# A Declarative Profile Model for QoS negotiation

Øyvind Hanssen
ohanssen@acm.org

Technical Report 2005-54
Department of Computer Science, University of Tromsø, 9037 Tromsø

December 2005

(Updated february 2006)

## Abstract

*In this report we define the semantics of a language for dynamic QoS expressions which can be evaluated at run-time for conformance. We define how expressions can be constructed from atomic expressions termed 'basic profiles' using composition operators. Two such operators are defined: The sum ( '+' ) which corresponds to simple conjunction and component-sum ('⊕') which assume that the operands denote properties of separate environments and therefore must be satisfied separately. Based on those, algorithms for conformance checking any pair of expressions can be developed. Concrete models are typically defined for specific application domains, they define the basic profile space and explicitly establishes conformance relationships between basic profiles. These are essentially sets of axioms from which we can infer conformance.*

# Contents

# A Declarative Profile Model for QoS negotiation

Øyvind Hanssen

## Abstract

*In this report we define the semantics of a language for dynamic QoS expressions which can be evaluated at run-time for conformance. We define how expressions can be constructed from atomic expressions termed 'basic profiles' using composition operators. Two such operators are defined: The sum ( '+' ) which corresponds to simple conjunction and component-sum ('⊕') which assume that the operands denote properties of separate environments and therefore must be satisfied separately. Based on those, algorithms for conformance checking any pair of expressions can be developed. Concrete models are typically defined for specific application domains, they define the basic profile space and explicitly establishes conformance relationships between basic profiles. These are essentially sets of axioms from which we can infer conformance.*

## 1. Introduction

There is much interest in how to build open and distributed applications which can be adapted to varying environmental properties as well as varying user requirements. Open systems are specified in terms of components, services and use of services. However, when implementing software components one often make implicit assumptions on extra-functional behaviour like performance, reliability, or other properties which are typically termed "Quality of Service". This may lead to implementations which are tied to specific environments and where reusability and adaptability is limited.

Extra-functional behaviour may be considered explicitly in the specification and design of systems. As well as describing functional interfaces and contracts, we may specify *QoS contracts* between components. This is for instance addressed in QoS modeling languages like QML [Frølund98a] or CQML [Aagdedal01]. However, in open systems components could be deployed or replaced at run-time, meaning that some aspects of their running environments cannot be known at design time. Requirements and properties of environments, may also change dynamically. Hence, deciding on QoS properties and contracts statically is not always sufficient.

We envisage that some aspects of component and system behaviour could be made *negotiable*. Reflective middleware, aspect oriented programming environments with dynamic aspect weaving are examples of how implementations can be supported. However, it is also necessary to perform negotiation between autonomous components, in order to reach agreements on how to configure their composition, and what extra-functional behaviour the configuration should result in. To support such negotiation, components and platforms must be able to express at run-time what they offer and what they require (c.f QoS-awareness [Frølund98b]). Expessions should be in a form which enables efficient exchange and evaluation, even though components and platforms may be highly heterogenous.

In this report we propose an expression language for QoS statements, i.e. descriptions of requirements, offers, etc., to be exchanged during negotiation. Any pair of expressions in this language can be evaluated at run-time for conformance (if one expression satisfies the other). Our model is partly based on *declared conformance*, meaning that conformance can be defined explicitly between identifiers rather than doing complex comparison of parameter values or value ranges. Our model also addresses *dynamic composition*, i.e., that expressions can be composed from simpler ones, possibly originating from different components of a system.

*Complex expressions* are constructed from atomic expressions (termed *basic profiles)* using composition operators. Basic profiles and rules for conformance are expressed in terms of *concrete models* defined for specific application domains, by domain experts. Our focus here is on an abstract *core profile model* which define how concrete profile models are defined, and how conformance can be inferred from those. We also define how expressions combining basic profiles relate to each other, thus allowing complex QoS statements to be formulated and compared at run-time.

The rest of the report is structured as follows: Section 2 gives some overall definitions and motivation, and section 3 defines the fundamentals of profile models. In section 4 we define how complex expressions are built from simpler ones by using two composing operators: '+' (sum) and '⊕' (component-sum). We develop conformance rules and a conformance evaluation algorithm for expressions. In section 5 we describe how concrete models can be defined as rule-bases. We also develop an experimental compiler which convert rule-base descriptions to testing-code and how this involves computing derived rules from axiom rules. In section 6 we analyse our model with respect to how we can check models for consistency problems, as well as how interoperability and composition can be addressed.

