenario, the server (system) is specified by a (temporal logic) formula defining its execution over time, instead of an operational model

DOI: 10.1002/smr

(like automata or transition systems) that is commonly adopted in the well-know model-checking approach. Verifying the satisfiability of the formula corresponds to determining whether there exists an execution of the server satisfying the specified behavior, i.e., find a possible allocation of computational resources in the server to satisfy the node request with a given requirement. When a temporal property is considered, the model-checking problem of the system with respect to the property is equivalent to verifying the entailment between the formula modeling the system and the formula representing the property. In this work, the manager uses CLTLoc with dense-time clocks to model both the server (an off-line model or a tentative on-line model) and the properties.

4.1. Basic of CLTL over clocks

Constraint LTL over clocks (CLTLoc) [5] is a semantic restriction of Constraint LTL (CLTL) [9] allowing atomic formulae over $(\mathbb{R}, \{<, =\})$ where the arithmetical variables behave like clocks of Timed Automata (TA) [1]. A clock x measures the time elapsed since the last time when $x = 0$, i.e., the last "reset" of x. Let V be a finite set of clock variables x over $\mathbb{R}$ and AP be a finite set of atomic propositions p. CLTLoc formulae are defined as follows:

$$\phi := p \mid x \sim c \mid \phi \wedge \phi \mid \neg\phi \mid \circ(\phi) \mid \bullet(\phi) \mid \phi \mathbf{U} \phi \mid \phi \mathbf{S} \phi$$

where $c \in \mathbb{N}$ and $\sim \in \{<, =\}$, $\bullet$, $\circ$, $\mathbf{U}$ and $\mathbf{S}$ are the usual "next", "previous", "until" and "since". An *interpretation* is a pair (π, σ), where $\sigma : \mathbb{N} \times V \to \mathbb{R}$ is a mapping associating every variable $x \in V$ and position in $\mathbb{N}$ with value $\sigma(i, x)$ and $\pi : \mathbb{N} \to \wp(AP)$ is a mapping associating each position in $\mathbb{N}$ with subset of AP. The difference between CLTL and CLTLoc derives from the following property. An interpretation (π, σ) is a CLTLoc interpretation if every clock progresses at the same rate, i.e., for every position $i \in \mathbb{N}$, there exists a "time delay" $\delta > 0$ such that, for any $x \in V$, the value of x in the next position $i + 1$ is either determined by the clock increment of δ time units, i.e., $\sigma(i + 1, x) = \sigma(i, x) + \delta$, or by the clock reset, i.e., $\sigma(i + 1, x) = 0$. The value $\sigma(i, x)$ of clock x at position i is the cumulative time elapsed over a finite set of positions of $\mathbb{N}$, consisting of all the consecutive adjacent positions from the last position where a reset $x = 0$ occurred until i. The initial value of a clock, $\sigma(0, x)$, may be any non-negative value. A clocks x might be initialized to c just by adding a constraint of the form $x = c$.

The semantics of CLTLoc is defined as for LTL, except for formulae $x \sim c$. At any position $i \in \mathbb{N}$, the truth value of $x \sim c$ is defined as $\sigma(i, x) \sim c$. Table I lists the semantics of CLTLoc given a CLTLoc interpretation (π, σ).

$$\begin{aligned}
(\pi, \sigma), i &\models p \Leftrightarrow p \in \pi(i) \text{ for } p \in AP \\
(\pi, \sigma), i &\models x \sim c \Leftrightarrow \sigma(i, x) \sim c \\
(\pi, \sigma), i &\models \neg\phi \Leftrightarrow (\pi, \sigma), i \not\models \phi \\
(\pi, \sigma), i &\models \phi \wedge \psi \Leftrightarrow (\pi, \sigma), i \models \phi \text{ and } (\pi, \sigma), i \models \psi \\
(\pi, \sigma), i &\models \circ(\phi) \Leftrightarrow (\pi, \sigma), i + 1 \models \phi \\
(\pi, \sigma), i &\models \bullet(\phi) \Leftrightarrow (\pi, \sigma), i - 1 \models \phi \wedge i > 0 \\
(\pi, \sigma), i &\models \phi \mathbf{U} \psi \Leftrightarrow \exists\, j \geq i : (\pi, \sigma), j \models \psi \ \wedge (\pi, \sigma), n \models \phi \ \forall\, i \leq n < j \\
(\pi, \sigma), i &\models \phi \mathbf{S} \psi \Leftrightarrow \exists\, j \leq i : (\pi, \sigma), j \models \psi \ \wedge (\pi, \sigma), n \models \phi \ \forall\, j < n \leq i
\end{aligned}$$

Table I. Semantics of CLTL

A formula ϕ is *satisfiable* if $(\pi, \sigma), 0 \models \phi$ for some (π, σ), called *trace* (or model).

CLTLoc is the first decidable extension of LTL that embeds the notion of dense time into explicit clocks occurring in the formulae and for which there is an implemented decision procedure freely available.

The standard technique to prove the satisfiability of CLTL and CLTLoc formulae is based on the construction of Büchi automata [5, 9]. However, for practical implementation [5], Bounded Satisfiability Checking (BSC) [29] is employed to avoid the onerous construction of automata. The

DOI: 10.1002/smr

possibly exponential growth of the size of the automaton derived from an LTL formula determines the worst case for CLTLoc. To limit the cost of the construction, a series of contributions targeting LTL were published to provide efficient ways to build automata from LTL formulae, like for instance, the method in [33]. On the other side, BSC is used to solve a satisfiability problem of LTL and CLTLoc with a different approach. Models of formulae are built in a similar way to the witness runs in the well-known Bounded model-checking technique [32]. The outcome of a BSC problem is either a bounded representation of infinite ultimately periodic model or unsat. Looking for a bounded representation of the models of a formula is equivalent to verifying the existence of looping paths in the automaton recognizing the language of the formula. [4, 5] show that BSC for CLTLoc is complete and that is reducible to a decidable Satisfiability Modulo Theory (SMT) problem. The results prove that any CLTLoc formula can be translated into the decidable theory of quantifier-free formulae with equality and uninterpreted functions combined with the theory of Reals $(\mathbb{R}, <)$. The translation preserves the satisfiability, i.e., the CLTLoc formula is satisfiable if, and only if, its translation is satisfiable. The BSC problem for CLTLoc is effectively solved by the tool ae^2zot [5].

OLIVE performs BSC of a formula representing (the model or a tentative model of) the virtualized server and the (temporal) requirements defined in the specification. Therefore, in this work, we specifically employ ae^2zot to implement the component of Fig. 2 called *Verifier*.

4.2. Property validation in OLIVE.

To model the server with CLTLoc, we adopt a finite set of atomic propositions that are used to represent the relevant configurations of the server. This way, we will identify the atoms that contribute to the definition of the possible states of the system, i.e., the server state, the external nodes requests and the discretized values of some quantitative measures, such as, for instance, the server load or the utility value (see Sect. 5). In propositional LTL-like formalisms, in fact, quantitative variables over infinite sets cannot, in general, be expressed with a finite number of formulae, unless a finite partition of the infinite domains is adopted. To this end, in Sect. 5, we provide suitable finite abstractions for the variables, with values over an infinite domain, that affect the adaptation process managed by OLIVE. Given a finite set D of values, abstracting a certain infinite domain, and a measure represented with a variable l over D, e.g., the server load, each element $d \in D$ is associated with l by means of a finite set of atomic propositions p_l^d. We write p_l^d to indicate that the value of l is d; then, when p_l^d holds we argue that $l = d$.

Descriptive specifications, such as those written in logic, are very useful in practice to model the semantics of systems that are inherently not convenient for being expressed with an operational formalism. Some example are, for instance, certain classes of non-functional requirements, the constraints describing the execution of processes based on a priority level or, in a light control system, the brightness of a lamp depending on the light intensity in a room. The presence of clocks in CLTLoc allows for writing an effective and intuitive definition of the system behavior over a dense time through logical formulae. As a matter of facts, CLTLoc enriches, with a metric time, what pure LTL can express only through an ordering of events. Despite the need for abstracting infinite domains, clock variables in CLTLoc ranging over $\mathbb{R}$ represent, without abstraction, physical (dense) clocks. Clocks appear in formulae of the form $x \sim c$ to express a bound c on the delay measured by clock x.

Specifying the behavior of a system requires to associate a finite set of clocks with all the system events (e.g., the system changes state from a certain state s to s'), that are relevant for the definition of the temporal semantics of the system. Given that a clock is reset when the associated event occurs and, in any moment, the clock value represents the time elapsed since the previous reset, then the value of clock x is the time elapsed since the last occurrence of the event which x is associated with.

We provide a small example to exemplify the approach we adopt in Sect. 5 to model OLIVE. Let us assume that a system controls a lamp which is either off or it can be active for 5 time units every time it is turned on by signal `switch_on`. Triggering the lamp on always precedes signal `switch_off` which enforces the lamp to switch to the off state. To specify this simple behavior, we can associate a clock x with signal `switch_on` to measure the duration of the activation of the

DOI: 10.1002/smr

lamp. The CLTLoc formula $(x = 0 \Leftrightarrow \texttt{switch\_on})$ constraints the reset $x = 0$ which occurs each time `switch_on` is triggered. To limit the duration of the lamp activation, the formula

$$\texttt{switch\_on} \Rightarrow \texttt{lamp\_on}\mathbf{U}(x = 5 \wedge \texttt{switch\_off})$$

imposes that `lamp_on` holds until a position where $(x = 5 \wedge \texttt{switch\_off})$ holds, that occurs exactly 5 time units after the occurrence of signal `switch_on`. We use similar constraints to define, for instance, the delay for delivering a service requested by an external node to the server.

All the *propositional variables* (atoms) and *clocks*, appearing in the CLTLoc model and specifying the virtualized server, contribute to the definition of the current configuration of the system. They are evaluated at run-time when the on-line engine is inquired to verify properties on the running systems. Upon a service request, OLIVE:

(1) extracts the set of predicates from the request and the current configuration of the system;
(2) generates an updated model, based on a set of CLTLoc formulae where the initial value of all the variables is set with the current configuration of the server;
(3) verifies the model by means of ae^2zot.

For instance, consider a simplified scenario where the server runs two virtual machines a and b that can execute the services s_1 and s_2 on demand. Assume that a running instance of OLIVE receives a request from node n with the following requirement: the node asks for service s_2 and the service must be completed before a certain deadline d; in addition, the execution of s_2 must conform to a specific requirement f. Moreover, assume the server configuration, at that moment, is such that the server is running service s_1 on virtual machine a and that s_1 was requested 3.2 time units ago by node n', with requirements f', and that service s_1 has being processed since 1 time unit. To verify a certain property, e.g., if there exists an execution where a request can be serviced, the current configuration is first extracted by the verification entity and then modeled through a CLTLoc formula of the form $\texttt{on}(n', s_1, a, f') \wedge rt^{f'}_{n',s_1} = 3.2 \wedge t^{n',s_1,a,f'}_{service} = 1 \wedge \texttt{re}(n, s_2, f)$, where $\texttt{on}(n', s_1, a, f')$ is the predicate which expresses that the system is currently serving on virtual machine a the request for a service s_1 issued by node n' with requirements f; clock $rt^{f'}_{n',s_1}$ measures the time elapsed since the instant where the request from n' for service s_1 was received; clock $t^{n',s_1,a,f'}_{service}$ measures the actual service time for service s_1 on virtual machine a; and, finally, $\texttt{re}(n, s_2, f)$ expresses the fact that a request for service s_2 issued by n with property f is sent to the manager. To carry out verification, the conjunction of the CLTLoc model of the server and the formula defining the current configuration along with the property are verified for satisfiability. The result (step (3)) is either SAT or UNSAT. In the current example, if the model is unsatisfiable then the request can not be served and the manager may reschedule it later, when enough resources will be available, or may adopt a new (tentative) model. Otherwise, the CLTLoc trace is a realistic execution of the system which satisfies the specification and that can be possibly executed at runtime.

The model used at step (3) can also be endowed with new formulae which may represent new requirements brought by service requests. For instance, OLIVE might need to verify if the system can deliver an already running service before a deadline that is different from the one specified in the request as new constraints have been injected into the system because of an adaption. In a full-logical framework, adding formulae to refine a specification is simple to integrate and it is just a matter of conjoining formulae. Alternatively, the CLTLoc specification can be used to perform off-line model-checking to design correct-by-construction monitors.

5. ADAPTIVE SERVER DESIGN

A virtualized server system has been designed integrating the OLIVE autonomic manager. We present a high level description of the implemented model, that is expressed in CLTLoc and fed to OLIVE. Later, the results of executing OLIVE component with the ae^2zot tool are presented that show the temporal cost of the on-line decision.

DOI: 10.1002/smr

5.1. Server description overview

The server runs two *virtual machines* (or VMs) (V_j); each VM can execute *services* (S_k) as per request of mobile nodes. Requests to the server have the form $(reqType, N_k, S_k, f, nf)$, where $reqType$ refers to the type of request performed (currently only $newnode$ is considered such that it indicates that a new node wants to request service); N_x refers to the identifier of the mobile node making the request; S_k is the requested service; f is the set of functional parameters that includes the *incompatibilities* (I); nf is the set of non functional parameters including the service *response-time deadline* (d_k), and the *priority* (p_k).

