Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 203 (2009) 103–120
www.elsevier.com/locate/entcs

A Formal Semantics for a Quality of Service Contract Language

Christiano Braga [1]

Universidad Complutense de Madrid (UCM)
Madrid, Spain

Fabricio Chalub [2]

Instituto de Computação
Universidade Federal Fluminense (UFF)
Niterói, Brazil

Alexandre Sztajnberg [3]

Instituto de Matemática e Estatística
Pós–Graduação em Eletrônica
Universidade do Estado do Rio de Janeiro (UERJ)
Rio de Janeiro, Brazil

Abstract

Current interests in the context of system development include non-functional aspects of an application and the quality of the service (QoS) it provides. In video on demand applications, for instance, properties such as delay, bandwidth and CPU utilization are *monitored* in order to identify if they are within acceptable limits. In our approach, non-functional requirements are described by contracts. A contract specifies acceptable variations on the availability of these properties and how *service* replacement can be *negotiated* to keep the QoS of the application within the acceptable limits. In this paper we give an operational semantics for QoS contracts and report its implementation in a prototype tool that allows us to execute and analyze QoS contracts. The QoS Tool, our prototype, transforms QoS contract descriptions into modular structural operational semantics (MSOS) specifications. MSOS specifications are executable and analyzable in the Maude MSOS Tool, which uses efficient rewriting to execute, search and model checking MSOS specifications. We exemplify how the QoS Tool can be used by analyzing a video on demand application against real data.

Keywords: software architecture description languages, QoS contracts, Maude, MSOS

[1] Email: cbraga@fdi.ucm.es
[2] Email: fchalub@ic.uff.br
[3] Email: alexszt@ime.uerj.br, Phone/Fax: +55 21 2587-7391

doi:10.1016/j.entcs.2009.03.029

1 Introduction

An increasing demand can be currently observed for non-functional aspects descriptions and its impositions *a posteriori* during application execution (e.g. [13]). This demand is driven by the application designer, who needs to specify operational and quality requirements, and by the user, who knows which are the acceptable quality parameters for the application execution.

The use of a specification language to describe such requirements is a way to gather these two visions. If this language is *formal*, it may be used to execute applications with such requirements. Moreover, if this language has a *mathematical* meaning, one can actually *reason* about them before *deploying* the application.

In this paper we give a formal semantics for *QoS contracts* [10] in the CBabel software architecture description language, our specification language of choice. Using QoS contracts one may specify quality parameters for application execution in the form of interval values for QoS properties. Also, besides static requirements such as property interval values, the dynamics of *service change* can be described to manage the requirements, a characteristic that distinguishes CBabel QoS contracts from other approaches to QoS management [9]. The semantics for QoS contracts is given in operational semantics, allowing *reasoning* about QoS contract descriptions. Moreover, this semantics is *executable* and *analyzable* in the Maude MSOS Tool (MMT) [2,3] using state search and model checking. We have automated the translation process from QoS contracts to specifications in the modular SOS definition formalism (MSDF), the specification language accepted by MMT. The QoS Tool is a prototype implementation of a QoS contract analysis Tool composed by this transformation function together with MMT. Its implementation, examples and analysis discussed in this paper are available at `http://maude-msos-tool.sf.net/qostool`.

Thus, we contribute to the effort of QoS management using an approach with a *high level of abstraction* and the support of a formal-based tool to analyze QoS contracts. To report this contribution, we have organized this paper as follows. In Section 2 we exemplify the QoS contract language by means of the video on demand (VoD) application example. Section 3 presents the abstract operational semantics for the QoS contract language. Section 4 describes the implementation of the semantics given in Section 3 in the Maude MSOS Tool. In Section 5 we describe our analysis of the VoD example given in Section 2 using real data. We conclude this paper in Section 6 with our final remarks.

2 QoS Contracts and the VoD Example

In our approach we consider QoS of distributed systems as a set of non-functional requirements. Non-functional requirements are constraints that a system must fulfill, which are related to *properties* that should be *monitored*. Properties are defined by (intervals of) values that rule "how properly" the functions of a system should execute. (The word *property* here should not be understood as a logical formula,

but "as the name of a value that should be monitored".) After monitoring the properties, depending on the obtained values, a *service* should be chosen. Services are essentially interfaces representing the actual functionality of the system, implemented by software components in a given software architecture. The process of choosing a service is called *negotiation*.

In the context of video streaming on demand (VoD) through the net, for example, if there is not enough bandwidth to use a high-quality standard for video streaming then a lower quality standard could be (temporarily) used. In this example, the QoS *property* is bandwidth. The QoS requirements are the intervals of values of the QoS property. The functionality is to cast video, and the configurations for the different acceptable video streaming standards are captured as *services*. After *monitoring* the bandwidth, a service must be *negotiated* and either a high-quality video cast or a low-quality one is chosen.

QoS contracts were originally proposed in [7] essentially as record types in a programming language. Each record index is a QoS property that may be bound to an element from a collection of values, manipulated by services. That is, they represented, essentially, static information. One of the authors [10,11] extended QoS contracts with the notion of *negotiation* among services and represented their notion of QoS contract in their software architecture description language named CBabel.

