The cost of formal verification in adaptive CPS. An example of a virtualized server node

Marcello M. Bersani
Dipartamento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy
marcellomaria.bersani@polimi.it

Marisol García-Valls
Departament of Telematic Engineering
Universidad Carlos III de Madrid
Leganés (Madrid), Spain
mvalls@it.uc3m.es

***Abstract*—Cyber-physical systems (CPS) are large scale systems highly integrated with the physical environment. Given the changing nature of physical environments, CPS must be able to adapt on-line to new situations while preserving their *correct* operation. *Correctness by construction* relies on using formal tools, which suffer from a considerable computational overhead especially if executed on-line. As the current system model of a CPS may change to adapt to the environment, the new system model has to be verified at run-time prior to its execution to ensure that the system properties are preserved. CPS development has mainly concentrated on the design-time aspects, existing only few contributions that support on-line adaptation.**

We undertake a practical exercise to research on the pros and cons of formal tools to support dynamic changes at run-time. We formalize the semantics of the adaptation logic of an autonomic manager (OLIVE) that performs on-line verification for a specific application, i.e., a dynamic virtualized server system. We explore the realization of the autonomic manager using formal tools based on CLTLoc to express functional and non-functional properties of the managed system. The on-line verification manager services requests from mobile clients that might require a change in both the running software components and server services. To establish if the adaptation preserves the temporal constraints provided in the specification, i.e., to decide whether a new client request can be serviced in the modified system, the on-line verification manager employs CLTLoc satisfiability checking. In this scenario, we then provide empirical results showing the temporal costs of our approach.

***Index Terms*—Cyber-physical systems, virtualization, resource management, real-time, linear temporal logic, verification.**

I. Introduction

Cyber physical systems (CPS) [1] are systems with stringent timing properties at the confluence area between embedded, real-time, wireless sensor networks, and control systems. CPS are new with respect to these traditional fields due to their scale and the uncertainty created by the surrounding environment; they can be influenced by other nodes/systems both sporadically and unexpectedly. Therefore, they should adapt to new situations while preserving correct operation. In essence, CPS are adaptive decentralized systems where each one of their constituent subsystems may have very distinct functional requirements. This is the case of, for instance, remote monitoring surveillance through mobile nodes that move about 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. To ensure the temporal and spatial isolation among the various functionalities, more and more servers execute virtualized software platforms. 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 [2], some solutions that start to overcome quite a few of the problems are available to (at least) provide 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 have the coexistence of applications with different criticality levels and requirements with respect to timeliness, reliability, or security.

Adaptation in CPS can be related to the principles of *autonomic computing*, a term firstly used to describe *self-managing* [3] computing systems. An autonomic manager can, therefore, be included in the server software architecture to arbitrate the adaptation process, deciding whether the request of a mobile node can be serviced or not. This decision will depend on the current resource availability in the system.

The design of the software has to follow rigorous techniques which apply both to the initial design and to the on-line adaptation of the server. Given that the server software configuration may have to adapt to new situations (e.g. an incoming request from a mobile node which may require a new functionality to be executed/downloaded), the model of the server will be modified to handle such changes. It must be verified that the needed incremental server model updates comply with the system specification. Using formal tools to carry out the above mentioned on-line verification of the properties of the new model is a hard task due to the inherent high overhead imposed by problem solvers. This paper recognizes that there is not a *one-solution fits all* needs of dynamic CPS. Indivitual formal techniques have to be studied for a specific system (or a kind of systems) to practically assess their limits in providing a suitable verification means.

This paper provides a practical ilustration of this idea for a kind of systems such as virtualized servers that rely

on the usage of spatially and temporally isolated partitions or virtual machines. This is a typical approach in critical software systems that are transitioning from federated architectures to integrated modular functionality. This is the case of, e.g., airborne software systems that are based on integrated modular avionics (IMA). We present the software structure of a virtualized server focusing on its *On-Line Verification Entity* (OLIVE). OLIVE decides if a request from a mobile node can be serviced and takes the decision while the server is in execution, 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 bounds as required by CPS with an a-priori simulation of the system adaptation, as we present in this paper. The semantic of the system behavior is defined in Constraint LTL over clocks (CLTLoc) [5]. OLIVE is based on MAPE-K model [3], [7].