An example of new constraints regarding *incompatibility* issues (I) expressed in an adaptation request is: *I am node x requesting service S_k, and I can only execute in the environment of a virtualized server if there is an encryption service running (i.e., of type $Crypt$).*

A request, ($reqType$, N_x, S_k, f, nf), is expanded as shown in expression (1) in its functional (f into S_k, I) and non-functional (nf into p_k, d_k) parts. Also, the service type is indicated in the request as follows:

$$(reqType, N_x, S_k, p_k, d_k, I) \tag{1}$$

where S_k refers to the specific *requested* service; p_k is the *priority* of the mobile node at which it wants to be serviced; d_k is the maximum response time (*deadline*) for the completion of the response; I is the set of restrictions or incompatibilities imposed by the mobile node in relation to the server operation.

Table II shows the finite sets that represent the discretized value ranges for the model.

Table II. Boundaries and finite sets

Var	**Values**
Service state (Presence)	[Zero, Off, On]
VM no. (v)	$[V_a, V_b]$
VM state	[On, Off]
Service no. (s)	$[S_{a,1}, S_{a,2}]$ $[S_{b,1}, S_{b,2}]$
Service types	[User, Priviledged, Critical, Crypt]
Priority (p_k)	[Normal, High]
Deadline (d_k)	$[DT_1, DT_2, DT_3, DT_4]$
Utility (u_k)	[Low, Med, High]
Server load (increasing order) (l)	$[SL_0, SL_1, SL_2, SL_3, SL_4]$
VM load (l_i)	$[VL_1, VL_2, VL_3]$

The variables identified in Table II are described as follows. *Service states* refer to the operation status of the services: No operation ($Zero$), present but not active (Off), present and in running (On). *Number of virtual machines* is v, and the specific *virtual machines* in the server are V_a and V_b. The possible *states of a virtual machine* are On and Off. The *number of services* (s) is specified per virtual machine, where V_a has two services ($S_{a,1}, S_{a,2}$), and V_b has two services ($S_{b,1}, S_{b,2}$). *Service types* are in accordance to their criticality level ($User, Privileged, Critical, Cryt$). The type of service is used to model the incompatibilities (or restrictions) posed by the mobile CPS nodes. *Priority levels* (p_k) indicate the urgency of the requests; there are two priorities ($Normal, High$). *Service request deadlines* (d_k) are the maximum service time; node x expects an answer from the server d_k time units after issuing the request. The values are discretized in four ranges (DT_1 through DT_4). *Utility* (u_x) is an indication of the level of satisfaction of the mobile nodes as consequence of the server attention. The utility value provided by the server indicates a lower bound value expected by node x. The utility for a node is a function of the service time t_k and the level of satisfaction of its incompatibilities (represented by set I_k explained later in table VI. Three utility values are defined ($Low, Med, High$). *Server load* (l) indicates the used capacity of the server derived from the current services and virtual machine/s that are running. Five load value ranges are defined

DOI: 10.1002/smr

$(SL_0, SL_1, SL_2, SL_3, SL_4)$. *Virtual machine load* (l_i, where i refers to a specific virtual machine inside the server) indicates the used capacity of the server given its current execution of different services and virtual machines. The load value ranges are defines (VL_1, VL_2, VL_3).

Table III. Determination of the server load value (l)

Name	**Description**
Server load maximum (A on B on) (l^j affects d_k)	If $[(V_a.state = On) \wedge (V_b.state = On)] \wedge [(l_a = VL_3) \wedge (l_b = VL_3)]$ or $[(l_a = VL_3) \wedge (l_b = VL_2)]$ or $[(l_a = VL_2) \wedge (l_b = VL_3)]$ then $l = SL_4)$
Server load medium (A on B on)	If $[(V_a.state = On) \wedge (V_b.state = On)] \wedge [(l_a = VL_3) \wedge (l_b = VL_1)]$ or $[(l_a = VL_2) \wedge (l_b = VL_2)]$ or $[(l_a = VL_1) \wedge (l_b = VL_3)]$ then $l = SL_3)$
Server load medium (A on B off)	If $[(V_a.state = On) \wedge (V_b.state = Off)] \wedge (l_a = VL_3)]$ then $l = SL_2)$
Server load medium (A off B on)	If $[(V_a.state = Off) \wedge (V_b.state = On)] \wedge (l_b = VL_3)]$ then $l = SL_2)$
Server load low (A on B on)	If $[(V_a.state = On) \wedge (V_b.state = On)] \wedge [(l_a = VL_3) \wedge (l_b = VL_3)]$ or $[(l_a = VL_3) \wedge (l_b = VL_2)]$ or $[(l_a = VL_2) \wedge (l_b = VL_3)]$ then $l = SL_1)$
Server load low (A on B off)	If $[(V_a.state = On) \wedge (V_b.state = Off)] \wedge [(l_a = VL_1) \vee (l_a = VL_2)]]$ then $l = SL_1)$
Server load low (A off B on)	If $[(V_a.state = Off) \wedge (V_b.state = On)] \wedge [(l_b = VL_1) \vee (l_b = VL_2)]]$ then $l = SL_1)$
Server idle (A off B off)	If $[(V_a.state = Off) \wedge (V_b.state = Off)]$ then $l = SL_0)$

The determination of the load of a system is shown in table III. The server load is the sum of the partial utilizations (l_i) of all virtual machines of the server. In a real-time system, the utilization can be calculated under a periodic model that is compatible with a hierarchical scheduling technique for the virtual machines. As a result, $l_i = \frac{C_i}{A_i}$ where C_i is the computation time of V_i over an activation period A_i (named T_i in real-time scheduling). A virtual machine is assigned a temporal partition that is a maximum utilization value that ensures temporal isolation between virtual machines; the verification of the model checks that the overall utilization value is not above a specified threshold [26]. Other response time analysis methods would check if $t_k \leq d_k$ holds for all nodes, where t_k is the response time.

In the current model, the service time value (st_k) provided by the server that runs service k in response to a request from node x depends on the number of currently running services. This determines the server load (l) as a measure of resource availability. Additionally, the server load (l) (see Table III) depends on the sum of the individual load caused by each virtual machine on the server. Table IV models the load experimented by virtual machines (l_i) depending on the running services and their expected service time.

Table V expresses the set of incompatibilities that are mandatory conditions for the execution of the different types of services. Table VI shows the utility function to determine the benefit for the requests. It indicates the relation between the request parameters and the obtained response times (rt_k) and the level of fulfillment of incompatibilities (I). The set of threshold values for the response time $(rt = [RT_1, RT_2, RT_3, RT_4])$ allows to determine the server load values. We show the utility type for type *Critical*; similar analysis is integrated for types *Privileged* and *User*.

The utility function (f^u) is:

$$u_k = f^u(rt_k, satLevel_{I_k}) \tag{2}$$

DOI: 10.1002/smr

Table IV. Load of V_a (l_a) in relation to the execution characteristics of the services. l_b is not expressed since the pattern is repeated

Name	**Description**
$l_a = VL_3$ when $[(V_a.state = On)]$	If $[(S_{a,1}.state = On) \wedge (S_{a,2}.state = On)] \wedge [((S_{a,1}.d_k = DT_2) \vee (S_{a,1}.d_k = DT_1)) \wedge ((S_{a,2}.d_k = DT_2) \vee (S_{a,2}.d_k = DT_1))]$ then $l_a = VL_3$
$l_a = VL_2$ when $[(V_a.state = On)]$	If $[(S_{a,1}.state = On) \wedge (S_{a,2}.state = On)] \wedge [(S_{a,1}.d_k = DT_3) \wedge (S_{a,2}.d_k = DT_3)]$ then $l_a = VL_2$ If $[(S_{a,1}.state = On) \wedge (S_{a,2}.state = Off)] \wedge (S_{a,1}.d_k = DT_1)]$ then $l_a = VL_2$ If $[(S_{a,1}.state = Off) \wedge (S_{a,2}.state = On)] \wedge (S_{a,2}.d_k = DT_1)]$ then $l_a = VL_2$
$l_a = VL_1$ when $[(V_a.state = On)]$	If $[(S_{a,1}.state = On) \wedge (S_{a,2}.state = Off)] \wedge (S_{a,1}.d_k = DT_4)]$ then $l_a = VL_1$
$l_a = VL_1$ when $[(V_a.state = On)]$	If $[(S_{a,1}.state = Off) \wedge (S_{a,2}.state = On)] \wedge (S_{a,2}.d_k = DT_4)]$ then $l_a = VL_1$

Table V. Incompatibility set

Incompatibilities for type $Critical$
If $servType = Critical$ then there is no $Privileged$ service in the server (in any VM) If $servType = Critical$ then there is no $User$ service in the same VM If $servType = Critical$ then if there are other services in the same VM, these are necessarily of type $CRYPT$

Table VI. Utility function

Name	**Description**
Utility (u_k) for type $Critical$	If $[(p_k = High) \wedge (rt_k \leq RT_1)]$ then $[u_k = High]$ If $[(p_k = High) \wedge [(RT_1 < rt_k \leq RT_3)]]$ then $[u_k = Normal]$ If $[(p_k = High) \wedge (rt_k > RT_3)]$ then $[u_k = Low]$
Utility (u_k) for type $Priviledged$	If $[(p_k = Normal) \wedge [(rt_k \leq RT_2)]$ then $[u_k = High]$ If $[(p_k = Normal) \wedge (RT_2 < rt_k < RT_4)]$ then $[u_k = Normal]$ If $[(p_k = Normal) \wedge (rt_k \geq RT_4)]$ then $[u_k = Low]$
Utility (u_k) for type $User$	If $[(p_k = High) \wedge (rt_k \leq RT_2)]$ then $[u_k = High]$ If $[(p_k = Normal) \wedge (RT_2 < rt_k < RT_4)]$ then $[u_k = Normal]$ If $[(p_k = Normal) \wedge (rt_k \geq RT_4)]$ then $[u_k = Low]$

where $satLevel_{l_k}$ is the degree of satisfiability of the constrains or incompatibilities (I_k). If all the constraints brought in by the node request are satisfied, the value of $satLevel_{l_k}$ is maximum. If not, a specific verification method is used and a *true/false* answer is obtained.

DOI: 10.1002/smr

The utility function determining the satisfiability of a request is directly derived from the expression (2). The utility is discretized, in accordance to the service types, into acceptable ($High$ or $Normal$) and unacceptable (Low) levels.

The provided scenario is set up to observe the temporal cost of our OLIVE component. In an actual scenario, the values assigned to ST in relation to D and RT are separated by a few orders of magnitude (for a desired response time of $450\ ms$ a typical execution would be $60\ ms$). The verification time for this situation is application dependent; e.g., for the case of an object tracking video analysis, up to one second would be acceptable; in the case of mobile nodes joining a new system to replace existing functionality, some minutes are tolerated.

5.2. Encoding CLTLoc model into OLIVE.

Let W be set $\{\mathrm{N}, \mathrm{H}\}$ of priorities (or weights), D be set $\{\mathrm{DT}_1, \mathrm{DT}_2, \mathrm{DT}_3, \mathrm{DT}_4\}$ of deadlines and T be set $\{\mathrm{U}, \mathrm{P}, \mathrm{C}, \mathrm{Y}\}$ of service types, where U represents "User", P represents "Priviledge", C represents "Critical" and Y represents "Crypt". Let V be a finite set of virtual machines, S be a finite set of services and N be a finite set of nodes and $F \subseteq W \times D \times T$. A *non-functional property* (or simply *feature*) $f \in F$ is a triple (w, d, t), specifying a priority, a deadline and a service type, which labels a request of a service by a node. The definition of set F of features is application specific.

The design of the CLTLoc model considers the following assumptions:

(*i*) The process of requesting and obtaining a service by a node, is unique during the time interval along which the service is executed. In fact, we may identify a node by its name and an information distinguishing the instance (of that node) that provides the request.

(*ii*) Between a request and the service delivery, there is only one virtual machine on the server which executes the service that is required by a node. In realistic scenarios, the number of requests of a node is finite and the value defining this bound can be assumed as a parameter of the server.

(*iii*) The server always guarantees the termination of the task associated with a service request as the system may be equipped with a scheduler and a *resume-after-failure* mechanism implementing reliable computations.

In our model, variables n, d, w and f range over N, D, W and F; and variables s and v range over S and V, respectively. The following propositions model events in the system. $\texttt{re}(n, s, f)$: node n requests service s with feature f. $\texttt{de}(n, s, f)$: service s, requested by node n with feature f, is delivered. $\texttt{on}(n, s, v, f)$: service s, requested by node n with feature $f \in F$, is active on VM v.

The CLTLoc model captures the real-time behavior of the server from the moment when it receives a service request to the moment of the delivery, as depicted in Figure 4. The process starts when a node n issues a request with a *request message* $\texttt{re}(n, s, f)$ which specifies the service s and the requirement to execute the task on the server, through the feature f. The request is processed immediately by the system which executes s as soon as there are enough resources for the service to be executed, i.e., when possible, the system activates a process, running the service, on virtual machine v. At this moment, $\texttt{on}(n, s, v, f)$ becomes true and it remains true as long as the process terminates, where $\texttt{on}(n, s, v, f)$ becomes false. State $\texttt{on}(n, s, v, f)$ always changes from false to true after event $\texttt{re}(n, s, f)$ as the time elapsed between the request and the start of the service represents the overhead to initiate the process itself on the selected virtual machine and the time needed to schedule the tasks. When the process terminates, the server starts releasing the allocated resources and later notifies node n of the termination of the service by sending a *delivery message* $\texttt{de}(n, s, f)$. A service request is always associated with a specific temporal constraint defining the maximum tolerated delay for the service delivery which is specified, at the moment of the request, in f through a deadline d. To measure the total time elapsed between a service request and the service delivery, the server allocates a clock $rt^f_{n,s}$. Although the server guarantees the termination of the processes, possible failures and scheduling delay, that are enforced by the system to comply with incompatibilities and availability of the computational resources, may delay the service execution. The total time required to complete the process determines the utility for the user. The value of clock

DOI: 10.1002/smr

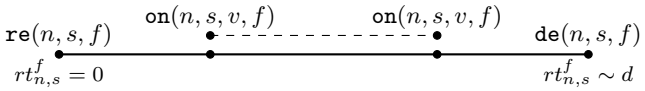


Figure 4. Events for a node n, service s, virtual machine v and features f. The dashed line represents the interval where $\texttt{on}(n, s, v, f)$ holds.