The VoD example could be represented with the following QoS contract. Our description begins with QoS *categories* that define the QoS properties to be monitored and their associated types. For the VoD example we define two such categories. The category '`Processing`' declares the properties '`utilization`', representing the process load of a host, '`clockFrequency`' represents the processor speed of the host and '`memReq`' captures the amount of memory required by the application. The QoS category '`Transport`' declares two QoS properties: '`delay`' captures the acceptable interaction time between two peers through the network and '`bandwidth`' represents the acceptable rate for data transport through the network.

```
QoSCategory Processing {
  utilization: numeric;
  clockFrequency: numeric;
  memReq: numeric;
};
```

```
QoSCategory Transport {
  delay: numeric;
  bandwidth: numeric;
};
```

QoS constraints are defined by *profiles*, which describe the acceptable intervals of values for the possible QoS properties of interest. For the VoD example we define two different profiles for each QoS category. The profiles '`cpu_01`' and '`network_01`', representing local processing properties and network resources proprieties, respectively describe a high-quality standard, and '`cpu_02`' and '`network_02`' represent a low-quality standard.

```
profile {
  Processing.clockFrequency >= 700;
  Processing.utilization <= 50;
} cpu_01;

profile {
  Transport.delay <= 50;
```

```
profile {
  Processing.clockFrequency >= 266;
  Processing.utilization <= 70;
} cpu_02;

profile {
  Transport.delay <= 200;
```

```
  Transport.bandwidth >= 3/2;          Transport.bandwidth >= 56/1000;
} network_01;                        } network_02;
```

A *contract* declares a set of services and a negotiation clause among the services in order to keep up with the established QoS requirements. The VoD example has two services, one for each casting standard. Each service may enact topological operations over the software architecture such as to instantiate a component with a given profile and link the instantiated component with the rest of the software architecture. (For the purposes of this paper, however, only the profiles associated with a service are relevant. The actual connection with the software architecture is not considered.) Thus, for the VoD example the service '`MPEG_video`' is associated to profiles '`cpu_01`' and '`network_01`' and service '`H_261_video`' to profiles '`cpu_02`' and '`network_02`'.

```
contract {
  service {                            service {
    instantiate player with cpu_01;      instantiate player with cpu_02;
    link  player to server               link player to server
      by UDP_socket with network_01;       by UDP_socket with network_02;
  } MPEG_video;                        } H_261_video;
  //...
```

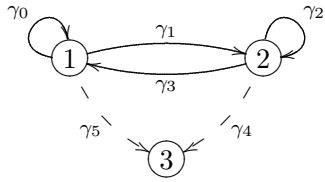
The informal semantics for the '`negotiation`' clause is as follows. Each time the system monitors QoS properties, it tries to apply a negotiation rule, following the declaration order. If the rule is positive, that is, the service on the left-hand side of the rule is not negated, the rule is applied by replacing the current service with the one on the right-hand side of the rule if the service on the left-hand side of the rule is the current service and the service on the right-hand side of the rule is *valid*. A service is said to be valid when the conjunction of the interval values on the profiles associated with the given service holds for the current monitored data. If the rule is negative, that is, the service on the left-hand side of the rule is negated, the negotiation rule is applied only if the service on the left-hand side is the current service and it is not valid and the one on the right-hand side is valid.

The negotiation clause for the VoD contract is quite simple. It simply tries to keep up with '`MPEG_video`' as much as possible. If '`MPEG_video`' is not valid, the system tries to change the current service to '`H_261_video`'. Whenever '`H_261_video`' is the current service, either if it is valid or not the system tries to change to '`MPEG_video`'.

```
//...
  negotiation {
    not MPEG_video -> H_261_video;
    H_261_video -> MPEG_video;
  }
} vod;
```

Informally, the state-transition semantics for contracts is the following one. Each state has its current monitored data and current service. A guard formula γ is a condition for the associated transition to occur. For the VoD example the state-transition system is shown in the diagram below, where the predicate *mpegvideo*, for instance, means that the profiles (that is, their intervals) bound to the service

`MPEG_video` are valid for the current monitored values. The states and transitions are as follows: state 1 has '`MPEG_video`' as the current service, in state 2 the current service is '`H_261_video`' and in state 3 the current service is the special service '`no-service`', which means that no other service can be set as the current one. The transition guards are defined by the following predicates: $\gamma_0 = mpegvideo$, $\gamma_1 = \neg mpegvideo \wedge h261video$, $\gamma_2 = h261video \wedge \neg mpegvideo$, $\gamma_3 = mpegvideo$, and $\gamma_4 = \gamma_5 = \neg h261video \wedge \neg mpegvideo$.



In Section 3 we give an abstract semantics for QoS contracts and QoS applications that capture the intuition given above.

3 An Abstract Semantics for QoS Contracts

A QoS *category* is a set $Id \times T$, where Id is a set of identifiers representing QoS properties and T is the set of basic data types in the QoS contract language.

A QoS *contract* specification is a tuple $C = ((S, <), P, I, R)$, where S is a set of identifiers representing service names with $<$ a partial order among the elements of S representing the order of declaration of the negotiation rules; P is a set of predicates parameterized by a given set D of data representing the profiles; $I \subseteq S \times P$ is a set with pairs of identifiers and predicates representing, for each service, the associated profiles; and $R \subseteq S \times S^*$ is the transition rule set given by the transition rule schemes defined below.

Let us present a few auxiliary definitions first. The set $N^+ \subseteq R$ represent the positive negotiation rules, that is whose lhs is not negated. The set $N^- \subseteq R$ represent the negative negotiation rules whose lhs is negated. A service is said to be *valid* if the predicates associated with it in I hold within the given monitored data. A service s' is one-step positively reachable from a service s or simply *positively reachable from* s if it appears as an element in the sequence $\overline{s'}$ of services related to s in N^+. A service s' is one-step negatively reachable from a service s or simply *negatively reachable* from s if it appears as an element in the sequence $\overline{s'}$ of services related to s in N^-. A service s' is one-step reachable from a service s or simply *reachable* from s if it appears as an element in the sequence $\overline{s'}$ of services related to s in $N^+ \cup N^-$. The expression $I(s)$, where s is a service identifier, yields the predicate associated with s in I.