The paper is structured as follows. Section II describes related works. Section III describes the virtualized server architecture and the role of OLIVE in the context of MAPE-K adaptation strategy. Section IV presents the formal engine executed by OLIVE. Section V presents the adaptive server model. Section VI experimentally validates the approach in a realistic setting. Section VII draws the conclusions and discusses the results.

II. Related Work.

As acknowledged by some authors [8], 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 specifically due to the cost of the verification or due to the number of restrictions given the possible time between the identification of the adaptation need and the adaptation transition itself. Some approaches such as [9] verify the temporal domain exclusively using simple utilization based schedulability analysis over distributed service based applications. Other approaches [10] 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 [7] is among the most widely adopted schemes for adaptive systems. It stands for *monitor*, *analyse*, *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 [12]. 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 [13] 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. A similar approach was taken by operating system level resource managers, such as [14] that monitor real-time multi-tasking applications in video processing for consumer electronics to extract information about the degree of deadline fulfillments, and the number of attempts to overrun the granted resource budgets.

Moreover, frameworks as [15] and [3] 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. The approach of [16] designs an autonomic element, termed as the combination of an autonomic manager and a managed element. Lastly, the recent work *OMA-cy* [17] proposes an *Overarching middleware architecture for CPS* that requires the existance of a *Fast Verifier* component in the specific *CPS functions* software layer. OLIVE autonomic manager is an alternative for the Fast Verifier component.

III. Dynamic execution support architecture

The proposed architecture (Figure 1) has a key component named OLIVE (*On-LIne VErification manager*) that follows the MAPE-K loop principles. 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.

A specific concretization 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 roles 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

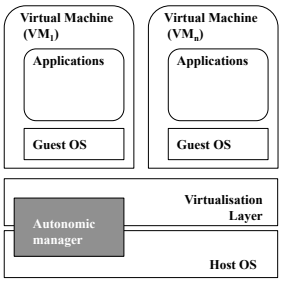

Fig. 1. *Software design of an adaptive virtualized server*

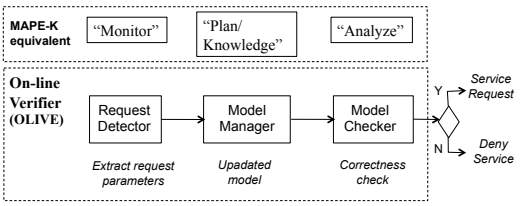

Fig. 2. *Mapping of OLIVE components to MAPE-K entities.*

for either low-level software reconfigurations based on services [9], components [18], or mode change techniques [11].

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 *Model Checker* entity that has the objective of verifying the tentative future model during execution. The Model Checker entity encapsulates the specific logic for verification. Therefore, if different formal techniques were to be used, only the *Model Checker* 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 *Model Checker* entity. The verification result depends on the fulfilment 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 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).

IV. Formal ground of OLIVE

This section describes the formal modeling and validation engine of OLIVE, i.e., the formal tools used by OLIVE for rigorous model representation and validation, which relies on *satisfiability checking* [22], an alternative approach to model-checking suitable both for verification and synthesis of systems. Instead of an operational model (like automata or transition systems), the server (system) is specified by a (temporal logic) formula that defines its execution over time. By verifying the satisfiability of the formula, the engine determines whether there exists an execution of the server satisfying the desired behavior, i.e., find a possible allocation of computational resources in the server to satisfy the node request with a given requirement. The manager uses CLTLoc with dense-time clocks to model the server (an off-line model or a tentative on-line model) and the properties.

A. Basic of CLTL over clocks

Constraint LTL over clocks (CLTLoc) [5] is a semantic restriction of Constraint LTL (CLTL) [20] allowing atomic formulae over $(\mathbb{R}, \{<, =\})$ where the arithmetical variables behave like clocks of Timed Automata (TA) [21]. 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 semantics of CLTLoc is defined as for LTL except for formulae $x \sim c$. At position $i \in \mathbb{N}$, $(\pi, \sigma), i \models x \sim c$ **iff** $\sigma(i, x) \sim c$. Formula ϕ is *satisfiable* if $(\pi, \sigma), 0 \models \phi$ for some (π, σ), called *trace* (or model).

The standard technique to prove the satisfiability of CLTL and CLTLoc formulae is based on of Büchi automata [5], [20] but, for practical implementation [5], Bounded Satisfiability

Checking (BSC) [22] is employed to avoids the onerous construction of automata. The outcome of a BSC problem, obtained by unrolling the semantics of a formula for k steps, is either an infinite ultimately periodic model or unsat. [4], [5] show that BSC for CLTLoc is complete and that is reducible to a decidable Satisfiability Modulo Theory (SMT) problem.