# 2. Context

In this section we give some overall definitions and some motivation for the ideas of our approach. The context is the need for negotiation of extra-functional behaviour resulting from *composition* in open systems. Composition may involve deployment of components in some environment, binding between components, etc. Either case would involve a decision on a *contract* and a corresponding configuration of implementation aspects, interaction protocols and resource management to realise the composition. A goal of negotiation would be to reach agreement between possibly autonomous parties on contract and configuration. The negotiation process may involve exchange of statements (offers and requirements).

## 2.1. Policy trading model

Consider some abstract service. Clients need to *bind* to it in order to use it. A binding represent a *contract*, which is a promise that the service will behave according to certain requirement, as long as its environment (including client) behaves according to certain expectations. A binding also represents a certain configuration of implementations and resources, which ensures that the client can invoke the service according to the contract.

In earlier papers, we introduce the concept of *"policy"* as an encapsulation of a potential contract and a corresponding *implementation* or *enforcement policy* [Hanssen99]. Furthermore, we propose to use *trading* of policies as a principle of negotiation [Hanssen00]. For a given service, there would exist a set of policies, each stating an *offer* and an *expectation*. The goal of negotiation is to find a policy whose offer satisfies the user requirement while its expectation is satisfied by the environment properties.

Our proposed language is meant to be used for the following: (1) expressions representing user or application requirements, (2) "*environment descriptors"*, which are expressions describing environments, (3) *"user profiles"*, which are expressions representing offers of policies, and (4) *"service profiles"*, which are expressions representing expectations of policies. All such expressions are termed *"profile expressions"*.

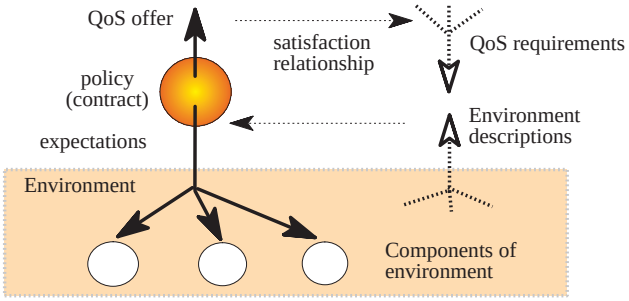


Figure 1. QoS statements and satisfaction relationships

The relationships between user requirements and offers, as well as the relationships between the QoS expectation and the capabilities of the environment, are *satisfaction* relationships. To facilitate conformance testing, our language should define a *partial order* on such expressions with respect to satisfaction. Thus, any statements could be mechanically evaluated for conformance. Figure 1 illustrates which roles profile expressions play in negotiation based on policy trading.

## 2.2. Profile models and declared conformance

Profile expressions are formulated according to a *profile model* which defines the vocabulary of expressions and rules for how these relate to each other wrt. conformance. A profile model would typically be defined for an application or application domain, but parts of it may also be shared between application domains. Traditionally (c.f. e.g. [ISO95]), QoS statements are typically predicates formulated explicitly as constraints on parameter values. Models are typically defined (for instance in QML [Frølund98a] or CQML [Aagedal01]) as a set of QoS characteristics, and contract-templates. Negotiation can be a complex task of matching parameter values, and possibly mapping between abstraction levels.

To reduce the computational complexity of matching QoS statements, it looks appealing to adopt the technique typically used in ODP trading [Bearman93, ISO97] where each requirement or offer is a reference to a type name, and where type conformance is declared a priori. This way of using declared conformance for negotiation was proposed in [Hanssen98]. Here, a profile model is defined as a set of simple names (profiles). Conformance relationships are declared explicitly. Conformance checking at run-time is then very simple, compared to evaluating a potentially complex set of QoS parameter values and constraints against each other. Figure 2 shows a simple example of a profile graph for an email application. Users can specify requirements for message delivery which are mapped to this graph. For instance, '*Authenticated*' is a subprofile of '*Secure*', i.e. it satisfies the requirements of '*Secure*'.