We assume a *normalized* transition rule set, that is, given a service S there may exist only one rule $(S, S_i) \in N^+$, and only one rule $(S, S_j) \in N^-$. The rules in R are given by the following terminating transition rule schemes. There are two cases to *maintain* a service. A service s may remain as the current service if it is a valid service with the current monitored data and none of the services positively reachable from s are valid. This case is specified by the following transition rule

scheme,

$$(\forall_{1\leq i\leq n}\ s_i, \neg I(s_i)(\overline{d})) \wedge I(s)(\overline{d})$$

$$(\overline{d}, s) \rightarrow_N s\ .$$

where $\overline{d} \in D^*$, $\rho \in N^+$, $S = \pi_1(\rho)$, $S_i \in \pi_2(\rho)$, where π_1 and π_2, are the first and second projections of a pair, respectively. Also, for each service s_i that is reachable from any given service, but no service is reachable from s_i, s_i may remain the current service if it is valid. This case is specified by the following transition rule scheme,

$$I(s)(\overline{d})$$

$$(\overline{d}, s) \rightarrow_N s\ .$$

where $\forall s \in \{s_i \mid \rho_1 \in R, s_i = \pi_1(\rho_1), s_j \in \pi_2(\rho_1), \nexists \rho_2 \in R \mid s_j = \pi_1(\rho_2)\}$.

To *change* a service one must consider the cases of positive and negative negotiation rules. The current service may be changed to any service s_i related to it in N^+ if s_i is valid and all services prior to s_i in the sequence of services bound to the current service in N^+ are invalid. This case is specified by the following transition rule scheme,

$$(\forall_{1\leq j<i}\ s_j, \neg I(s_j)(\overline{d})) \wedge I(s_i)(\overline{d})$$

$$(\overline{d}, s) \rightarrow_N s_i\ .$$

where $\rho \in N^+$, $s = \pi_1(\rho)$, $s_i \in \pi_2(\rho)$. If the current service is not valid, then the same intuition of the positive case applies, but of course for set N^-. This case is specified by the following transition rule scheme,

$$\neg I(s)(\overline{d}) \wedge (\forall_{1\leq j<i}\ s_j, \neg I(s_j)(\overline{d})) \wedge I(s_i)(\overline{d})$$

$$(\overline{d}, s) \rightarrow_N s_i\ .$$

where $\rho \in N^-$, $s = \pi_1(\rho)$, s_i, $s_j \in \pi_2(\rho)$.

There are two rules that specify the *impossibility of setting any of the services in S as the current service*, denoted by $\Box$. The first case is when none of the services reachable from the given service are valid. This case is specified by the following transition rule scheme,

$$\neg I(s)(\overline{d}) \wedge \forall_{1\leq i\leq k} \neg I(s'_i)(\overline{d})$$

$$(\overline{d}, s) \rightarrow_N \Box\ .$$

where $\rho_1 \in N^+$, $\rho_2 \in N^-$, $s = \pi_1(\rho_1) = \pi_1(\rho_2)$, $k = \ \mid \pi_2(\rho_1) \cup \pi_2(\rho_2) \mid$, $s_i \in (\pi_2(\rho_1) \cup \pi_2(\rho_2))$.

The second case is when the current service is invalid and there is no rule to change the service. This case is specified by the following transition rule scheme,

$$\neg I(s)(\overline{d})$$

$$(\overline{d}, s) \rightarrow_N \Box\ .$$

where $\forall s \in \{s_i \mid \rho_1 \in R, s_i = \pi_1(\rho_1), s_j \in \pi_2(\rho_1), \nexists \rho_2 \in R \mid s_j = \pi_1(\rho_2)\}$.

The transition rules in R induce a transition relation for a contract $\rightarrow_N \subseteq (D^* \times \{S \cup \Box\} \times ((D^* \times S) \cup \{\bot\}))$, representing the negotiation process, between sequences of data, the set of services, and the set of services extended with the special service $\Box$.

A QoS application is specified by $A = (C, M, T)$ where C is a contract; M is a set of transition rules, structured according to D, representing the monitor specification, inducing a transition relation $\rightarrow_M \subseteq D^* \times D^*$ between sequences of data representing the monitoring process; and T is a set of transition rules representing how the application evolves as whole, given by the following two rules,

$$(1) \qquad \frac{\overline{d} \rightarrow_M \overline{d'} \wedge (\overline{d'}, s) \rightarrow_N s' \quad if\, s' \neq \Box}{(\overline{d}, s) \rightarrow (\overline{d'}, s')} ,$$

$$(2) \qquad (\overline{d}, \Box) \rightarrow \bot .$$

where $\overline{d}, \overline{d'} \in D^*$ and $s, s' \in S$. The first rule specifies how the system evolves (to $(\overline{d'}, s')$), with the given monitored data and current service $((\overline{d}, s))$ by first enacting the monitoring process ($\overline{d} \rightarrow_M \overline{d'}$) (that is, inspecting the QoS properties) and then negotiating a new service (s') given the new monitored data ($\overline{d'}$) and current service (s'). The second rule specifies that the system goes to a deadlock state ($\bot$) if the service "no-service" ($\Box$) is produced by the negotiation process.

Note that the specification of the monitor M is *outside* the contract specification. In Section 5 we further discuss this issue.

4 Implementing QoS Contracts in MMT

The Maude MSOS Tool (MMT) [3] is an executable environment for Modular SOS (MSOS), a modular variant of structural operational semantics. It is implemented as a formal tool in the precise sense of [4], as a realization of a semantics preserving mapping between Modular SOS and rewriting logic [12]. The modular SOS definition formalism (MSDF) [2] is the specification language supported by MMT. It allows MSOS specifications to be written in a quite succinct syntax. MSOS has SOS as a special case. Since MSOS modularity capability is not explored in this work, MSDF should be understood *here* as a concrete syntax for SOS.