$rt^f_{n,s}$ at the service delivery determines the level of utility that the server actually provisions a node waiting for a service.

5.2.1. Event ordering. Specific formulae model the sequences of events of Figure 4 by imposing the following:

(*i*) a service request $\texttt{re}(n, s, f)$ always precedes its delivery $\texttt{de}(n, s, f)$ and a service delivery $\texttt{de}(n, s, f)$ always follows its request $\texttt{re}(n, s, f)$;
(*ii*) the server initiates a thread to run the service, between the request and the delivery. Hence, when a service request is issued, the service will eventually start on some virtual machine v ($\texttt{on}(n, s, v, f)$ becomes true, before the service delivery) and, moreover, when a service delivery occurs, the service has been started on some virtual machine v ($\texttt{on}(n, s, v, f)$ was true in the past, after the service request);
(*iii*) the necessary condition for a service to be executed on v.

Formula (3) fulfills requirement *(i)* and states that a service request $\texttt{re}(n, s, f)$ always precedes its delivery $\texttt{de}(n, s, f)$, and a service delivery $\texttt{de}(n, s, f)$ always follows its request $\texttt{re}(n, s, f)$. The first formula states that, from the next position of the one where $\texttt{re}(n, s, f)$ occurs, $\texttt{re}(n, s, f)$ does not hold until the occurrence of $\texttt{de}(n, s, f)$. The second formula imposes that, when $\texttt{de}(n, s, f)$ occurs, at some position, then either that position is the origin or from the previous position back to the one where $\texttt{re}(n, s, f)$ occurs, $\texttt{de}(n, s, f)$ does not hold. To distinguish the origin, we introduce $\texttt{0}$ as an abbreviation for formula $\neg \bullet (true)$ that holds only at position 0. From now on, we intend n, d, w and f as variables over N, D, W and F; and variables s and v as variables over S and V, respectively.

$$\bigwedge_{n,s,f} \Big(\texttt{re}(n, s, f) \Rightarrow \circ \left(\neg \texttt{re}(n, s, f) \, \mathbf{U} \, \texttt{de}(n, s, f) \right) \Big) \quad \wedge \quad \bigwedge_{n,s,f} \Big(\texttt{de}(n, s, f) \Rightarrow \texttt{0} \vee \bullet \left(\neg \texttt{de}(n, s, f) \, \mathbf{S} \, (\texttt{re}(n, s, f) \vee \texttt{0}) \right) \Big) \tag{3}$$

Formula (4) states that between a request and the delivery, the server initiates a thread to run the service, therefore meeting requirement *(ii)*. The thread is executed by a virtual machine having enough resources and satisfies all the incompatibilities among the tasks in execution (that are modeled later). The first formula of (4) imposes that, when a service request is issued, the service will eventually start on some virtual machine v ($\texttt{on}(n, s, v, f)$ becomes true before the service delivery). It states that, from the next position of the one where $\texttt{re}(n, s, f)$ occurs, before the occurrence of $\texttt{de}(n, s, f)$ the service eventually starts $\texttt{on}(n, s, v, f)$. The second part of the formula states that when a service delivery occurs, the service has been started on some virtual machine v ($\texttt{on}(n, s, v, f)$ was true in the past, after the service request). It imposes that, when $\texttt{de}(n, s, f)$ occurs, at some position, then either that position is the origin or from the previous position back to the one where $\texttt{on}(n, s, v, f)$ was true, no request $\texttt{re}(n, s, f)$ occurred.

$$\bigwedge_{n,s,f} \Big(\texttt{re}(n, s, f) \Rightarrow \neg \texttt{de}(n, s, f) \, \mathbf{U} \bigvee_{v} \texttt{on}(n, s, v, f) \Big) \quad \wedge \quad \bigwedge_{n,s,f} \Big(\texttt{de}(n, s, f) \Rightarrow \neg \texttt{re}(n, s, f) \, \mathbf{S} \, (\bigvee_{v} \texttt{on}(n, s, v, f) \vee \texttt{0}) \Big) \tag{4}$$

DOI: 10.1002/smr

The following two formulae, (5) and (6), complete the fulfillment of requirement *(iii)*. Formula (5) defines the necessary condition for a service to be currently executed on a server v, i.e., only after an event $\texttt{re}(n,s,f)$ and before $\texttt{de}(n,s,f)$. The formula states that if $\texttt{on}(n,s,v,f)$ holds at the current position then (i) no request $\texttt{re}(n,s,f)$ nor answer $\texttt{de}(n,s,f)$ may occur and (ii) no $\texttt{re}(n,s,f)$ can occur until the next event $\texttt{de}(n,s,f)$ and, since the previous $\texttt{re}(n,s,f)$, no $\texttt{de}(n,s,f)$ can occur while no other VMs can execute the service simultaneously. Moreover, when $\texttt{on}(n,s,v,f)$ holds no VM $v' \neq v$ can execute, at the same time, service s. In other words, Formula (5) impedes the system from concurrent execution of tasks serving node n requiring service s on virtual machine v. Given a triple (n,s,f) at most one active thread can execute the service and there is only one v such that $\texttt{on}(n,s,v,f)$ holds.

$$\bigwedge_{n,s,v,f} \Big(\texttt{on}(n,s,v,f) \Rightarrow \begin{pmatrix} \neg\texttt{re}(n,s,f)\,\mathbf{U}\,(\texttt{de}(n,s,f) \wedge \neg\texttt{on}(n,s,v,f)) \\ \wedge \\ \neg\texttt{de}(n,s,f)\,\mathbf{S}\,((\texttt{re}(n,s,f) \vee \texttt{0}) \wedge \neg\texttt{on}(n,s,v,f)) \end{pmatrix} \wedge \neg \bigvee_{v' \neq v} \texttt{on}(n,s,v',f)\Big) \tag{5}$$

Formula (6) constraints the behavior of active services on the server. We assume that an active execution can not be preempted. Therefore, when a service has been completed, i.e., the thread computing it is terminated and $\texttt{on}(n,s,v,f)$ does not hold, then the computation can not be restarted until the delivery of the service which occur when $\texttt{de}(n,s,f)$ holds.

$$\bigwedge_{n,s,v,f} \Big((\neg\texttt{on}(n,s,v,f) \wedge \bullet\,(\texttt{on}(n,s,v,f)) \Rightarrow \neg\texttt{on}(n,s,v,f)\,\mathbf{U}\,\texttt{de}(n,s,f)\Big) \tag{6}$$

5.2.2. Timing constraints. To measure the time elapsed between the request $\texttt{re}(n,s,f)$ and answer $\texttt{de}(n,s,f)$, we introduce a clock $rt^f_{n,s}$, for any triple n,s,f. The value of $rt^f_{n,s}$ at the moment of a service delivery, i.e., when $\texttt{de}(n,s,f)$ holds, determines the level of utility for the node requesting the service (see later the formula modeling the utility). Formula (7) imposes that clock $rt^f_{n,s}$ is reset when a request for a service is issued.

$$\bigwedge_{n,s,f} (\texttt{re}(n,s,f) \Leftrightarrow rt^f_{n,s} = 0) \tag{7}$$

To model the system elapsed time, we define how the server load affects the total duration of the running services in the server. The speed of computation of the VMs depends on the server load that is, in turn, influenced by the running tasks. The model captures the relation between the load and the duration by dividing the computation into phases which are intervals over the time with constant load. Each phase has a specific duration, determined by the load. Intuitively, the model captures how fast, or slow, the server is in the current phase on the basis of the current load, i.e., the higher the load, the slower the computation of the server. The amount of time to complete a phase is defined at design time or can be estimated by monitoring a deployed system (Table III shows the conditions to have the maximum server load). To measure the duration of the phases, we use a pair of clocks t^0_{server} and t^1_{server} which are alternatively reset when the server load varies. In any position, the *active clock* measuring the current phase is the last one reset in the past. The fact that we need two clocks is rather a technical aspect related to the semantics of CLTLoc logic. In CLTLoc, in fact, a clock can not be evaluated (i.e., its value is used to evaluate the truth value of arithmetical constraints like $x \sim c$, where x is a clock and c a constant) and, at the same time, reset to value 0. To do simultaneously a reset and the evaluation we use two clocks. One stores the value of time which is used to verify time bounds and the other is reset. The active clock to be used in formulae is the one with non-null evaluation. The two clocks alternately reset by means of the next Formula (8).

$$\begin{gathered} t^0_{server} = 0 \Rightarrow \circ\left((t^1_{server} = 0)\mathbf{R}(t^0_{server} > 0)\right) \wedge \neg(t^1_{server} = 0) \\ \wedge \\ t^1_{server} = 0 \Rightarrow \circ\left((t^0_{server} = 0)\mathbf{R}(t^1_{server} > 0)\right) \end{gathered} \tag{8}$$

DOI: 10.1002/smr

The initial configuration is set by $t^0_{server} = 0$. We use the shorthand $t_{server} \sim c$ to indicate the formula

$$t^0_{server} > 0 \wedge (t^1_{server} = 0 \vee (t^1_{server} > 0)\mathbf{S}(t^0_{server} = 0)) \Rightarrow t^0_{server} \sim c$$
$$\wedge$$
$$t^1_{server} > 0 \wedge (t^0_{server} = 0 \vee (t^0_{server} > 0)\mathbf{S}(t^1_{server} = 0)) \Rightarrow t^1_{server} \sim c$$

defining which clock keeps the measure of the time elapsed between the current position and the last clock reset, i.e., the valid value of time since the last reset. We use $t_{server} = 0$ to indicate the formula $(t^0_{server} = 0) \vee (t^1_{server} = 0)$. Formula (9) determines the time delay which each server load is associated with. It states that if the server load is $\texttt{sv\_l}(i)$, then $\texttt{sv\_l}(i)$ holds until the moment where the amount of time required by the server to complete the current phase is expired. The time elapsing associated with $\texttt{sv\_l}(i)$ is defined by a specific constraint on t_{server} as specified in Table XVII (third row). The first formula states that one of the two clocks t^i_{server} is reset whenever the server load changes.

$$\begin{array}{c} t_{server} = 0 \Leftrightarrow \bigvee_{i\in\{0,4\}} (\texttt{sv\_l}(i) \wedge \neg \bullet (\texttt{sv\_l}(i))) \\ \wedge \\ \texttt{sv\_l}(4) \Rightarrow \texttt{sv\_l}(4)\mathbf{U}(t_{server} \geq ST_4 \wedge \neg\texttt{sv\_l}(4)) \\ \wedge \\ \texttt{sv\_l}(3) \Rightarrow \texttt{sv\_l}(3)\mathbf{U}(ST_3 \leq t_{server} < ST_4 \wedge \neg\texttt{sv\_l}(3)) \\ \wedge \\ \texttt{sv\_l}(2) \Rightarrow \texttt{sv\_l}(2)\mathbf{U}(ST_2 \leq t_{server} < ST_3 \wedge \neg\texttt{sv\_l}(2)) \\ \wedge \\ \texttt{sv\_l}(1) \Rightarrow \texttt{sv\_l}(1)\mathbf{U}(ST_1 \leq t_{server} < ST_2 \wedge \neg\texttt{sv\_l}(1)) \end{array} \tag{9}$$

To model the delay affecting the computation of services on the basis of the server load, Formula (10) imposes that a running service, i.e., such that $\texttt{on}(n, s, v, f)$ holds, can not terminate before the end of a phase and also that a service can not begin after the start of a phase. Formula (10) states that if a service is active, and the server load is $\texttt{sv\_l}(i)$, then it has been active since the beginning of the phase (where $t^i_{server} = 0$, for some $i \in \{0, 1\}$) and it will be active until the end of the phase (where $t^{i+_2 1}_{server} = 0$, for some $i \in \{0, 1\}$) including the current position and all adjacent positions where $\texttt{sv\_l}(i)$ holds.

$$\bigwedge_{i\in\{0,1\}} \bigwedge_{n,s,v,f} \left(\left(\texttt{on}(n, s, v, f) \wedge \begin{pmatrix} t^i_{server} > 0 \\ \wedge \\ t^{i+_2 1}_{server} > 0 \end{pmatrix} \mathbf{S}(t^i_{server} = 0)\right) \Rightarrow \right.$$
$$\left. \texttt{on}(n, s, v, f)\mathbf{S} \begin{pmatrix} \texttt{on}(n, s, v, f) \\ \wedge \\ t^i_{server} = 0 \end{pmatrix} \wedge \texttt{on}(n, s, v, f)\mathbf{U}(t^{i+_2 1}_{server} = 0)\right) \tag{10}$$