We consider this simple scheme to be too limiting since each declared type will need to capture all aspects relevant for the application. This may lead to conformance graphs which are too complex and application specific. Obviously, profiles would need to capture all relevant QoS dimensions, and in some cases, one may need to define a large number of profiles, for instance to cover all relevant values of a variable. Examples of this include bandwidth or latency time constraints where the differences within a group of profiles are just a numeric value and it would be more efficient and readable to use numeric metrics.

To address these problems we propose a *compromise* between simple declared conformance and parameter-based conformance. First, we propose a scheme for dynamic composition (see section 2.3 below). Second, we propose to allow profiles with simple numeric parameters. We may introduce profiles which takes one or more parameters from totally ordered domains (typically numbers) along with predicates denoting rules for how conformance relates to the parameters.

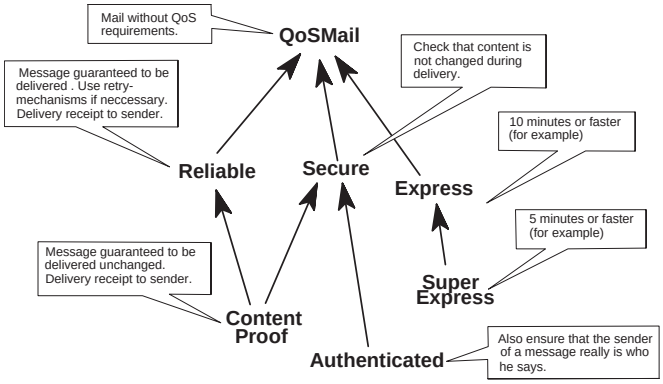


Figure 2. Example profile graph

## 2.3. Dynamic composition

QoS expressions and QoS negotiation should support *composition*, in the sense that expressions from components of a system, which do not necessarily know each other, need to be combined into one describing the composed system. Given statements about the behaviour of individual components, it is not obvious how to infer the behaviour of the total system. There are three different problems to be addressed when it comes to expressing the total behaviour:

- Autonomous users may issue different requirements for the same object and all users should be satisfied. For instance, the participants in a binding may have separate requirements for its QoS.
- We may need to combine expressions regarding the same component, but in more than one dimension.
- Open systems are systems interacting with environments neither they or their implementers controls [Abadi94]. Expectations may need to characterise a number of abstract components, for instance, client, server and communication channel, with a separate QoS expectation for each. Our model should then support dynamic composition of statements about different components of the environment.

To address the first and the second problem we introduce an operator to construct statements from simpler sub-expressions meaning that the predicates stated by each part must be true in the same environment (c.f. logical conjunction). We address the third problem by introducing an additional composing operator. It is partly addressed in QoS specification models like [Aagedal01], by allowing QoS-characteristics to be defined with such composition in mind. Work on formal models has shown that with certain assumptions on the temporal relationships [Abadi93] it is possible to make statements about the behaviour of composite systems as conjunctions of statements about each component.

# 3. Fundamentals

We are interested in defining models that let us state QoS and resource requirements as simple expressions. In addition we will introduce operators for combining simple statements into complex expressions. Profile models define a (possibly infinite) set of values called *basic profiles* and how these are related to each other by conformance. In this section we introduce basic profiles.

***Definition 1 (profile expression and environment):*** A profile expression[1] $x$ denotes a predicate $\sigma(x)$. $\sigma$ represents the interpretation of the profile expression $x$ in a specific environment. □

In the following definitions we use the term *'environment'* about $\sigma$. A requirement or offer may be stated as a reference to the profile by name, for instance by saying *'HighBW'*. The actual definition of $\sigma$ is implicit in negotiation, but may be needed by the implementers of application components. For instance, *'HighBW'* may be defined as *"(bandwidth >= 10)"* in a particular environment, meaning that a measured bandwidth of a communication channel should be higher than 100 MB/s.

***Definition 2 (basic profile expression):*** A basic profile $a$ is either a symbol $s_a$ or a pair consisting of a symbol $s_a \in Identifier$, and a parameter $t_a \in D_a$ where $D_a$ is a tuple domain $[T_1, \ldots T_n]$. □

A profile may have parameters[2], which would be given some interpretation in the context of the profile symbol. For instance *'Rtt[100]'* may denote a mean round-trip delay of 100 milliseconds. For the rest of this discussion, we limit the domain of parameters to integer numbers.