The QoS Tool is essentially a transformation function from the concrete syntax for QoS Contracts into MSDF specifications following the abstract semantics given in Section 3. One can load MSDF specifications produced by the QoS Tool into MMT to execute and analyze the QoS contracts. In this section we exemplify the application of the transformation function showing excerpts of the MSDF specifications generated from the VoD example. Space constraints prevent us from describing the complete generated specification which can be found in the tool's web site at `http://maude-msos-tool.sf.net/qostool`.

The MSDF specifications generated by the transformation function extend the MSDF '`SYSTEM`' module that includes the specifications for the abstract sets '`Profile`', '`Data`' and '`Service`', and specifies the set '`System`' together with MSDF transition rules for Transition Rules 1 and 2 from Section 3. (The specifications for '`Profile`', '`Data`' and '`Service`' are omitted.)

```
msos System is
 System . System ::= deadlock | system Data ServiceId .

      (monitor Data) --> Data', ((negotiate Data' ServiceId) --> ServiceId')
[system] -- ------------------------------------------------------------
      (system Data ServiceId) : System --> system Data' ServiceId' .

[deadlock] (system Data no-service) : System --> deadlock .
sosm
```

A QoS category is transformed into an MSDF module, where each QoS property in a category is transformed into a function of type '`Data`' parameterized by the set associated with the QoS property type. At the moment only '`numeric`' and '`enum`' basic types are being handled as QoS property types. The '`numeric`' type is mapped to MSDF '`Rat`' built in data type for rational numbers. An enumeration is mapped to an MSDF set named after the concatenation of the QoS category name with the QoS property name. Each constant in an enumeration gives rise to a constant function typed as the generated set. For instance, the QoS Category '`VideoMedia`' gives rise to the MSDF module '`QoSCategory/VideoMedia`'. (The specification for '`QoSCategory/VideoMedia`' is omitted.)

Essentially, a QoS profile is represented as a conjunction of the predicates associated with each QoS profile property. The transformation affects both the grammar and transition rules of the generated MSDF specification. A QoS profile is transformed into a MSDF module that first imports the QoS category modules referenced by the profile, declares a set S, named after the QoS profile name, includes the booleans in S, and declares a function, named after the QoS profile, typed as the set generated from the QoS profile and parameterized by the set '`Data`'. Each QoS property in a profile gives rise to a function of type S, named after the concatenation of the profile name with the QoS property, which is parameterized by '`Data`'. (Note that a given QoS property may have different intervals of values in different profiles.) The transformer replaces numbers in the names of the profiles by characters coded by that number. (This is an idiosyncrasy of MSDF: types may not have numbers, because variables are automatically declared after the type names concatenated with numbers or quotes.) The transformation that affects the *syntax* of the generated MSDF module for the profile '`cpu_01`' is given by the MSDF module '`Profile/cpuAB`' below.

```
profile {
 Processing.clockFrequency = 700;
 Processing.utilization <= 50;
} cpu_01;
```

```
msos Profile/cpuAB is
 see QoScategory/Processing .
 Cpuab .
 Profile ::= Cpuab .
 Cpuab ::= Boolean | cpuab Data |
           cpuab-clockFrequency Data |
           cpuab-utilization Data .
 --- ...
sosm
```

Each QoS property in a profile gives rise to transition rules specifying when the boolean expression related to the QoS property is true or false w.r.t. the current monitored '`Data`'. (The example transition rule for the case when the predicate does not hold is omitted.)

```
Rat == 700
-- ------------------------------------------------------------
cpuab-clockFrequency (Data clockFrequency: Rat) : Cpuab --> tt .
```

Transition rules are also generated for the predicate representing the QoS profile as a conjunction of the predicates representing each QoS constraint in the profile. (Again we omit the transition rules for the cases when the predicate does not hold.)

```
((cpuab-clockFrequency Data) --> tt), ((cpuab-utilization Data) --> tt)
-- ------------------------------------------------------------------
(cpuab Data) : Cpuab --> tt .
```

Each QoS contract is transformed into an MSDF module. The contract transformation is two-fold: i) the service declaration part is transformed into set declarations, function symbols in the grammar, and transition rules specifying when a predicate representing a service is true or false w.r.t. the current '`Data`'; ii) the negotiation clause is transformed into transition rules that specify when the current service may be changed, remain the same or when none of the services can be set as the current one. In this last case, a special service named '`no-service`' becomes the current one.

The transformation for the declaration of services is quite similar to the one for profiles, that is, essentially, declaring a predicate that holds when the conjunction of the predicates for the profiles associated with the given service hold. (The generated specification is omitted.)

The transformation for the negotiation clause has two steps: i) normalization of the negotiation rules, and ii) generation of rules to change the current service, to keep the current service, and the '`no-service`' case, following the semantics in Section 3.

The normalization happens as follows. Given two rules with the same service on the left-hand side they should be merged. If the two right-hand sides are disjoint, the two rules are replaced by a rule with the service S on the left-hand side and right-hand side given by appending to the tail of the right-hand of the second rule the right-hand side of the first rule. Otherwise the order of the first rule should prevail. Our example does not need normalization but it should be easy to see how this is done in general.