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.

B. Property validation in OLIVE.

To model the server in CLTLoc, we discretize the values of quantitative measures ranging over an infinite domain, like, for instance, the server load or the utility value (see Sect. V), in order to elaborate a set of finite domains and predicates. In propositional LTL-like logical formalism, in fact, quantitative variables over infinite sets cannot be expressed with propositional formulae unless a finite partition of the domains is considered. Given a finite set A and variable x over A, each element $a \in A$ is associated with x by atom p_x^a so that when p_x^a holds we then argue that $x = a$. Despite this, CLTLoc allows the specification of temporal constraints using clock variables ranging over $\mathbb{R}$, whose value is not abstracted. Clock variables represent, in the logical language and with the same precision, physical (dense) clocks. They appear in formulae of the form $x \sim c$ to express a bound c on the delay measured by clock x. Clocks are associated with specific events to measure time elapsing over the execution. As they are reset when the associated event occurs, in any moment, the clock value represents the time elapsed since the previous reset and corresponds to the elapsed time since the last occurrence of the event associated to it. We use such constraints to define, for instance, the time delay for delivering a service.

The value of all the *propositional variables* and *clocks* appearing in the CLTLoc model are evaluated at run-time when the on-line engine is inquired. 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 where the initial value of all the variables is set with the current configuration of the server; (3) verifies the satisfiability of the model by means of ae^2zot tool [5].

The result at step (3) is either SAT or UNSAT. 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 executed at runtime.

The model at (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, plugging formulae refining 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.

V. 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, results of executing OLIVE component with the ae^2zot tool are presented that show the temporal cost of the on-line decision.

A. 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 identier 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 I(a) shows the finite sets that represent the discretized value ranges for the model. The variables identified in Ta-

Var	**Values**
VM no. (v)	$[V_a, V_b]$
Service no. (s)	$[S_{a,1}, S_{a,2}]$ $[S_{b,1}, S_{b,2}]$
Service types	[User, Privileged, Critical, Crypt]
Server load (increasing order) (l)	$[SL_0, SL_1, SL_2, SL_3, SL_4]$
Rule	**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 $l = SL_4$)

TABLE I
(A) Boundaries and finite sets; (B) Definition of (max) load value l

ble I(a) are described as follows. *Number of virtual machines* is v, and the specific *virtual machines* in the server are V_a and V_b. The *number of services* (s) is specified per virtual machine, where V_a contains 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. *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 ($SL_0, SL_1, SL_2, SL_3, SL_4$).

The determination of the load of a system is shown in table I(b). 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 [11]. 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 I(b)) depends on the sum of the individual load caused by each virtual machine on the server.

Table II(a) models the specification of the server behavior with respect to the deadline values to be fulfilled, the incompatibilities and the node utility. Table II(b) 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:

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

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.

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.

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 $CRYPT$

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 $[u_k = Low]$

TABLE II
(A) SATISFACTION LEVEL OF THE INCOMPATIBILITIES (I) IN RELATION TO THE SERVICE TYPE ($servType$); (B) UTILITY FUNCTION

B. 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), specifing 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 through 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.

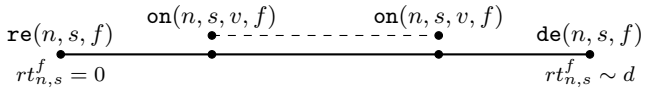

Fig. 3. Events for a node n, service s, virtual machine v and features f. The dashed line represents the interval where $\texttt{on}(n, s, v, f)$ holds.

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

Specific formulae model the sequences of events of Figure 3 by imposing that: (*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 $\texttt{re}(n, s, f)$ (Formula (3), where $\texttt{0}$ is an atom that holds only in the origin of the trace); (*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.

$$\begin{aligned} \texttt{re}(n, s, f) \Rightarrow \circ\,(\neg\texttt{re}(n, s, f)\,\mathbf{U}\,\texttt{de}(n, s, f)) \wedge \\ \texttt{de}(n, s, f) \Rightarrow \texttt{0} \vee \bullet\,(\neg\texttt{de}(n, s, f)\,\mathbf{S}\,(\texttt{re}(n, s, f) \vee \texttt{0})) \end{aligned} \tag{3}$$

We assume that an active execution cannot be preempted, i.e., when a service has been completed ($\texttt{on}(n, s, v, f)$ becomes false) the process that runs the service cannot be restarted until the delivery.