***Definition 3 (Direct conformance relationship):*** A basic profile $a$ is *satisfied* by (it is a *subprofile* of) a basic profile $b$ iff for all possible environments $\sigma$, the predicate of $a$ implies the predicate of $b$.

$a \le b \;\Leftrightarrow\; \forall\sigma\colon (\,\sigma(a) \Rightarrow \sigma(b)\,)$. □

***Definition 4 (profile model):*** A profile model defines the domain of basic profiles, $F$, and a mapping $C(F, F) \rightarrow boolean$, defining conformance directly. A conformance relationship between $a$ and $b$ exist iff $C(a,b)=true$ or if there exist a set of conformance relationships which leads to this transitively.

$$\big(C(a,b) \;\vee\; \exists c \in F\colon (a \le c \wedge c \le b)\big) \;\Leftrightarrow\; a \le b.$$ □

A concrete profile model will define a set of basic profiles plus a set of rules defining conformance relationships between them. For conformance testing, the actual meaning of a profile, $\sigma(a)$ is not used directly, since we use conformance rules to test if one basic profile satisfies another. However, policy implementers and implementers of certain local middleware components, may need to know the exact meaning of a basic profile, for instance that a profile *'HighBW'* means a bandwidth higher than e.g. 100 MB/s. A concrete environment represented by for instance a middleware platform running on a node, may incorporate software fragments which evaluate or generate simple expressions by evaluating the current situation. Thus implementers of such components would need to know the definition of $\sigma$ for basic profiles of interest. Policy implementers define and implement configurations representing mappings between a current situation and a particular QoS to be observed by the client. Thus policy implementers would also need to know the definition of $\sigma$. In contrast, a policy trading service only need to know that for instance *'SuperHighBW'* is a subprofile of *'HighBW'*.

---

[1] In this section we use the symbols *a, b, c* for profile expressions assumed to be restricted, for instance to basic (atomic) or sum profiles only. For other profiles we use the symbols *x, y, z, u, v* and *w*.

[2] We use the notation of placing parameters inside brackets following the profile symbol.

# 4. Profile expression composition

The simplest possible expression (except the empty one) is the reference of a single basic profile. However, if we were to cover all possible situations by single references to a priori defined profiles, basic profile models would need to be complex and domain specific, since an expression typically needs to describe many different dimensions, and possibly the properties of more than one component of the system. If models could be made as simple and generic as possible this would allow higher degrees of interoperability across different components and domains. We therefore allow profiles to be stated as combinations of basic profiles.

We introduce two operators, '+' (sum) and '⊕' (component-sum) to form expressions combining profiles and deduce a set of rules defining the semantics of expressions (conformance rules). Hence, we develop an algebra of profiles. Those ideas was first introduced in [Hanssen00] and they are generalised and formalised here.

## 4.1. Sum operator

The '+' operator is used to combine requirements (or offers) to be applied to the same environment. To satisfy a sum $x+y$ both $x$ and $y$ must be satisfied.

***Definition 5 (sum):*** For any environment σ, for the predicate of a sum to be true, the predicate of both operands must be true:

$\sigma(x+y) \Leftrightarrow \sigma(x) \wedge \sigma(y)$. □

Note that our model do *not* assume orthogonality between $x$ and $y$. It is also obvious from the rules of logic, that $x + x = x$.

***Theorem 1 (associaticity and commutativity):*** Sums follows the associative law: $x+(y+z) = (x+y)+z = x+y+z$ and the commutative law: $x+y = y+x$. □

This can be proved by writing profiles as predicates using definition 1 and the associative and commutative laws for logic.

$\sigma(x+(y+z)) \Leftrightarrow \sigma(x) \wedge \sigma(y+z) \Leftrightarrow \sigma(x) \wedge \sigma(y) \wedge \sigma(z)$
$\sigma((x+y)+z) \Leftrightarrow \sigma(x+y) \wedge \sigma(z) \Leftrightarrow \sigma(x) \wedge \sigma(y) \wedge \sigma(z)$

***Theorem 2 (satisfaction by sum):*** A basic profile is satisfied by a sum, if and only if it is satisfied by at least one of the operands of the sum.