The current service can only be changed by another if it is valid, that is, if the profiles of the latter are valid with the current monitored data values. (Or "if the latter service is valid", for short.) If the change of service is specified by a list of possible services in the right-hand side of the negotiation rule, for a particular service in the list to be set as the current one all its previous services must be invalid. A negotiation rule may also specify that a change may only occur if the current service is invalid. The rules to change services for the VoD example are

specified below.

```
negotiation {
  not MPEG_video -> H_261_video;
  H_261_video -> MPEG_video;
}
```

```
((mpegvideo Data) --> ff), ((hcgbvideo Data) --> tt)
-- ------------------------------------------------
(negotiate Data mpegvideo) : ServiceId --> hcgbvideo .

((mpegvideo Data) --> tt)
-- ------------------------------------------------
(negotiate Data hcgbvideo) : ServiceId --> mpegvideo .
```

The current service may remain the current one if it is valid and none of the services the system may change to, in one step, from the current service, are valid.

Rules of the form `not` $S_i \rightarrow S_{i_1}||\ldots||S_{i_m}$ are not considered since they do not influence on the maintenance of S_i. Rules to maintain a service in the VoD example are given by the rules below.

```
((mpegvideo Data) --> tt)
-- ------------------------------------------------
(negotiate Data mpegvideo) : ServiceId --> mpegvideo .

((hcgbvideo Data) --> tt), ((mpegvideo Data) --> ff)
-- ------------------------------------------------
(negotiate Data hcgbvideo) : ServiceId --> hcgbvideo .
```

As defined in Section 3, the special service '`no-service`' should become the current one in two situations. First, neither the current service is valid nor any of its one-step reachable services. The second situation is if there is no negotiation rule for the current service and it is invalid. The generated rules to handle '`no-service`' for the VoD example are as follows.

```
((hcgbvideo Data) --> ff), ((mpegvideo Data) --> ff)
-- ------------------------------------------------
(negotiate Data hcgbvideo) : ServiceId --> no-service .

((mpegvideo Data) --> ff), ((hcgbvideo Data) --> ff)
-- ------------------------------------------------
(negotiate Data mpegvideo) : ServiceId --> no-service .
```

5 An Empirical Approach for QoS Contract Analysis

The semantics specified in Section 3 and implemented in Section 4 can be used to analyze QoS contracts. The QoS Tool generates MSDF specifications that can be analyzed using MMT with different techniques such as state-space search. (Rewriting modulo axioms and model checking are also possible, though not described in this paper.) These analyses can assist:

- the designer, allowing the simulation of an execution of the contract to verify if the state machine is behaving as expected. A contract with profiles with many different properties can easily lead to inconsistencies. By simulating the execution of the contract the designer can make fine grained adjustments on the contract itself and on the profiles, and verify QoS properties of the system during design-

time;

- the deployer, that can verify if the provided resources are meeting the contract requirements. For instance, it can be verified (i) if the service with the higher priority is running most of the time and in case a switching to a lower priority service occurs, (ii) if the system can switch back again to the preferred one, or (iii) if the correct price is calculated;
- the user, that can gather information about the resources, usually provided by commercial providers, as a form to attest that the Service Level Agreement (SLA) is being meet. He may verify how and when the different services described in the contract were running, and if the correct price is being charged.

The general technique used to verify QoS contracts is based on the representation of a contract as a transition system. Therefore, "questions" that the deployer might want to make are represented by *invariants*, which are captured by predicates and verified using *search* commands. The *search* command provides an algorithm to verify invariants on finite transition systems. The "questions" are written as "search not I", where I is the invariant expressed by a predicate. The "search" execution will stop with either (i) solutions to the *search*, where I is invalid (a counter-example to the invariant), or (ii) after trasversing all the (finite) state-transition system, a solution is not found, indicating the invariant is valid.

In the case of infinite state systems the technique provides a semi-decidable procedure, that is, if there are invariant violations, they *will* be found. The *search* command is implemented with breadth-first traversal technique. Thus, if there is a solution to the "question" written in the *search* command it will always be found. (Otherwise the search will never end because the system is infinite.) Details on the invariant checking technique can be found in [5], chapter 12 ("Model checking invariants through search").

We might need to verify if the service with the higher priority (declared as the first rule in the negotiation clause) will ever be deployed. The "question" would be written as a formula "search not S", where S is an equation returning "true" if the current service is the one of higher priority.

In the next section we discuss how these techniques were applied to the VoD example and how they can be systematically used in other contexts.

5.1 An experiment on QoS contract verification

Consider the QoS contract for the VoD application (Section 2). We apply the invariant checking technique to the VoD application to exemplify how the QoS Tool can used to analyze QoS contracts. The complete runs are not shown due to space constraints. All the files used in the experiment can be downloaded from the tool's web site at `http://maude-msos-tool.sf.net/qostool`.

Given the semantics specified in Section 3, the QoS contract designer must supply, together with the QoS contract, a model of the *monitoring process*. This process is captured in our abstract semantics as the transition relation $\rightarrow_M$ and is implemented in the QoS Tool as an MSDF module with transition rules that induce

$\rightarrow_M$. However, to the best of our knowledge, there are no general models to represent properties such as the QoS properties that the VoD application handles. (The workload model being developed in [6], for instance, could be considered in future work.). We opted for an *empirical* approach to analyze the VoD application using *real data* collected form actual systems to simulate typical situations (Section 5.2 describes how this data was obtained).

With the QoS Contract and the *monitoring* module one can perform verifications using the search tools. Typical questions are:

- Is service S ever established? (i.e., there exists a state where S is the current service?);
- Is the special service '`no-service`' reached? (i.e., did the system reach a "deadlock" state, there is a state where the "no-service" is the current service?);
- If service S is the current service, will the service S' be deployed next?;
- Given a finite state transition, show the state transition system;
- Given a finite state transition system, list all states where the current service is S.