To measure the time elapsed between a service request $\texttt{re}(n, s, f)$ and its delivery $\texttt{de}(n, s, f)$, we use a clock $rt^f_{n,s}$, which is reset when $\texttt{re}(n, s, f)$ occurs. The value of $rt^f_{n,s}$, at each moment of the computation, stores the time elapsed since the occurrence of $\texttt{re}(n, s, f)$. To model the system elapsed time, we define how the server load affects the total duration of running services in the server. The speed of computation of the VMs depends on the server load that is, in turn, influenced by the tasks 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 I shows the conditions to have the maximum server load). To measure the duration of the phases, we use a pair of clocks 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.

Beside modeling the duration of phases, the model imposes also that all the running services are executed at least over one phase, and within any phase no service can either start or end. 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 $t^{n,s,v,f}_{service}$ 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 (4) 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 $t^{n,s,v,f}_{service}$ 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.

$$\begin{aligned} \texttt{on}(n, s, v, f) \Rightarrow \\ \texttt{on}(n, s, v, f)\mathbf{U}(\texttt{on}(n, s, v, f) \wedge t^{n,s,v,f}_{service} \geq k(s, v) \cdot ST_1) \end{aligned} \tag{4}$$

Propositions $\texttt{user}(s, v)$, $\texttt{priviledge}(s, v)$ and $\texttt{critic}(s, v)$ define *incompatibilities*. For instance, $\texttt{user}(s, v)$ holds when an active thread executing a service of type "User" is active. "Crypt" services are modelled by proposition $\texttt{crypt}(s)$. Proposition $\texttt{vm\_l}(v, i)$ is the *VM load* of v, where $i \in \{0, \ldots, 3\}$. When $\texttt{vm\_l}(v, i)$ holds then v is running with load VL_i (see Section V). $\texttt{sv\_l}(i)$ is server load, where $i \in \{0, \ldots, 4\}$ with similar meaning of $\texttt{vm\_l}(v, i)$ but for

the server. Proposition $\texttt{utility}(n, i)$, with $i \in \{H, N, L\}$, models utility function for a node n (H is "high", N is "normal" and L is "low"). The value of $\texttt{utility}(n, i)$ is defined only at the service delivery, i.e., concurrently with $\texttt{de}(n, s, f)$. The constraints in Table I and Table II defining the incompatibilities, server load and utility are translated straightforwardly as they do not have temporal constraints.

The CLTLoc model is defined by the conjunction of all the formulae described before, globally quantified over the time.

VI. Empirical results

This section presents the temporal cost of executing ae^2zot in response to an adaptation event. ae^2zot is the arithmetical plugin of Zot toolkit [19] 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). All the tests were carried out by feeding the solver with the CLTLoc formula defining the server (we name the formula with $(*)$) and an integer value k (see IV-A). 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 server parameters: the number of nodes in the system and the number of the features (priority, deadline and type) that nodes require. The experiment simulates tight timing requirements over the service execution: given any initial system configuration, all service requests $\texttt{re}(n, s, f)$ have to be served within 5 seconds. This amounts to verifying the satisfiability of the specification $(*)$ conjoined with a formula requiring that any $\texttt{on}(n, s, v, f)$ eventually holds, within 5 seconds from the origin. We have considered the following sets of values for the parameters as we believe that experimenting configurations with larger value has little interest (the trend is clear from Fig. 4). 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 (see Section V). The values of deadline (DT_i) and response time (RT_i) are $50, 100, 200, 400ms$ and of the service time (ST_i) are $5, 20, 40, 60ms$. In our experiments, we fix $k = 10$, being enough to represent simple (yet significant) and complex execution of the server. In fact, as k is the number of events defining periodic executions of the server captured by the CLTLoc model and $4 \times |N| \times |F|$ is the number of all possible active tasks that are executed in the server in 5 secs., we can represent from 32 up to 480 services over 10 discrete instants. Figure 4 shows the time requirements needed to check the satisfiability of the CLTLoc model.