To measure the duration of the execution of a running service we introduce a set of clock $t^{n,s,f}_{service}$ storing the time elapsed since the position where the execution of s, requested by n, started on virtual machine v. Formula (11) imposes that clock $t^{n,s,f}_{service}$ is reset when the execution of the service starts, i.e., when $\texttt{start}(n, s, s, f)$ becomes true, for some v.

$$\bigwedge_{n,s,f} \begin{pmatrix} t^{n,s,f}_{service} = 0 \Rightarrow \bigvee_v (\texttt{on}(n, s, v, f) \wedge \neg \bullet (\texttt{on}(n, s, v, f))) \\ \wedge \\ \neg\texttt{0} \wedge \bigvee_v (\texttt{on}(n, s, v, f) \neg \bullet (\texttt{on}(n, s, v, f))) \Rightarrow t^{n,s,f}_{service} = 0 \end{pmatrix} \tag{11}$$

DOI: 10.1002/smr

The model defines the total duration of a service in terms of the *smallest computation*. For each tuple (n, s, v, f) we introduce a clock $t_{service}^{n,s,v,f}$ which is always reset when s, requested by n, starts the execution on some v, i.e., when $\texttt{on}(n, s, v, f)$ becomes true for some v. Formula (12) constraints the duration of $\texttt{on}(n, s, v, f)$ which remains true for at least the minimum time required to complete the service, that is, until the position where $t_{service}^{n,s,v,f}$ is greater than or equal to $k(s, v)$ times the duration ST_1 of the smallest phase, for some positive value $k(s, v)$ depending on the service s and the VM v. The minimum time $k(s, v) \cdot ST_1$ is the time that the server needs to run the service in an empty machine (i.e., when it is the only service in the system). The value $k(s, v)$ abstracts the *computational power* of v and provides and estimation of the *cost of executing service* s on v. Its value can be obtained by monitoring the system or by design assumptions.

$$\begin{gathered}\texttt{on}(n, s, v, f) \Rightarrow \\ \texttt{on}(n, s, v, f)\mathbf{U}(\texttt{on}(n, s, v, f) \wedge t_{service}^{n,s,v,f} \geq k(s, v) \cdot ST_1)\end{gathered} \tag{12}$$

5.2.3. Server load and Virtual machine load. We introduce the following propositions: $\texttt{user}(s, v)$, $\texttt{priviledge}(s, v)$, and $\texttt{critic}(s, v)$ for defining incompatibilities. If $\texttt{user}(s, v)$ holds, then, at that position, there is an active thread executing a service of type "User" in v. Propositions $\texttt{priviledge}(s, v)$ and $\texttt{critic}(s, v)$ have a similar meaning but for "Priviledge" and "Critical" services. "Crypt" services are modeled by proposition $\texttt{crypt}(s)$, with $s \in S$. Formula (13) defines incompatibilities among the services that are executed on the server. The first part defines when each proposition holds based on which service is currently running in the server. For example, $\texttt{user}(s, v)$ holds for some service s and virtual machine v if, and only if, there is a running instance serving s on v which was requested by some node n with feature f, so that $\texttt{on}(n, s, v, f)$ holds. The last formula draws a necessary condition for a service s to be "Crypt" by imposing that it is "Critical" and there exists a virtual machine v that executes it.

$$\bigwedge_{s,v} \left(\begin{array}{l} \Big(\bigvee_{f=(p,d,U)}^{n,p,d} \texttt{on}(n, s, v, f) \Big) \Leftrightarrow \texttt{user}(s, v) \wedge \\ \Big(\bigvee_{f=(p,d,P)}^{n,p,d} \texttt{on}(n, s, v, f) \Big) \Leftrightarrow \texttt{priviledge}(s, v) \wedge \\ \Big(\bigvee_{f=(p,d,C)}^{n,p,d} \texttt{on}(n, s, v, f) \Big) \Leftrightarrow \texttt{critic}(s, v) \end{array} \right) \wedge \bigwedge_{s} (\texttt{crypt}(s) \Rightarrow \bigvee_{v} \texttt{critic}(s, v)) \tag{13}$$

We model incompatibilities of Table XVI by Formula (14). The first part states that any critical service can not coexist with a priviledged one in the server (in any VM) and any critical service can not execute in a VM with a user service. If $\texttt{critic}(s, v)$ holds, then, in any virtual machine $v' \in V$ and for any service $s' \in S$, no service of type "Priviledge" can be active and, at the same time, in the same virtual machine v and for any service $s' \in S$, no service of type "User" can be active. The second part imposes that critical services can coexist in the same VM only with "Crypt" services.

$$\begin{gathered}\bigwedge_{v,s} \Big(\texttt{critic}(s, v) \Rightarrow \bigwedge_{v' \in V, s' \in S} \neg\texttt{priviledge}(s', v') \wedge \bigwedge_{s' \in S} \neg\texttt{user}(s, v) \Big) \\ \wedge \\ \bigwedge_{\substack{s,s',v \\ s \neq s'}} \Big(\texttt{critic}(s, v) \wedge \texttt{critic}(s', v) \Rightarrow \texttt{crypt}(s) \wedge \texttt{crypt}(s') \Big)\end{gathered} \tag{14}$$

Modeling the server load and virtual machine load requires the definition of a finite abstraction of their value. In Table XIV, we have already defined a suitable finite set of values so that they can

DOI: 10.1002/smr

be represented through CLTLoc formulae. We introduce the following propositions that are indexed with i defined over a finite set of values and we explain their meaning to understand the following formulae.

- $\texttt{vm\_l}(v, i)$: represents that virtual machine v is running with load value equal to i, where i is in $\{1, 2, 3\}$.
- $\texttt{sv\_l}(i)$: models server load whose value is equal to i, where i is in $\{0, \ldots, 4\}$, when $\texttt{sv\_l}(i)$ holds.

We also introduce the following shorthand which are helpful for defining the value of $\texttt{sv\_l}(i)$.

- $\texttt{vm\_on}(s, v)$: abbreviates $\bigvee_{n,f} \texttt{on}(n, s, v, f)$ and captures the fact that in virtual machine v service s is running.
- $\texttt{vm\_on}(v)$: abbreviates $\bigvee_{s} \texttt{vm\_on}(s, v)$ and represents that there is a service running on v.

The formula defining $\texttt{sv\_l}(i)$, for each i in the set abstracting the load value, establishes the relation between $\texttt{sv\_l}(i)$ and a specific condition on both $\texttt{vm\_on}(s, v)$ and $\texttt{vm\_l}(v, i)$ based on the correspondent case in Table XIV. The complete encoding of each $\texttt{sv\_l}(i)$ is provided in the appendix B and the following Formula (15) shows, as an example, only the case for $\texttt{sv\_l}(0)$ and $\texttt{sv\_l}(1)$. The condition defining $\texttt{sv\_l}(0)$ is straightforward whereas the one for $\texttt{sv\_l}(0)$ is more complex as it captures many cases. The server load is 0 when both the virtual machines are idle and it is 1 when one of the three possible situation modeled in the left hand-side of the double implication holds. In the first two cases, one of the two virtual machine is idle while the other is running with low load (either 1 or 2). In the third case both the virtual machines are working and at most one load value of the two is at most 2.

$$\begin{array}{c} \neg\texttt{vm\_on}(a) \wedge \neg\texttt{vm\_on}(b) \Leftrightarrow \texttt{sv\_l}(0) \\ \wedge \\ \left(\begin{array}{c} \texttt{vm\_on}(a) \wedge \neg\texttt{vm\_on}(b) \wedge (\texttt{vm\_l}(a,1) \ \vee \ \texttt{vm\_l}(a,2)) \\ \vee \\ \neg\texttt{vm\_on}(a) \wedge \texttt{vm\_on}(b) \wedge (\texttt{vm\_l}(b,1) \ \vee \ \texttt{vm\_l}(b,2)) \\ \vee \\ \texttt{vm\_on}(a) \wedge \texttt{vm\_on}(b) \wedge \left(\begin{array}{c} \texttt{vm\_l}(a,1) \wedge \texttt{vm\_l}(b,1) \vee \\ \texttt{vm\_l}(a,2) \wedge \texttt{vm\_l}(b,1) \vee \\ \texttt{vm\_l}(a,1) \wedge \texttt{vm\_l}(b,2) \end{array} \right) \\ \wedge \end{array} \right) \Leftrightarrow \texttt{sv\_l}(1) \end{array} \tag{15}$$

Formula (15) employs the value of $\texttt{vm\_l}(v, i)$, whose definition is given in Formula (16) which translates the rules of Table XV. The definition is given on the basis of parameter v, which is either a or b in our particular implementation of the system considering only two virtual machines. Formula (16) consists of three cases as predicate $\texttt{vm\_l}(v, 2)$, the one which is not shown, is derived as a consequence of the definition of $\texttt{vm\_l}(v, 0)$ $\texttt{vm\_l}(v, 1)$ and $\texttt{vm\_l}(v, 3)$ and the fact that at each position there is at least one predicate $\texttt{vm\_l}(v, l)$ which holds. The first conjunct imposes this requirement, i.e., that the server load value is defined in each time position or, in other words, that there is at least a load value l such that $\texttt{vm\_l}(v, l)$ holds. The second disjunct is needed to restrict the previous one and imposes the uniqueness of the server load value. It states that if $\texttt{vm\_l}(v, l)$ holds, for some l, then $\texttt{vm\_l}(v, l')$ is false for all the l' different from l and belonging to $\{0, \ldots, 3\}$. The third part specifies the definition of $\texttt{vm\_l}(v, l)$ for $l = 0$, $l = 1$ and $l = 3$. Each value $\texttt{vm\_l}(v, l)$ depends on the property of the service running on v which is captured by the feature f characterizing the proposition $\texttt{on}(n, s, v, f)$ which holds in that moment. We consider the following abbreviations in the formulae: $\psi_2(f) := (d \leq \mathrm{DT}_2)$ and $\psi_4(f) := (d = \mathrm{DT}_4)$. For example, $\psi_2(f)$ is true if, in

DOI: 10.1002/smr

the feature f, deadline d is $\leq \mathrm{DT}_2$.

$$
\begin{aligned}
&\bigvee_{l\in 0,\ldots,3} \texttt{vm\_l}(a,l) \\
&\wedge \\
&\bigwedge_{l\in 0,\ldots,3} \big(\texttt{vm\_l}(a,l) \Rightarrow \neg \bigvee_{l'\in 0,\ldots,3, l\neq l'} \texttt{vm\_l}(a,l')\big) \\
&\wedge \\
&\bigvee_{\substack{n,n'\in N, f,f'\in C \\ \psi_2(f),\psi_2(f')}} \big(\texttt{on}(n,s_1,a,f) \wedge \texttt{on}(n',s_2,a,f')\big) \Leftrightarrow \texttt{vm\_l}(a,3) \\
&\wedge \left(\begin{array}{l} \big(\bigvee_{\substack{n,f \\ \psi_4(f)}} \texttt{on}(n,s_1,a,f) \wedge \neg\texttt{vm\_on}(s_2,a)\big) \vee \\ \big(\neg\texttt{vm\_on}(s_1,a) \wedge \bigvee_{\substack{n,f \\ \psi_4(f)}} \texttt{on}(n,s_2,a,f)\big) \end{array} \right) \Leftrightarrow \texttt{vm\_l}(a,1) \\
&\wedge \\
&\neg\texttt{vm\_on}(a) \Leftrightarrow \texttt{vm\_l}(a,0)
\end{aligned}
\tag{16}
$$

5.2.4. Node utility. The utility function (Table XIX) is modeled through proposition $\texttt{utility}(n,i)$, which is defined for all $n \in N$ and $i \in \{H,N,L\}$. It represents value u_k introduced in Formula (2) associated with node n and levels H ("high"), N ("normal") and L ("low"). We assume that the utility $\texttt{utility}(n,i)$ for node n is defined only when node n receives the answer from the server. Formula (17) states that $\texttt{utility}(n,i)$ holds only when a service delivery $\texttt{de}(n,s,f)$ occurs, for some s and f. This is specified by means of two implications. First, if $\texttt{de}(n,s,f)$ holds then node n has utility i, for some i in $\{H,N,L\}$. Then, to avoid cases such that $\texttt{utility}(n,i)$ is true even though no delivery is done, the second conjunct imposes that if $\texttt{utility}(n,i)$ holds, for some n and i, then, at the same time, there must occur a service delivery $\texttt{de}(n,s,f)$, for some service s and feature f. Finally, the third conjunct imposes that the value of the utility is unique.

$$
\begin{aligned}
&\bigwedge_{n,s,f} \Big(\texttt{de}(n,s,f) \Rightarrow \bigvee_{i\in\{H,N,L\}} \texttt{utility}(n,i)\Big) \\
&\wedge \\
&\bigwedge_{i\in\{H,N,L\},n} \Big(\texttt{utility}(n,i) \Rightarrow \bigvee_{s,c} \texttt{de}(n,s,f)\Big) \\
&\wedge \\
&\bigwedge_{i\in\{H,N,L\},n} \Big(\texttt{utility}(n,i) \Rightarrow \bigwedge_{j\in\{H,N,L\}\setminus\{i\}} \neg\texttt{utility}(n,j)\Big)
\end{aligned}
\tag{17}
$$

Formula (18) translates constraints in the first row of Table XIX. For brevity, we omit the formulae for the case "Priviledge" and "User" (second and third rows), as they can easily obtained from formulae of the case "Critical" by simply changing values for parameter c and time bounds RT_i. The first part states that answer $\texttt{de}(n,s,f)$ occurs and response time $rt^f_{n,s}$ is less, or equal to, RT_1

DOI: 10.1002/smr

if the utility for node n is "high" $\texttt{utility}(n, H)$. The other formulae have a similar meaning.

$$\begin{aligned}
&\bigwedge_{\substack{n,s,f \\ f=(H,d,C)}} \Big(\texttt{utility}(n,H) \Rightarrow rt^f_{n,s} \leq RT_1\Big) \\
&\wedge \\
&\bigwedge_{\substack{n,s,f \\ f=(H,d,C)}} \Big(\texttt{utility}(n,N) \Rightarrow RT_1 < rt^f_{n,s} \leq RT_3\Big) \\
&\wedge \\
&\bigwedge_{\substack{n,s,f \\ f=(H,d,C)}} \Big(\texttt{utility}(n,L) \Rightarrow rt^f_{n,s} \geq RT_3\Big)
\end{aligned} \tag{18}$$

Formula (19) can be added by OLIVE to the current model when a node, requiring further incompatibilities when it joins the systems, brings a new constraints γ to further constraint the service execution on a virtual machine.

$$\bigvee_{v} (\texttt{on}(n, s, v, f) \Rightarrow \gamma) \tag{19}$$

If a node does not requires new incompatibilities then its γ is simply $true$, meaning that its set of constraints is empty.