Analyzing the answers, and based on the expected dynamic behavior described in the contract, the designer can reason about the contract and identify potential problems. For instance:

- If a service described in the QoS contract is never established.
- If the preferred service (the one with higher priority in a given situation) is constantly being replaced by another service.
- If an unexpected or undesired sequence of service switching can occur.

5.2 Obtaining data for the monitoring model

We obtained data for the monitoring model using the following scenario. A machine at Universidade Federal Fluminense (UFF) was configured as a *Media Server*. Another machine running the client *Display* was configured at Universidade do Estado do Rio de Janeiro (UERJ). (Respectively 1.2 and 2.8 GHz, with 256 and 1.5 GB RAM.) These machines were connected trough a 24 Mbps ATM-based Internet link. A notebook running as a network background load injector was also configured at the UERJ. We used a 100 Mbps hub to connect the two machines at UERJ to the router. The idea was to make possible the injection of background load in the network with no additional impact to the observed nodes (i.e., in a shared media). Otherwise, using a switch, which does not emulate a shared media as a hub does, but multiple point-to-point channels, it would be necessary to inject a background flow specifically addressed to the client *Display* machine under test, generating side-effects, such as OS and buffer overheads, other than only networking overheads. Figure 1 presents the overall scenario.

In a second step we setup practical experiments to confirm some parameters regarding the properties to be measured [8,15]. For instance, H.261 codec is designed

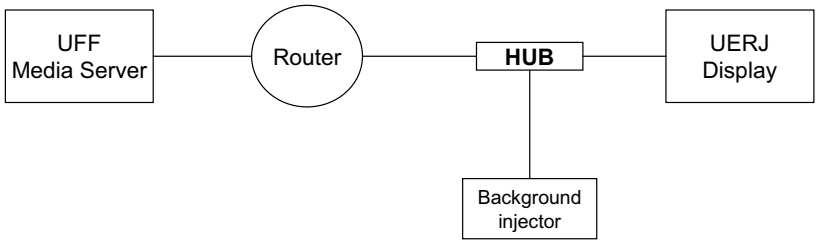


Fig. 1. Experiment scenario.

to be used "comfortably" on 128 Kbps links and MPEG usually demands a 1.6 Mbps link to be transported with low loss rate. The ITU G.114 states that a 150 ms one-way (mouth-to-ear) delay is acceptable for high voice quality. Also, [15] presents a study on human perception on media synchronization, where parameters such as delay and jitter are indexed by human comfort on viewing multimedia presentations. These measurements facilitated setting the profile parameters for the VoD contracts. The experiment consisted in transmitting pre-recorded audio and video flows from the *Media Server* to the client *Display* and monitoring the properties of interest (CPU utilization, bandwidth availability, communication delay).

We then monitored the CPU utilization of the client node and the bandwidth availability of the router connecting UFF to UERJ, using IBM's Netview SNMP [4] tool. We used a five minutes monitoring interval in a twenty four hours window, a monitoring pattern empirically useful to network management activities. Delay measurements were performed at the same time using a software agent, based on the Abing tool from Stanford [14], which performs active measures. Also, we used the default mechanisms, provided by those tools, regarding value accumulation and average calculation. In some points of time of this monitoring we injected the background load in the network and launched some processes in the client *Display* to provoke interference in resource use, thus simulating typical work load in this scenario, allowing us to evaluate how the contract would handle these events. (Details on the experiment are reported in [1].)

In the sequence, the monitored data was automatically transformed into transition rules (without premises) in a MSDF module, where, for each transition, each side has a configuration of type '`Data`' representing the values of the QoS properties required by VoD at a given moment in time. For instance, the values for the two first timestamps, that is, the first and second sets of data values that the monitoring process produced, are transformed into the following transition rule,

```
[monitor-0] (monitor (timeStamp: 1 clockFrequency: 1200 utilization: 20
                          delay: 22 bandwidth: 19 Data)) : Data -->
                    (timeStamp: 2 clockFrequency: 1200 utilization: 21
                          delay: 22 bandwidth: 19 Data) .
```

[4] SNMP - Simple Network Management Protocol is a standard application-level protocol for the internet protocol suite.

where an element of set '`Data`' is a record with '`timeStamp`', '`clockFrquency`', '`utilization`', '`delay`', and '`bandwidth`' indices, typed according to the translation of the QoS categories explained in Section 4.

5.3 Verifications

At this point we had the necessary elements to begin the verification. The MSDF module that specifies the monitor rules, together with the MSDF modules produced from the VoD contract by the QoS Tool were loaded into MMT. Then, we have searched the state space for states where different properties hold such as all the timestamps where '`mpegvideo`' is the current service,

```
(search < (system inicial mpegvideo) ::: 'System, {null} > =>+
        < (system D:Data mpegvideo) ::: 'System, {null} > .)
```

or if '`no-sevice`' is reachable after n steps, among other properties.

```
(search < (system inicial mpegvideo) ::: 'System, {null} > =>+
        < (system D:Data no-service) ::: 'System, {null} > .)
```

While running this search we have identified a situation where '`no-sevice`' was reached, at timestamp 51, out of 306 possible timestamps, reflecting a context where the client machine was running an anti-virus program together with the VoD client *Display*. The semantics of the '`no-sevice`' state indicates that, according to the contract, there are no options to continue running the application with the required quality. In this case, as soon as the CPU utilization raised, none of the services could be maintained, and the application stopped.