The second experiment illustrates an example of on-line verification of a server with 4 nodes and 6 features. As

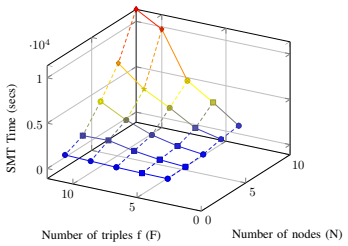

Fig. 4. SMT time (secs) w.r.t. number of nodes (N) and features f (F).

explained in Sec. IV-B, OLIVE checks the satisfiability of formula $(*)$ 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 the first experiment, we assume that, when the online 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 high 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 $(H, 50, C)$). First, we verify the feasibility of the service delivery of $\texttt{re}(n_2, s_2, (H, 50, C))$ when the service demand is high. This 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. We verify whether the delivery $\texttt{de}(n_2, s_2, (H, 50, C))$ occurs with a certain utility and, for our experiments, we consider two values: high (H) and normal (N). Formula $\neg\texttt{de}(n_2, s_2, (H, 50, C))\mathbf{U}(\texttt{de}(n_2, s_2, (H, 50, C)) \wedge \texttt{utility}(n_2, u))$ defines the reachability of the desired utility level u, with $u = H$ or $u = N$. When $u = H$, the outcome (UNSAT), obtained in 94 secs. (82 secs. SMT time), proves that the system cannot serve the incoming request according to the timing constraints (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). In the second experiment, we verify if the following statement 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 high utility*. To prove the property, the negated formula translating the statement is conjoined with the specification. The outcome (SAT), 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, the statement is a property of the system if the current configuration has server level equal to 3 (the outcome is UNSAT and it is obtained in 110 secs. (88 secs. SMT time)).

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. We verify the property saying that *if the server terminates the execution of a critical service, say s_1, with high priority and deadline equal to $50ms$ and the delivery is provided with high utility then, when the server terminates the execution, the service level is either 1 or 2*. The property holds as the solver returns UNSAT in no more than 110 secs. (98 secs. SMT time).

The previous results allows us to obtain an estimation of the upper bound of the temporal cost of the correctness check of the tentative model.

VII. 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 to 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. 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, modeling real-time executions of the server, where the relation between the computation delay and server load and the temporal relation among events matter, required a model that is affected by a combinatorial blow-up. This drawback is unavoidable in the current setting and characterizes also the approaches based on Timed automata which can be considered as an alternative to CLTLoc, being equivalent to CLTLoc [6]. In addition, their inherent operational nature is not suitable to model logical constraints on quantitative measures, like the servers load, the incompatibilities and the utility function, resulting into 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 to perform on-line verification and discourages the use of general formalisms, like full LTL/CLTLoc and TA which have PSPACE-c theoretical complexity.

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).

References

[1] 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.

[2] 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.

[3] J. O. Kephart, D. M. Chess. *The vision of autonomic computing*. Computer, vol. 36(1), pp. 41-50. Jan. 2003.

[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] IBM Corporation. *An architectural blue print of autonomic computing*. Tech. report, 4^{th} ed. 2006.

[8] M. C. Huebscher, J. A. McCann. *A survey of autonomic computing - Degrees, models and applications.* ACM Computing Surveys, vol. 40(3). 2008.

[9] M. García-Valls, L. Fernández Villar, I. Rodríguez López. *iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems* Transactions on Industrial Informatics 9(1), pp. 228-236. 2013.

[10] 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.

[11] 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.

[12] 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.

[13] MIT CSAIL. *Application heartbeats project.* http://code.google.com/p/hearbeats/ (on-line). 2014.

[14] M. García-Valls, A. Alonso, J. Ruiz, A. Groba. *An architecture for a quality of service resource manager middleware for flexible multimedia embedded systems* Proc. 3^{rd} Int'l Conference on Software Engineering and Middleware (SEM). LNCS, vol. 2596, pp. 36–55. 2003.

[15] G. Tesauro. *Reinforcement Learning in autonomic Computing: A Manifesto and Case Studies*. IEEE Internet Computing, vol. 11(1). 2007.

[16] J. Panerati et al. *On self-adaptive resource allocation through reinforcement learning.* Proc. NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 23-30. 2013.

[17] 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.

[18] 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.

[19] Deepse. *ae2zot tool*. https://github.com/fm-polimi/zot (on-line). 2015.

[20] S. Demri, D. D'Souza, An automata-theoretic approach to constraint LTL. Information and Computation 205 (3), pp. 380–415. 2007.

[21] R. Alur and D. L. Dill. A theory of timed automata. Theoretical Computer Science, vol. 126, no. 2, pp. 183–235. 1994.

[22] 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.