The complete model is defined by formula $\Box\left(\bigwedge_{i=3}^{19}(i)\right)$, that is the conjunction of all the previous formulae, globally quantified over the time. We refer later to this formula as SYSTEM.

6. EMPIRICAL RESULTS

This section presents the temporal cost of executing OLIVE with ae^2zot in response to an adaptation event. ae^2zot is the arithmetical plugin of Zot toolkit [8] that implements the procedure for solving the Bounded Satisfiability Checking (BSC) of CLTLoc formulae. It translates the CLTLoc input formula through the reduction in [4] and [5], and it verifies the satisfiability of the outcome by invoking an external SMT-solver (Microsoft Z3, github.com/Z3Prover/z3). Zot is written in Common Lisp and ae^2zot is the first automatic tool supporting satisfiability checking for CLTLoc formulae over dense time. Satisfiability checking is a verification approach that is alternative to model-checking where a logical formula defines the system behavior instead of an operational model (like, for instance, automata or transition systems). This approach is helpful when verification is required for system that are specified by means of descriptive specifications as claimed in [29]. Although the standard technique to prove the satisfiability of CLTL and CLTLoc formulae is based on the construction of a Büchi automata [5, 9], the evidence has turned out that it is rather expensive in practice, even in the case of LTL (the size of the automaton is exponential with respect to the size of the formula) and that new techniques should be investigated. BSC tackles the complexity of checking the satisfiability for CLTL formulae by avoiding the unfeasible construction of the whole automaton. By unrolling the semantics of the formula for a finite number of steps $k > 0$, BSC tries to build a bounded representation $\alpha\beta$, of length k over a certain alphabet of atoms appearing in the formula, of an infinite ultimately periodic model for the formula of the form $\alpha\beta^\omega$. All the tests were carried out by feeding the solver with the CLTLoc formula defining the server (we name the formula with SYSTEM) and an integer value k (see 4.1). The solver runs on an Ubuntu Linux machine 14.04.2, Xeon E5530 2.4GHz with 3GB Ram.

The first experiment is devised to measure the scalability and the cost of real-time verification of the (tentative) server model. To account for different system implementations, each experiment is run with a specific value of the following server parameters:

- number of nodes in the system and

DOI: 10.1002/smr

- number of the features (priority, deadline and type) that nodes require.

The experiment simulates tight timing requirements on the service execution: given any initial system configuration, all service requests $\texttt{re}(n, s, f)$ have to be served within 5 seconds. This demand amounts to verifying the satisfiability of the specification SYSTEM conjoined with a formula requiring that any $\texttt{on}(n, s, v, f)$ eventually holds, within 5 seconds from the origin. Informally, the cost for resolving the server model and the formula gives the magnitude of the time and memory cost to foresee the evolution of the server, undergoing a specific service demand, in the next time window 5 time unit long. In this case, the prediction is related to the possibility of allocating the processes in the server so that it executes all the service requests according to the timing constraints which specify the server behavior. To verify this property, we introduce the following Formula (20) that requires a new fresh clock t, reset in the origin, to guarantee that each occurrence of $\texttt{on}(n, s, v, f)$ is instantiated before 5 time unit from the origin:

$$t = 0 \wedge \bigwedge_{n,s,v,f} \Diamond \left(\texttt{on}(n, s, v, f) \wedge t < 5\right) \tag{20}$$

To obtain a significant estimation of the scalability of the approach, we have considered the following sets of increasing values for the parameters. The number of nodes, i.e., the cardinality of N, is chosen over the set $\{2, 4, 6, 8, 10\}$ and the cardinality of F over the set $\{4, 6, 8, 10, 12\}$ (the maximum number of features in F is 24). The set of triples in F is randomly generated accordingly with the constraints on service types, priorities and deadline presented in Section 5. The values of the deadline (DT_i), occurring in a feature defined by a triple of F, and the response time (RT_i) are set to $50, 100, 200, 400ms$ and the values of the service time (ST_i) ranges over $5, 20, 40$ and $60ms$. For example, when the features supported in the server are four, set F is defined as $= \{(H, 50, C), (N, 200, U), (H, 200, U), (N, 50, P)\}$. The feature $(H, 50, C)$ represents the class of processes with the following properties: high (H) priority service request, deadline $50ms$ time units, type Critical (C)). The state space of the system is influenced by the size of the sets N and F along with the number of services and of the virtual machines in the server. If n_{VMs} and $n_{services}$ are the number of virtual machines and the number of services, the size of the system model can be measured with the value $|N| \times n_{services} \times n_{VMs} \times |F|$. In the current adopted setting, $n_{services} \times n_{VMs}$ amounts to 4, i.e., the number of all the possible ways of executing 2 services s_1 and s_2 with 2 virtual machines a and b. Therefore, considering the minimum and maximum value of $|N|$ and $|F|$, the number of all possible active tasks that are executed in the server., i.e., the number of propositions $\texttt{on}(n, s, v, f)$, ranges from 32 to 480. In addition, we experimented increasing values for k, in the set $\{6, 8, 10, 12, 15, 20\}$, to highlight the impact of the length k in the resolution of the server specification. The value k has a key role as it determines the maximum number of relevant time positions that can be used to represent periodic executions of the server represented by the CLTLoc model satisfying the server specification. Based on the experimental results we obtained, we claim that experimenting configurations with larger value for k and bigger sets N and F, has little interest as these settings would exceed 10 hours of computation. Figure 5 shows the time needed to carry out the satisfiability of the formula SYSTEM $\wedge$ (20) for $k = 10$. Each point of the curve plots the SMT time required for solving an instance of the server model instantiated with a specific pair of number of nodes and number of services. The graph reports the time in all the possible configurations generated from the sets. The full and detailed collection of data on time and memory consumption of the experiment in Figure 5 and all the other cases with k different from 10 are provided in the Appendix A.

In the graphs of Figure 6, we show the dependency of the (SMT) solving time with the length k of the traces and the cardinality of N and F. To show the results of the experiments for all the k's from 6 to 20, we restrict the sets N and F, as the case $N = 10, F = 12$ run out of time. The graphs (a), (b), (c) and (d) in Figure 6, respectively, represents the solving time for a specific cardinality of $|N|$ in $\{2, 4, 6, 8\}$. Each one collects four curves, differentiated with a specific increasing value of $|F|$. Similarly, the graphs (e), (f), (g) and (h) show the solving time for a specific cardinality of $|F|$ in $\{4, 6, 8, 10\}$ and the four curves in each of them characterize a cardinality $|N|$. The graphical evidence points out a non linear dependency of the time and the parameter k in all the cases.

DOI: 10.1002/smr

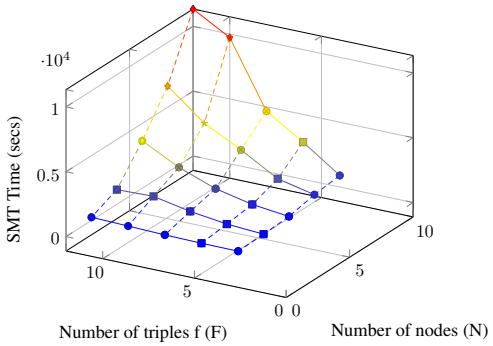


Figure 5. SMT-solver time (secs) for $k = 10$ w.r.t. number of nodes (N) and feature (F).

Moreover, a simple numerical analysis on the data shows that the time to solve the verification problem, in this experiment, grows more rapidly with respect to an increment of the number of supported nodes than with respect to an increment of the available features that characterize the service execution. In other words, solving the model for increasing value of $|N|$ is harder than solving the same model for increasing value of $|F|$.

The second experiment illustrates an example of on-line verification of a server with 4 nodes and 6 features, with $F = \{(H, 50, C), (N, 200, U), (N, 400U,), (H, 200, U), (H, 400, U), (N, 50, P)\}$. As explained in Sect. 4.2, OLIVE checks the satisfiability of formula SYSTEM specifying the system possibly conjoined with a CLTLoc formula defining the current configuration and a property. Grounding on the verification outcome, the monitor defines the strategy for adaptation. In this experiment, we assume that, when the on-line engine is inquired, the system has the following configuration: node n_1 requested for a high priority service s_1, with type *critical* and deadline $50ms$, $10ms$ in the past and the service has been executing since $1ms$ time unit on virtual machine a (so, its feature is $(H, 50, C)$). On server b, a normal priority service s_1, requested by n_2 more than $120ms$ ago with type *User* and deadline $200ms$, is starting. Node n_2 is also requesting a critical service s_2 with high priority and deadline $50ms$ (so, its feature is $(H, 50, C)$). Formula (21) captures the initial configuration by constraining in a suitable way the values of proposition on, req and the value of clocks rt and $t_{service}$.

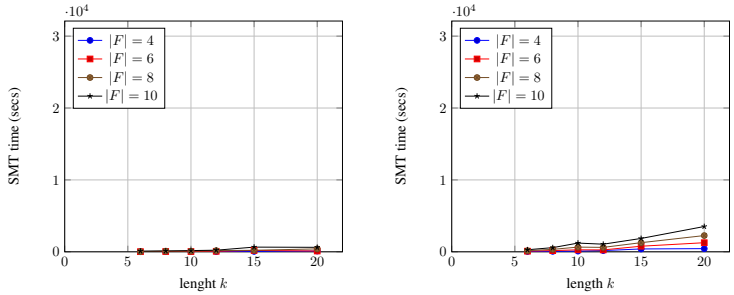
$$\begin{aligned} \texttt{on}(n_1, s_1, a, (H, 50, C)) \wedge rt_{n_1,s_1}^{(H,50,C)} = 10 \wedge t_{service}^{s_1,(H,2,C)} = 1 \wedge \texttt{re}(n_2, s_2, (N, 50, U)) \wedge \\ \texttt{on}(n_2, s_1, b, (N, 200, U)) \wedge rt_{n_2,s_1}^{(N,200,U)} \geq 120 \wedge t_{service}^{s_1,(N,200,U)} = 0 \end{aligned} \tag{21}$$

Given the initial configuration, we verify the feasibility of the service delivery associated with the request $\texttt{re}(n_2, s_2, (H, 50, C))$ with some specific requirements. The first one is the level of utility for n_2 when $\texttt{de}(n_2, s_2, (H, 50, C))$ is delivered, which we choose either to be high (H) or normal (N). The second one concerns the running environment of server when the service request is executed. For the experiment, we assume it to be on a "high demand" state situation that occurs when the server load is either $\texttt{sv\_l}(2)$ or $\texttt{sv\_l}(3)$ or $\texttt{sv\_l}(4)$ and corresponds to the case where the server is active and one (or both) VM is running with the maximum VM level 3 or both VMs are running with level 2. Formula (22)

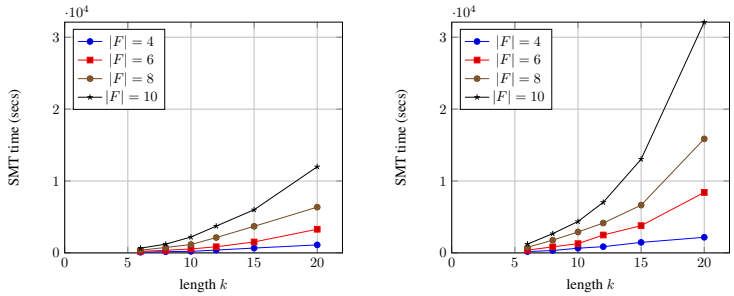
$$\neg\texttt{de}(n_2, s_2, (H, 50, C))\mathbf{U}(\texttt{de}(n_2, s_2, (H, 50, C)) \wedge \texttt{utility}(n_2, u)) \tag{22}$$

defines the reachability of the desired utility level u, with $u = H$ or $u = N$. When $u = H$, the outcome (UNSAT) of the satisfiability checking for the formula SYSTEM $\wedge$ (22), obtained in 94 secs. (82 secs. SMT time), proves that the system cannot serve the incoming request according to the timing constraints conforming to the server specification (the delivery must be provided before $100ms$ from the request to have high utility). However, when $u = N$, the system has enough time to deliver the service with normal utility, between $100ms$ and $400ms$ from the request. The outcome of the solver, obtained in 202 secs. (190 secs. SMT time), is SAT and the model of the formula shows the allocation of resources on the server (the tentative model is correct).