This simple analysis of the *search* result has shown that the contract was too rigid. Our solution was to write a more "flexible" contract with an additional *stand-by* service, A service that can be reached if none of the services are valid and form which the system may move to any of the possible services. This was specified by declaring the service bound to a profile which is always true and by changing the negotiation clause on the contract to allow the system to retry the '`MPEG_video`' or the '`H_261_video`' while the *stand-by* service is running. The extensions and changes introduced in the new contract w.r.t the previous one is given by the following declarations.

```
service {
   instantiate
       player with p_true;
} standBy;
```

```
negotiation {
    not MPEG_video -> (H_261_video || standBy);
    not H_261_video -> standBy;
    H_261_video -> MPEG_video;
    standBy -> (MPEG_video || H_261_video);
}
```

With this new version of the contract, all the possible states became reachable.

```
Solution 1
C:Conf --> < system alwTrue: 1 bandwidth: 19 clockFrequency: 1200 delay: 22
    timeStamp: 2 utilization: 21 mpegvideo ::: 'System,{null}>

[...]

Solution 305
C:Conf --> < system alwTrue: 1 bandwidth: 19 clockFrequency: 1200 delay: 20
    timeStamp: 306 utilization: 10 mpegvideo ::: 'System,{null}>
```

```
No more solutions.
```

Figure 2 shows how the QoS properties values and the current service evolved in time. The curves were derived from a *search* for all reachable states starting from the one with timestamp equal to 1. In points A and B on the CPU utilization curve (where numbers represent a percentage of the total capacity of the CPU being used), virus and malware scanners began to run at the client node. At timestamp 80, approximately, the measured delay values begin to increase. At timestamp 131 the client's machine clock is switched to 25%. At timestamp 181 the 24 Mbps link is close to saturation.

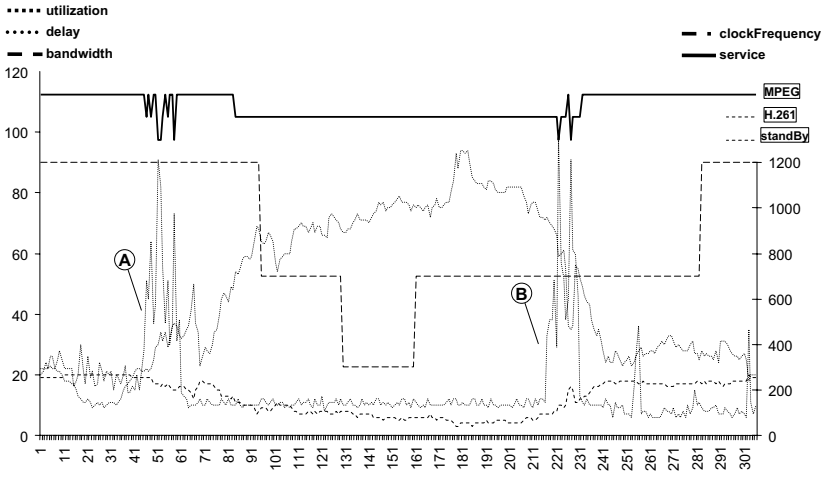


Fig. 2. Evolution of VoD's properties values over time

Intuitively one can try to interpret how the values from each individual curve can affect the application's non-functional requirements, but this is typically a multivariable function and it is not trivial to calculate how the set of measures together affect the service switching on the contract *without the help of a tool*.

For instance, further analyzing the search results we additionally identified some state sequences where the '`MPEG_video`' and '`H_261_video`' services were switched back and forth very quickly (between one or two measurement steps). These switching bursts occurred near points A and B where it can be seen utilization pikes (and the switching between services). These could be interpreted as real trends of high CPU utilization if they occurred during long periods of time (e.g. 15 minutes) or could be interpreted as transient oscillation if they occurred in short periods of time (e.g. 5 seconds). Considering the VoD application, the first case should lead to a service switch in our approach, but the second one should not. The following excerpt of the execution trace shows the "service switching problem", that is, the services change too many times within a short period of time. (Of course, the concept of "changing too often in a period of time" is application dependent.)

```
timeStamp: 50 mpegvideo
timeStamp: 51 standby
timeStamp: 52 standby
timeStamp: 53 hcgbvideo
timeStamp: 54 mpegvideo
timeStamp: 55 hcgbvideo
timeStamp: 56 mpegvideo
timeStamp: 57 mpegvideo
timeStamp: 58 standby
```

Based on the "trend interpretation", described above, we can apply data filters, such as moving average or polynomial regression to smooth the transients and pikes. Depending on the application's characteristics it can also be adopted a coarse grained sample period, and an appropriate filter when deploying the applications. This helps to avoid instability when running the application in a real environment [1].

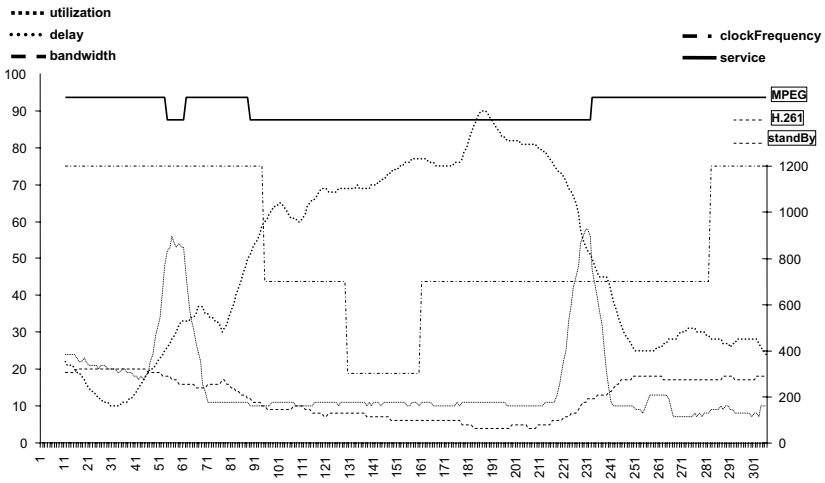


Fig. 3. Evolution of VoD's properties values over time using a moving average of 10 periods

For this experiment, we have chosen the moving average data filter. Figure 3 presents the same measurements as in Figure 2, but with a moving average of 10 periods. The values were calculated over the whole value set. These values were converted again in a new monitor and another verification run was performed. With the new results it could be verified in the behavior of the contract was more satisfactory (more stable). (See an excerpt of the execution trace below.)

```
timeStamp: 50 mpegvideo
timeStamp: 51 mpegvideo
timeStamp: 52 mpegvideo
timeStamp: 53 mpegvideo
timeStamp: 54 hcgbvideo
timeStamp: 55 hcgbvideo
timeStamp: 56 hcgbvideo
timeStamp: 57 hcgbvideo
timeStamp: 58 hcgbvideo
```

The switching between '`mpeg`' and '`hvideo`' services now occurs according to more consistent resource-variation trends and there is no need to go to the '`standby`'

service. The designer could repeat the verifications with other moving average period to evaluate the best filter to be applied while monitoring the application during run-time.

With the assistance of the QoS tool we detected two different problems in a simple application with simple QoS contract, otherwise not easily detectable. We tested and corrected the QoS contract prior to deploying it in a real situation. Also, QoS tool also helped us refining the model of the run-time monitor, which has to be actually deployed in the real system, turning it more stable and yet allowing resources constraints to be identified. These experiments also pointed us some directions to improve the verification methodology and some features on the tool. For instance, several search propositions can be automatically derived from the contract description. This would give the designer straight-forward information to plan more refined searches.

A prototype of the VoD application was actually developed using CBabel, QoS contracts and the CR-RIO infrastructure [1] and practical tests were performed. The same approach could be used on other software development and management scenarios where the concept of QoS contracts can be adopted.

6 Final Remarks

Approaches for QoS management are in general quite close to network related issues [13,8]. The approaches that are based on a specification language focus more on describing the acceptable data intervals and component interfaces and not so much on the dynamics of service negotiation. The Contract Description Language [9] is a counter-example. However, to the best of our knowledge, none are based on a formal semantics.

Our current analysis approach is based on concrete configurations for monitored data. We are currently exploring a representation for QoS services using a theorem prover so that negotiations can be analyzed in *general*. At the moment, our prototype requires from one to work directly with MSDF syntax. An interface to allow the use of syntax for QoS contracts is part of our future work.

Acknowledgement

The authors would like acknowledge the support from RNP and the Anubis Project (CTINFO/CNPq). Braga would like to acknowledge support from the Ramón y Cajal program of the Ministerio de Educación y Ciencia de España. Sztajnberg would like to acknowledge support from Faperj (APQ1 E26/171.130/05).

References

[1] L. Cardoso, A. Sztajnberg, and O. Loques. Self-adaptive applications using adl contracts. In A. Keller and J.-P. Martin-Flatin, editors, *Proceedings of The Second IEEE International Workshop on Self-Managed Networks, SelfMan 2006*, volume 3996 of *LNCS*, pages 87–101. Springer, 2006.

[2] F. Chalub. An implementation of modular structural operational semantics in maude. Master's thesis, Universidade Federal Fluminense, 2005.

[3] F. Chalub and C. Braga. Maude MSOS Tool. In C. Talcott and G. Denker, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications, WRLA 2006*, Electronic Notes in Theoretical Computer Science, pages 3–17. Elsevier, 2006.

[4] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Sther. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, FM'99, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer, 1999.

[5] M. Clavel, F. Durn, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott All About Maude: A High-Performance Logical Framework. Springer, To appear (2007).

[6] D. Feitelson. Workload characterization and modeling book. http://www.cs.huji.ac.il/~feit/wlmod/.

[7] S. Frolund and J. Koistinen. QML: A language for quality of service specification. Technical Report HPL-9810, Hewlett Packard, 1998.

[8] C. Hattingh and T. Szigeti. *End-to-End QoS Network Design: Quality of Service in LANs, WANs, and VPNs*. Cisco Press, 2004.

[9] J. Jin and K. Nahrstedt. QoS specification languages for distributed multimedia applications: a survey and taxonomy. *IEEE Multimedia*, 11(3):74–87, July-September 2004.

[10] O. Loques, R. Curty, S. Ansaloni, and A. Sztajnberg. A contract-based approach to describe and deploy non-functional adaptations in software architectures. *Journal of the Brazilian Computer Society*, 10(1), July 2004.

[11] O. Loques and A. Sztajnberg. Customizing component-based architectures by contract. In *Proceedings of the Second International Conference on Component Deployment*, volume 3083 of *LNCS*, pages 18–34. Springer, 2004.

[12] J. Meseguer. Conditional rewriting as a unified model of concurrency. Theoretical Computer Science, 96(1):73155, April 1992.

[13] D. Miras. Network QoS needs of advanced internet applications – A survey. Internet2 QoS Working Group, 2002. http://qos.internet2.edu/wg/apps/fellowship/Docs/Internet2AppsQoS Needs.pdf

[14] J. Navratil, and R. Cottrell A Practical Approach to Available Bandwidth Estimation, Passive / Active Measurement Workshop, pp.1-11, La Jolla, CA, April, 2003.

[15] R. Steinmetz, Human Perception of Jitter and Media Synchronization IEEE J. Selected Areas on Communication, Vol. 14, No. 1, pp.61-72, January, 1996.

[16] R. Steinmetz, and K. Nahrstedt Multimedia: Computing, Communications and Applications Upper Saddle River, N.J.: Prentice Hall, 1995.