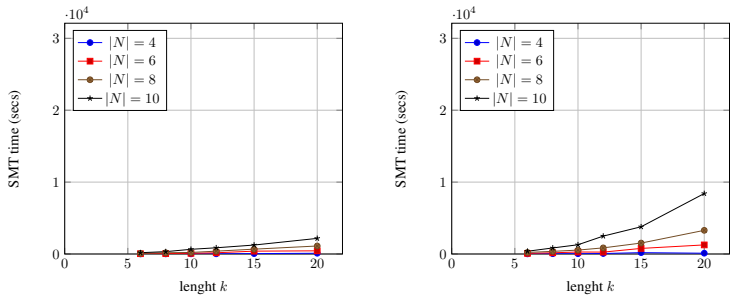
DOI: 10.1002/smr



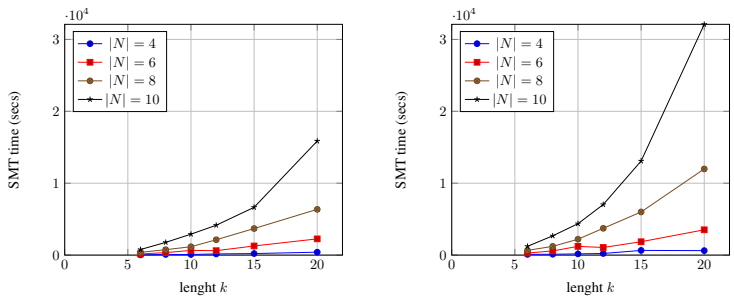
(a) $|N| = 2$ **(b)** $|N| = 4$



(c) $|N| = 6$ **(d)** $|N| = 8$



(e) $|F| = 4$ **(f)** $|F| = 6$



(g) $|F| = 8$ **(h)** $|F| = 10$

Figure 6. Solving time (SMT) for fixed $|N|$ in (a), (b), (c) and (d); and for fixed $|F|$ in (e), (f), (g) and (h).

In the second experiment, we verify if the following statement, symbolized with $(*)$, is a property for the system: *if the system delivers the service for request* $\texttt{re}(n_2, s_2, (H, 50, C))$ *with high utility before delivering the service for node* n_2 *and* n_1*, then it is not possible to service the latter with*

DOI: 10.1002/smr

high utility. The property is captured by Formula (23),

$$(\texttt{de}(n_2,s_1,(N,200,U)) \vee \texttt{de}(n_1,s_1,(H,50,C)))\mathbf{R}$$
$$\left(\begin{array}{c}\texttt{de}(n_2,s_2,(H,50,C))\\ \wedge \\ \texttt{utility}(n_2,H)\end{array}\right) \Rightarrow \neg \left(\begin{array}{c} \neg\texttt{de}(n_2,s_1,(N,200,U))\mathbf{U}\left(\begin{array}{c}\texttt{de}(n_2,s_1,(N,200,U))\\ \wedge \\ \texttt{utility}(n_2,H)\end{array}\right) \\ \wedge \\ \neg\texttt{de}(n_1,s_1,(H,50,C))\mathbf{U}\left(\begin{array}{c}\texttt{de}(n_1,s_1,(H,50,C))\\ \wedge \\ \texttt{utility}(n_1,H)\end{array}\right)\end{array}\right) \tag{23}$$

Formula (23) uses the temporal modality $\mathbf{R}$ to impose that the implication, occurring in the right hand-side of $\mathbf{R}$, holds from the origin until the position (included) where $(\texttt{de}(n_2,s_1,(N,200,U)) \vee \texttt{de}(n_1,s_1,(H,50,C)))$ holds, i.e., when one service delivery is completed. The implication expresses property $(*)$. To prove $(*)$, the negated formula $\neg$(23) translating the statement is conjoined with the specification. The outcome (SAT) of the satisfiability checking for $\texttt{SYSTEM} \wedge$ (22), that is obtained in about 210 secs. (197 secs. SMT time), shows a counterexample. The delivery $\texttt{de}(n_2,s_2,(H,50,C))$ occurs before the delivery of the running tasks, that are dealt with high utility after $\texttt{de}(n_2,s_2,(H,50,C))$. However, we can show that the statement $(*)$ is a property of the system if, in the current configuration, the server is running at level equal to 3, $\texttt{sv\_l}(3)$ is true in the origin. In this case, the outcome is UNSAT and it is obtained in 110 secs. (88 secs. SMT time). The same result is obtained for $k = 50$ in not more than 300 secs.

Finally, we show an example of off-line verification where no initial configuration is given. In this case, if a formula is a property for the system, then the property holds for all the executions and for all initial configurations that may occur when the on-line engine is invoked. To this end, we verify the validity of the following statement $(**)$ in the server. *If the server terminates the execution of the critical service in machine a and, at that time, no other critical services are running in a, and the delivery is provided with high utility, then the service level is either 1 or 2 or 3*. The statement is captured by Formula (24).

$$\Box\left(\left(\begin{array}{c}\bigwedge\limits_{n,s}\neg\texttt{on}(n,s,a,(H,50,C))\\ \wedge \\ \texttt{on}(n_1,s_2,a,(H,50,C)) \wedge \circ(\neg\texttt{on}(n_1,s_2,s,(H,50,C)))\\ \wedge \\ \circ(\neg\texttt{de}(n_1,s_2,(H,50,C))\mathbf{U}(\texttt{de}(n_1,s_2,(H,50,C)) \wedge \texttt{utility}(n_1,H)))\end{array}\right) \Rightarrow \bigvee_{i\in\{1,2,3\}} \texttt{sv\_l}(i)\right). \tag{24}$$

Checking the satisfiability of $\texttt{SYSTEM} \wedge$ (24) proves that statement $(**)$ is a property for the system as the solver returns UNSAT, in no more than 110 secs. (98 secs. SMT time) and the same test with $k = 50$ provides the same result, in almost 900 secs. To conclude the experiment, we slightly modify Formula (24) to show that the property it captures, is not a straightforward consequence of the formulae defining the server behavior and opposite veredicts can be obtained. In fact, the verification provides different outcomes from the previous one, which do not follow trivially from the server specification. The first formula is similar to Formula (24), except for the consequent which is replaced by $\bigvee_{i\in\{1,2\}} \texttt{sv\_l}(i)$. This way, the statement $(**)$ is changed in the conclusion and the modified property now requires that the service level is either 1 or 2, when the critical service terminates. The second formula is defined by removing from (24) the third conjunct in the antecedent of the implication to capture a weaker statement than $(**)$, where "the delivery is provided with high utility" of $(**)$ is ruled out. In both the cases, the outcome of the verification is SAT and the solver provides two counterexample which disprove the new formulae originated from Formula (24).

DOI: 10.1002/smr

7. DISCUSSIONS AND CONCLUSION

We presented a practical approach to research the limits of formal tools based on full CLTLoc to support the practical design of dynamic CPS, with particular interest on the design and logic of an autonomic manager for on-line verification of the tentative models of adaptive systems. We designed a specific open virtualized server containing the on-line verification manager (OLIVE) which is based on MAPE-K and employs a logic view and reasoning about the architectural model. We demonstrated experimentally that the use of formal verification, provisioning temporal boundaries for the decisions over the server operation, is possible but suitable for on-line verification when small systems and k values are considered. Precisely, by applying CLTLoc, it is ensured that the critical system properties (functional and timeliness) are preserved. We validate the manager design by implementing the model on *ae2zot* tool. We point out the cost of solving complex CLTLoc specifications, whose theoretical complexity may, in general, yield to unfeasible on-line verification processes. In fact, for modeling real-time executions of the server, the relation among the computation delay, the server load, and the temporal relation among events matter greatly; therefore, this requires a model that is affected by a combinatorial blow-up. This drawback is unavoidable in the current setting; it is also a characteristic of the approaches based on Timed automata which can be considered as an alternative to CLTLoc with the same expressiveness [6]. In addition, their inherent operational nature is not suitable to model logical constraints on quantitative measures, such as the servers load, the incompatibilities and the utility function, resulting in an elaborated representation which, on the other hand, is very natural in a logical language.

Our approach is the first developed attempt, based on a dense time temporal logic, that experiments on the support of adaptation of CPS exemplified for a virtualized server design, taking a practical step to research the applicability of pure logical models in practice, for correct construction of dynamic CPS and on-line verification. It also points out the need of ad-hoc approaches and abstraction techniques to perform on-line verification and discourages the use of general formalisms, like full LTL/CLTLoc and TA.

ACKNOWLEDGMENT

This work has been partly supported by the Salvador de Madariaga Programme for International Research Stays from the Spanish Ministry of Education (PRX12/00252); by projects REM4VSS (TIN 2011-28339), M2C2 (TIN2014-56158-C4-3-P) funded by the Spanish Ministry of Economy and Competitiveness and Italian project PRIN 2010/11 (CUP D41J12000450001); and by Horizon 2020 project no. 644869 (DICE).

REFERENCES

1. R. Alur and D. L. Dill. A theory of timed automata. Theoretical Computer Science, vol. 126, no. 2, pp. 183–235. 1994.
2. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A.l Rabkin, I. Stoica, M. Zaharia *A view of cloud computing*. Communications of the ACM, vol. 53(4), pp. 50–58. April 2010.
3. M. M. Bersani, M. García-Valls. *The cost of formal verification in adaptive CPS. An example of a virtualized server node.* In Proc. of 17^{th} IEEE International Symposium on High Assurance Systems Engineering (HASE). Orlando, FL, USA. January 2016.
4. M. M. Bersani, A. Frigeri, M. Rossi, P. San Pietro, Completeness of the bounded satisfiability problem for constraint LTL. Reachability Problems, LNCS 6945. 2011.
5. M. M. Bersani, M. Rossi, P. San Pietro *A tool for deciding the satisfiability of continuous-time metric temporal logic* Acta Informatica. DOI: 10.1007/s00236-015-0229-y. 2015.
6. M. M. Bersani, M. Rossi, P. San Pietro *A Logical Characterization of Timed (non-)Regular Languages.* MFCS, Springer 8634. 2014.
7. J. Cano, M. García-Valls. *Scheduling component replacement for timely execution in dynamic systems*. Software: Practice and Experience, vol. 44(8), pp. 889-910. 2013.
8. Deepse. *ae2zot tool*. https://github.com/fm-polimi/zot (on-line). 2015.
9. S. Demri, D. D'Souza, An automata-theoretic approach to constraint LTL. Information and Computation 205 (3), pp. 380–415. 2007.
10. M. C. Huebscher, J. A. McCann. *A survey of autonomic computing - Degrees, models and applications.* ACM Computing Surveys, vol. 40(3). 2008.
11. IBM Corporation. *An architectural blue print of autonomic computing*. Tech. report, 4^{th} ed. 2006.

DOI: 10.1002/smr

12. J. O. Kephart, D. M. Chess. *The vision of autonomic computing*. Computer, vol. 36(1), pp. 41-50. Jan. 2003.
13. J. O. Kephart, W. E. Walsh. *An artificial intelligence perspective on autonomic computing policies*. In Proc. of the 5_{th} International Workshop on Policies for Distributed Systems and Networks (POLICY). 2004.
14. K.D. Kim, P.R. Kumar. *Cyber Physical Systems: A Perspective at the Centennial.* Proceedings of the IEEE, vol. 100 (13), pp. 1287-1308. 2012.
15. M. García Valls, R. Baldoni. *Adaptive middleware design for CPS: Considerations on the OS, resource managers, and the network run-time*. Proc. 14^{th} Workshop on Adaptive and Reflective Middleware (ARM). Co-located to ACM ACM/IFIP/USENIX Middleware. 2015.
16. M. García-Valls, T. Cucinotta, C. Lu. *Challenges in real-time virtualization and predictable cloud computing*. Journal of Systems Architecture 60(9), pp. 726–740. 2014.
17. M. García-Valls, L. Fernández Villar, I. Rodríguez López. *iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems*. IEEE Transactions on Industrial Informatics, vol. 9(1), pp. 228–236. February 2013.
18. M. García-Valls, D. Perez-Palacin, R. Mirandola. *Time sensitive adaptation in CPS through run-time configuration generation and verification.* Proc. of 38^{th} IEEE Annual Computer Software and Applications Conference (COMPSAC), pp. 332–337. 2014.
19. M. García-Valls, J. Domínguez-Poblete, I. E. Touahria. *Using DDS in distributed partitioned systems.* ACM Sigbed Review. 2017. (To appear)
20. M. García-Valls, C. Calva-Urrego. *Improving service time with a multicore aware middleware*. 32^{nd} ACM Symposium on Applied Computing. Marrakech, Morocco. April 2017.
21. M. García-Valls, C. Calva-Urrego, A. Alonso, J. A. de la Puente. *Adjusting middleware knobs to assess scalability limits of distributed cyber-physical systems*. Computer Standards & Interfaces, vol. 51, pp. 95-103. 2017.
22. S. K. Khaitan, J. D. McCalley. *Design Techniques and Applications of Cyberphysical Systems: A Survey*. IEEE Systems Journal, vol. 9(2), pp. 350-365. June 2015.
23. MIT CSAIL. *Application heartbeats project.* http://code.google.com/p/hearbeats/ (on-line). 2014.
24. RTCA DO-297–EUROCAE ED-124. *Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*. `www.rtca.org` and `www.eurocae.org` (Accessed 2016)
25. G. Tesauro. *Reinforcement Learning in autonomic Computing: A Manifesto and Case Studies*. IEEE Internet Computing, vol. 11(1). 2007.
26. K. W. Tindell, A. Burns, A. J. Wellings. *Mode Changes in Priority Pre-emptively Scheduled Systems*. Proc. of the IEEE Real-Time Systems Symposium. 1992.
27. M. Toerngren, S. Tripakis, P. Derler, E.A. Lee. *Design Contracts for Cyber-Physical Systems: Making Timing Assumptions Explicit*. Tech Rep. UCB/EECS-2012-191. UC Berkeley. 2012.
28. J. Panerati, F. Sironi, M. Carminati, M. Maggio, G. Beltrame, P. Gmytrasiewicz, D. Sciuto, M. Santambrogio. *On self-adaptive resource allocation through reinforcement learning.* Proc. NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 23-30. 2013.
29. M. Pradella, A. Morzenti and P. San Pietro. Bounded Satisfiability Checking of Metric Temporal Logic Specifications. TACM vol. 22, pp. 20:1–20:54. 2013.
30. R. Schantz, D. Schmidt. *Towards adaptive and reflective middleware for network-centric combat systems.* Encyclopedia of Software Engineering. Wiley and Sons. 2002.
31. K. Ye, D. Huang, X. Jiang, H. Chen, S. Wu *Virtual Machine Based Energy-Efficient Data Center Architecture for Cloud Computing: A Performance Perspective*. IEEE/ACM International Conference on Green Computing and Communications. 2010.
32. A. Biere, A. Cimatti, E.M. Clarke and Y. Zhu, Symbolic Model Checking Without BDDs. Proc. of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS, Springer, pp. 193–207, 1999
33. F. Somenzi and R. Bloem, Efficient Büchi Automata from LTL Formulae Proc. of Computer Aided Verification: 12th International Conference, CAV, Springer, pp. 248–263, 2000.

DOI: 10.1002/smr

A. EXTENDED EXPERIMENTAL RESULTS

The following tables collect all data related to the first experiment shown in Section 6 where we try to solve the model with the constraint requiring that all the services are eventually run on the server. These experiments show the impact of the value of length k on the SMT resolution. The tests are instrumented with different values of k and by varying the cardinality of sets N and F, respectively, of nodes and features. For k equals to 6,8 and 10 we consider the cardinality of N in $\{2, 4, 6, 8, 10\}$ and the cardinality of F in $\{4, 6, 8, 10, 12\}$. Results are shown in Tables VII, VIII and IX, respectively.

The first line of each row shows the total processing time (i.e., parsing and solving) and the time taken by the SMT-solver. The second line reports the heap size (in Mbytes) required by Z3[‡]. The top line of the table shows the number of nodes N in the system and the left column the number of elements in F defining the requirements associated with requests. Graphs are plotted with the same scale on vertical z axis to allow an immediate comparison among the experiments.

All the data reported hereafter can be found at http://home.deib.polimi.it/bersani/.

Table VII. Experimental results with $k = 6$, reporting Time (secs) and heap size (MB).

	N=2	N=4	N=6	N=8	N=10
$\|F\|$=4	12.18s/15.25s 51.8	43.26s/49.31s 99.4	91.97s/100.85s 154.2	181.83s/203.25s 197.1	303.63s/319.1s 282.9
$\|F\|$=6	19.1s/23.5s 75.8	87.05s/95.67s 149.7	221.8s/234.24s 211.1	383.23s/401.36s 294.8	685.25s/708.53s 368.3
$\|F\|$=8	27.8s/33.62s 95.5	156.66s/168.47s 182.8	382.01s/398.37s 290.0	773.29s/796.25s 374.5	1179.28s/1208.0s 462.4
$\|F\|$=10	100.96s/111.7s 112.6	286.16s/299.82 222.1	654.44s/674.9s 343.3	1219.72s/1249.33s 446.2	1919.69s/1957.1s 592.0
$\|F\|$=12	139.82s/154.0s 142.8	393.92s/409.44s 284.1	874.04s/899.39s 396.5	1872.47s/1905.1s 572.4	2715.21s/2757.39s 662.9

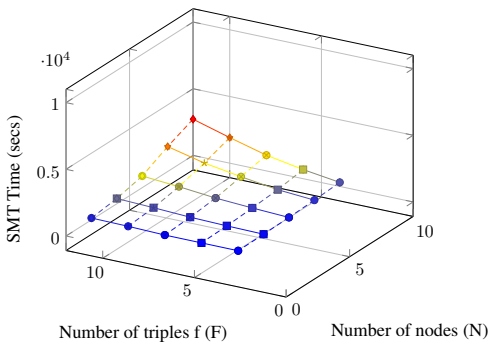


Figure 7. SMT time (secs) w.r.t. number of nodes (N) and features f (F). $k = 6$

[‡] http://research.microsoft.com/en-us/um/redmond/projects/z3/

DOI: 10.1002/smr

Table VIII. Experimental results with $k = 8$, reporting Time (min) and heap size (MB).

	N=2	N=4	N=6	N=8	N=10
$\|F\|$=4	16.88s/20.61s 72.1	61.79s/69.27s 137.76	173.73s/184.95s 184.46	327.35s/342.4s 278.9	442.99s/463.13s 329.0
$\|F\|$=6	41.04s/46.54s 91.96	168.21s/178.76s 180.0	367.83s/383.58s 284.51	841.98s/863.51s 375.22	1510.49s/1538.30s 451.04
$\|F\|$=8	93.38s/100.25s 113.91	343.24s/357.69s 227.8	754.99s/776.17s 345.77	1771.14s/1818.33s 463.25	2520.01s/2571.1s 598.41
$\|F\|$=10	119.58s/128.47s 150.92	579.83s/596.98s 292.12	1216.60s/1243.52s 431.74	2681.78s/2719.09s 588.87	4483.51s/4528.03s 802.02
$\|F\|$=12	128.69s/138.48s 167.8	735.58s/755.03s 332.7	1702.08s/1731.77s 553.7	3985.78s/4027.39s 674.5	6208.69s/6262.55s 935.42

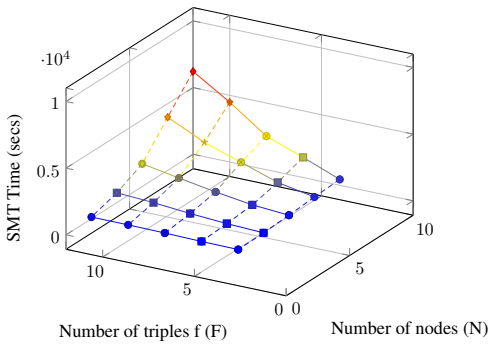


Figure 8. SMT time (secs) w.r.t. number of nodes (N) and features f (F). $k = 8$

Table IX. Experimental results with $k = 10$, reporting Time (min) and heap size (MB).

	N=2	N=4	N=6	N=8	N=10
$\|F\|$=4	23.21s/28.03s 79.2	94.8s/109.48s 152.7	215.93s/241.03s 218.7	652.97s/672.31s 303.0	890.94s/914.12s 396.0
$\|F\|$=6	39.02s/45.77s 107.0	277.05s/293.07s 210.3	544.77s/563.74s 323.0	1285.56s/1311.7s 435.2	2864.6s/2898.26s 579.3
$\|F\|$=8	87.10s/99.09s 149.3	652.02s/668.79s 288.8	1164.75s/1190.25s 433.5	2911.91s/2948.35s 585.6	4655.56s/4700.2s 795.5
$\|F\|$=10	168.46s/179.1s 171.9	1213.62s/1234.17s 339.1	2207.97s/2239.44s 560.3	4366.73s/4410.21s 558.2	9732.69s/9787.41s 996.9
$\|F\|$=12	249.89s/262.85s 201.4	1137.71s/1163.53s 416.7	3658.55s/3697.27s 622.4	6634.5s/6684.09s 921.6	11334.68s/11400.76s 1152.1

DOI: 10.1002/smr

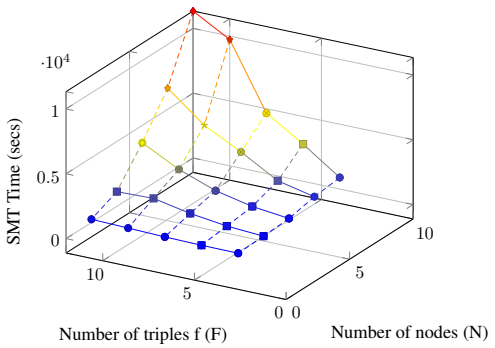


Figure 9. SMT time (secs) w.r.t. number of nodes (N) and features f (F). $k = 10$

DOI: 10.1002/smr

For k equals to 12,15 and 20 we consider the cardinality of N in $\{2, 4, 6, 8\}$ and the cardinality of F in $\{4, 6, 8, 10\}$. Results are shown in Tables X, XI and XII, respectively.

Table X. Experimental results with $k = 12$, reporting Time (sec) and heap size (MB).

	N=2	N=4	N=6	N=8
$\|F\|$=4	28.0s/33.21s 90.4	171.87s/181.94s 171.7	396.70s/412.93s 282.2	865.25s/886.13s 351.2
$\|F\|$=6	69.92s/77.18s 139.1	263.19s/277.48s 239.3	852.66s/874.27s 395.5	2489.38s/2521s 558.0
$\|F\|$=8	160.47s/169.88s 164.9	615.71s/635.55s 324.3	2134.96s/2164.37s 499.5	4154.18s/4192.99s 669.5
$\|F\|$=10	223.41s/235.12s 201.8	1059.59s/1083.1s 420.3	3726.38s/3765.78s 637.3	7033.32s/7085.17s 942.3

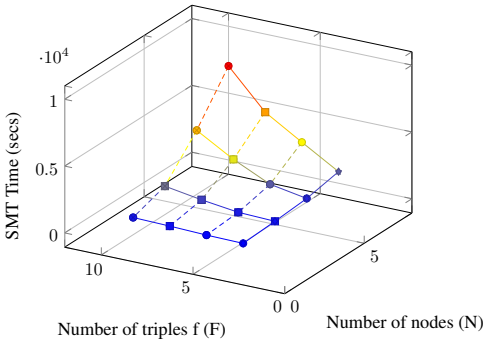


Figure 10. SMT time (secs) w.r.t. number of nodes (N) and features f (F). $k = 12$

Table XI. Experimental results with $k = 15$, reporting Time (sec) and heap size (MB).

	N=2	N=4	N=6	N=8
$\|F\|$=4	54.55s/61.1s 107.57	396.22s/419.33s 299.4	673.98s/692.88s 314.96	1460.53s/1487.1s 431.9
$\|F\|$=6	190.37s/201.33s 157.0	777.57s/803.87s 239.3	1521.18s/1564.32 462.7	3787.84s/3824.21s 635.8
$\|F\|$=8	213.57s/229.86s 196.6	1270.65s/1294.45s 415.4	3694.58s/3731.93s 624.9	6645.21s/6693.45s 908.0
$\|F\|$=10	652.4s/667.7s 241.9	1858.14s/1903.18s 494.5	6001.18s/6044.54s 842.9	13034.9s/13099.51s 1134.9

DOI: 10.1002/smr

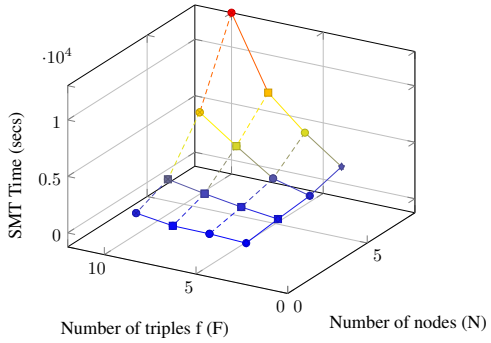


Figure 11. SMT time (secs) w.r.t. number of nodes (N) and features f (F). $k = 15$

Table XII. Experimental results with $k = 20$, reporting Time (sec) and heap size (MB).

	N=2	N=4	N=6	N=8
$\|F\|$=4	104.77s/115.52s 150.0	441.46s/458.63s 287.1	1113.65s/1137.94s 429.9	2171.8s/2204.86s 584.1
$\|F\|$=6	110.58s/120.41s 157.0	1264.96s/1299.37s 416.4	3291.27s/3325.76s 629.8	8412.52s/8461.12s 914.s
$\|F\|$=8	406.09s/432.78s 280.1	2258.58s/2289.21s 557.6	6363.28s/6408.2s 889.8	15860.32s/15942.6s 1209.5
$\|F\|$=10	622.6s/641.22s 315.8	3518.46s/3557.85s 669.9	11984.42s/12042.89s 1117.7	32100.97s/32189.58s 1656.1

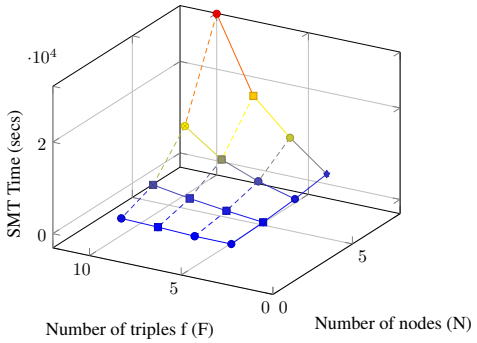


Figure 12. SMT time (secs) w.r.t. number of nodes (N) and features f (F). $k = 20$

DOI: 10.1002/smr

B. SPECIFICATION FORMULAE

Following Formulae (25) translates rules in Table XIV. The explanation is straightforward as each rule translates the correspondent row in the order. The first formula imposes that, at each position, the server load is defined and its value ranges from 0 to 4.

$$
\begin{array}{c}
\neg \texttt{vm\_on}(a) \wedge \neg \texttt{vm\_on}(b) \Leftrightarrow \texttt{sv\_l}(0) \\
\wedge \\
\left(\begin{array}{c}
\texttt{vm\_on}(a) \wedge \neg \texttt{vm\_on}(b) \wedge (\texttt{vm\_l}(a,1) \ \vee \ \texttt{vm\_l}(a,2)) \\
\vee \\
\neg \texttt{vm\_on}(a) \wedge \texttt{vm\_on}(b) \wedge (\texttt{vm\_l}(b,1) \ \vee \ \texttt{vm\_l}(b,2)) \\
\vee \\
\texttt{vm\_on}(a) \wedge \texttt{vm\_on}(b) \wedge \left(\begin{array}{c}
\texttt{vm\_l}(a,1) \wedge \texttt{vm\_l}(b,1) \vee \\
\texttt{vm\_l}(a,2) \wedge \texttt{vm\_l}(b,1) \vee \\
\texttt{vm\_l}(a,1) \wedge \texttt{vm\_l}(b,2)
\end{array} \right)
\end{array} \right) \Leftrightarrow \texttt{sv\_l}(1) \\
\wedge \\
\left(\begin{array}{c}
\texttt{vm\_on}(a) \wedge \neg \texttt{vm\_on}(b) \wedge \texttt{vm\_l}(a,3) \\
\vee \\
\neg \texttt{vm\_on}(a) \wedge \texttt{vm\_on}(b) \wedge \texttt{vm\_l}(b,3)
\end{array} \right) \Leftrightarrow \texttt{sv\_l}(2) \\
\wedge \\
\texttt{vm\_on}(a) \wedge \texttt{vm\_on}(b) \wedge \left(\begin{array}{c}
\texttt{vm\_l}(a,3) \wedge \texttt{vm\_l}(b,3) \vee \\
\texttt{vm\_l}(a,3) \wedge \texttt{vm\_l}(b,2) \vee \\
\texttt{vm\_l}(a,2) \wedge \texttt{vm\_l}(b,3)
\end{array} \right) \Leftrightarrow \texttt{sv\_l}(4) \\
\wedge \\
\texttt{vm\_on}(a) \wedge \texttt{vm\_on}(b) \wedge \left(\begin{array}{c}
\texttt{vm\_l}(a,3) \wedge \texttt{vm\_l}(b,1) \vee \\
\texttt{vm\_l}(a,2) \wedge \texttt{vm\_l}(b,2) \vee \\
\texttt{vm\_l}(a,1) \wedge \texttt{vm\_l}(b,3)
\end{array} \right) \Leftrightarrow \texttt{sv\_l}(3)
\end{array}
\tag{25}
$$

DOI: 10.1002/smr

C. HIGH LEVEL SERVER SYSTEM MODEL

This appendix compiles the comprehensive description of the model of the server software.

Table XIII. Boundaries and finite sets

Var	**Values**
Service state (Presence)	[Zero, Off, On]
Server state (On mode)	[Normal, SpecialAct, Critical]
VM no. (v)	$[V_a, V_b]$
VM state	[On, Off]
Service no. (s)	$[S_{a,1}, S_{a,2}]$ $[S_{b,1}, S_{b,2}]$
Service types	[User, Priviledged, Critica, Crypt]
Priority (p_k)	[Normal, High]
Deadline (d_k)	$[DT_1, DT_2, DT_3, DT_4]$
Utility (u_k)	[Low, Med, High]
Server load (increasing order) (l)	$[SL_0, SL_1, SL_2, SL_3, SL_4]$
VM load (l_i)	$[VL_1, VL_2, VL_3]$

The determination of the load of a system is shown in table XIV. The server load is the sum of the partial utilizations (l_i) of all virtual machines of the server. In a real-time system, the utilization can be calculated under a periodic model that is compatible with a hierarchical scheduling technique for the virtual machines. As a result, $l_i = \frac{C_i}{A_i}$ where C_i is the computation time of V_i over an activation period A_i (named T_i in real-time scheduling). A virtual machine is assigned a temporal partition that is a maximum utilization value that ensures temporal isolation between virtual machines; the verification of the model checks that the overall utilization value does not go beyond a specified threshold [26]. Other mechanisms based on response time analysis would check if $t_k \leq d_k$ holds for all nodes.

In the current model, the service time value (st_k) provided by the server that runs service k in response to a request from node x depends on the number of currently running services. This determines the server load (l) as a measure of resource availability. Additionally, the server load (l) (see table XIV) depends on the sum of the individual load that each virtual machine constitutes for the server.

Table XV determines the load of the virtual machines, that depends on the number of active services and their expected service time.

Table XVI models the specification of the server behavior with respect to the deadline values to be fulfilled, the incompatibilities and the node utility.

Table XVII determines what must be the service time (st_k) depending on the response time deadline (d_k). It models the behavior $rt_k \leq d_k$. It also shows correlation between service time (st) and the server load. T_5 is a value that represents a large time period. It is defined a set of threshold values for the service time $(st = [ST_1, ST_2, ST_3, ST_4])$ that allows to determine the server load values.

Table XIX shows the utility function to determine the benefit for the requests. It indicates the relation between the request parameters and the obtained response times (rt_k) and the level of fulfillment of incompatibilities (I). It is defined a set of threshold values for the response time

DOI: 10.1002/smr

Table XIV. Determination of the server load value (l)

Name	**Description**
Server load maximum (A on B off) (l^j affects d_k)	If $[(V_a.state = On) \wedge (V_b.state = On)] \wedge [(l_a = VL_3) \wedge (l_b = VL_3)]$ or $[(l_a = VL_3) \wedge (l_b = VL_2)]$ or $[(l_a = VL_2) \wedge (l_b = VL_3)]$ then $l = SL_4$)
Server load medium (A on B off)	If $[(V_a.state = On) \wedge (V_b.state = On)] \wedge [(l_a = VL_3) \wedge (l_b = VL_1)]$ or $[(l_a = VL_2) \wedge (l_b = VL_2)]$ or $[(l_a = VL_1) \wedge (l_b = VL_3)]$ then $l = SL_3$)
Server load low (A on B off)	If $[(V_a.state = On) \wedge (V_b.state = On)] \wedge [(l_a = VL_3) \wedge (l_b = VL_3)]$ or $[(l_a = VL_3) \wedge (l_b = VL_2)]$ or $[(l_a = VL_2) \wedge (l_b = VL_3)]$ then $l = SL_1$)
Server load low (A on B off)	If $[(V_a.state = On) \wedge (V_b.state = Off)] \wedge [(l_a = VL_1) \vee (l_a = VL_2)]]$ then $l = SL_1$)
Server load low (A off B on)	If $[(V_a.state = Off) \wedge (V_b.state = On)] \wedge [(l_b = VL_1) \vee (l_b = VL_2)]]$ then $l = SL_1$)
Server load medium (A on B off)	If $[(V_a.state = On) \wedge (V_b.state = Off)] \wedge (l_a = VL_3)]$ then $l = SL_2$)
Server load medium (A off B on)	If $[(V_a.state = Off) \wedge (V_b.state = On)] \wedge (l_b = VL_3)]$ then $l = SL_2$)
Server idle (A off B of)	If $[(V_a.state = Off) \wedge (V_b.state = Off)]$ then $l = SL_0$)

($rt = [RT_1, RT_2, RT_3, RT_4]$) that allows to determine the server load values. We show the utility type for type *Critical*.

DOI: 10.1002/smr

Table XV. Load of V_a (l_a) in relation to the execution characteristics of the services. l_b is not expressed since the pattern is repeated

Name	**Description**
$l_a = VL_3$ when $[(V_a.state = On)]$	If $[(S_{a,1}.state = On) \wedge (S_{a,2}.state = On)] \wedge [((S_{a,1}.d_k = DT_2) \vee (S_{a,1}.d_k = DT_1)) \wedge ((S_{a,2}.d_k = DT_2) \vee (S_{a,2}.d_k = DT_1))]$ then $l_a = VL_3$
$l_a = VL_2$ when $[(V_a.state = On)]$	If $[(S_{a,1}.state = On) \wedge (S_{a,2}.state = On)] \wedge [(S_{a,1}.d_k = DT_3) \wedge (S_{a,2}.d_k = DT_3)]$ then $l_a = VL_2$ If $[(S_{a,1}.state = On) \wedge (S_{a,2}.state = Off)] \wedge (S_{a,1}.d_k = DT_1)]$ then $l_a = VL_2$ If $[(S_{a,1}.state = Off) \wedge (S_{a,2}.state = On)] \wedge (S_{a,2}.d_k = DT_1)]$ then $l_a = VL_2$
$l_a = VL_1$ when $[(V_a.state = On)]$	If $[(S_{a,1}.state = On) \wedge (S_{a,2}.state = Off)] \wedge (S_{a,1}.d_k = DT_4)]$ then $l_a = VL_1$
$l_a = VL_1$ when $[(V_a.state = On)]$	If $[(S_{a,1}.state = Off) \wedge (S_{a,2}.state = On)] \wedge (S_{a,2}.d_k = DT_4)]$ then $l_a = VL_1$

Table XVI. Satisfaction level of the incompatibilities (I) in relation to the service type ($servType$)

Name	**Description**
Incompatibilities for type $Critical$	If $servType = Critical$ then there is no $Priviledged$ service in the server (in any VM) If $servType = Critical$ then there is no $User$ service in the same VM If $servType = Critical$ then if there are other services in the same VM, these are necessarily of type $CRYPT$
$satLevel_I$ per $servType$	If $servType = Critical$ then $[rt_k \leq d_k \wedge satLevel_I$ is strict] If $servType = Priviledged$ then $[rt_k \leq d_k \wedge satLevel_I$ is permissive] If $servType = User$ then $[rt_k \leq d_k \wedge satLevel$ is permissive]

DOI: 10.1002/smr

Table XVII. Priorities and deadlines according to service type

Name	**Description**
Service priority (p_k)	If $[servType = Critical]$ then $[p_k = High]$ If $[servType = User]$ then $[(p_k = High) \vee (p_k = Normal)]$ If $[servType = Priviledged]$ then $[(p_k = Normal]$
Service deadline (d_k)	If $[p_k = High]$ then $[(d_k = DT_1) \vee (d_k = DT_2) \vee (d_k = DT_3) \vee (d_k = DT_4)]$ If $[p_k = Normal]$ then $[(d_k = DT_3) \vee (d_k = DT_4)]$
Service time (st_k)	If $[l = SL_4]$ then $[st_k \geq T_4]$ If $[l = SL_3]$ then $[(ST_3 \leq st_k < ST_4)]$ If $[l = SL_2]$ then $[(ST_2 \leq st_k < ST_3)]$ If $[l = SL_1]$ then $[(ST_1 \leq st_k < ST_2)]$

Table XVIII. Utility function

Name	**Description**
Utility (u_k) for type $Critical$	If $[(p_k = High) \wedge (rt_k \leq RT_1)]$ then $[u_k = High]$ If $[(p_k = High) \wedge [(RT_1 < rt_k \leq RT_3)]]$ then $[u_k = Normal]$ If $[(p_k = High) \wedge (rt_k > RT_3)]$ then $[u_k = Low]$
Utility (u_k) for type $Priviledged$	If $[(p_k = Normal) \wedge [(rt_k \leq RT_2)]$ then $[u_k = High]$ If $[(p_k = Normal) \wedge (RT_2 < rt_k < RT_4)]$ then $[u_k = Normal]$ If $[(p_k = Normal) \wedge (rt_k \geq RT_4)]$ then $[u_k = Low]$
Utility (u_k) for type $User$	If $[(p_k = High) \wedge (rt_k \leq RT_2)]$ then $[u_k = High]$ If $[(p_k = Normal) \wedge (RT_2 < rt_k < RT_4)]$ then $[u_k = Normal]$ If $[(p_k = Normal) \wedge (rt_k \geq RT_4)]$ then $[u_k = Low]$

DOI: 10.1002/smr

Table XIX. Utility function

Name	**Description**
Utility (u_k) for type $Critical$	If $[(p_k = High) \wedge (rt_k \leq RT_1)]$ then $[u_k = High]$ If $[(p_k = High) \wedge [(RT_1 < rt_k \leq RT_3)]]$ then $[u_k = Normal]$ If $[(p_k = High) \wedge (rt_k > RT_3)]$ then $[u_k = Low]$
Utility (u_k) for type $Priviledged$	If $[(p_k = Normal) \wedge [(rt_k \leq RT_2)]$ then $[u_k = High]$ If $[(p_k = Normal) \wedge (RT_2 < rt_k < RT_4)]$ then $[u_k = Normal]$ If $[(p_k = Normal) \wedge (rt_k \geq RT_4)]$ then $[u_k = Low]$
Utility (u_k) for type $User$	If $[(p_k = High) \wedge (rt_k \leq RT_2)]$ then $[u_k = High]$ If $[(p_k = Normal) \wedge (RT_2 < rt_k < RT_4)]$ then $[u_k = Normal]$ If $[(p_k = Normal) \wedge (rt_k \geq RT_4)]$ then $[u_k = Low]$

DOI: 10.1002/smr