$(x+y) \le a \Leftrightarrow x \le a \vee y \le a$ . □

Proof: From definition 3 and definition 5 we can see that the property holds where $x$ and $y$ are assumed to be basic profiles only:

$$(a+b) \le c \Leftrightarrow \Big(\big(\sigma(a) \wedge \sigma(b)\big) \Rightarrow \sigma(c)\Big) \Leftrightarrow \neg\big(\sigma(a) \wedge \sigma(b)\big) \vee \sigma(c)$$

$$\Leftrightarrow \big(\neg\sigma(a) \vee \sigma(c) \vee \neg\sigma(b) \vee \sigma(c)\big) \Leftrightarrow \Big(\big(\sigma(a) \Rightarrow \sigma(c)\big) \vee \big(\sigma(b) \Rightarrow \sigma(c)\big)\Big) \Leftrightarrow a \le c \vee b \le c$$

Let us now assume that the theorem is true also where $x$ and $y$ can be sums of length $m$ and $n$ operands (induction hypothesis). If we substitute $x$ with $(x+a)$ or $y$ with $(y+a)$, where $a$ is a basic profile, we observe that the theorem is still true for sums of length $m+1$ and $n+1$, hence it is true where $x$ and/or $y$ are sums of any length:

$((x+a) + b) \le c \Leftrightarrow (x+a) \le c \vee b \le c \Leftrightarrow (x \le c \vee a \le c) \vee b \le c$

***Theorem 3 (satisfaction of sum):*** A profile $x$ satisfies a sum if and only if it satisfies both sides of the sum. Here $x$, $y$ and $z$ may be any sum or basic profile.

$x \le (y+z) \;\Leftrightarrow\; x \le y \;\wedge\; x \le z$ . □

We can prove this by first proving from definition 1 and definition 3 that the property holds for basic profiles.

$$a \le (b+c) \;\Leftrightarrow\; \Big(\sigma(a) \Rightarrow \big(\sigma(b) \wedge \sigma(c)\big)\Big) \;\Leftrightarrow\; \neg\sigma(a) \vee \big(\sigma(b) \wedge \sigma(c)\big)$$

$$\Leftrightarrow\; \big(\neg\sigma(a) \vee \sigma(b)\big) \wedge \big(\neg\sigma(a) \vee \sigma(c)\big) \;\Leftrightarrow\; \big(\sigma(a) \Rightarrow \sigma(b)\big) \wedge \big(\sigma(a) \Rightarrow \sigma(c)\big) \;\Leftrightarrow\; a \le b \wedge a \le c$$

To show that $x$ can be any sum, let us now assume that the theorem is true for sum $x$ which has a length of $n$ operands (induction hypothesis). We then show that it is true for a sum of length $n+1$, i.e. where $x$ is substituted with $a+x$. By using theorem 2 and the induction hypothesis we get:

$$a \le (b+c) \;\Leftrightarrow\; a \le (b+c) \wedge x \le (b+c)$$

$$\Leftrightarrow\; (a \le b \wedge a \le c) \vee (x \le b \wedge x \le c) \;\Leftrightarrow\; (a \le b \vee x \le b) \wedge (a \le c \vee x \le b) \wedge (a \le b \vee x \le c) \wedge (a \le c \vee x \le c)$$

$$\Leftrightarrow\; (a \le b \vee x \le b) \wedge (a \le b \vee x \le b) \;\Leftrightarrow\; (a+x) \le b \;\wedge\; (a+x) \le c$$

The sub-expressions $(a \le c \vee x \le b)$ and $(a \le b \vee x \le c)$ are always true due to the induction hypothesis and can then be eliminated. To show that $y$ and $z$ can be any sum, let us assume that the theorem is true for any sums $y$ and $z$. We observe that it is also valid where $y$ or $z$ is substituted with $y+a$ (or $z+a$).

$$(x \le ((y+a) + z) \;\Leftrightarrow\; x \le (y+a) \wedge x \le z \;\Leftrightarrow\; (x \le y \wedge x \le a) \wedge x \le z$$

***Theorem 4 (comparison of sums)***: For any pair of sums $x$ and $y$ of basic profiles, a sum $x$ satisfies the sum $y$ iff there for all components of $x$ exist a component of $y$ that satisfies it.

$$\sum_{i=1}^{n} a_i \le \sum_{j=1}^{m} b_j \;\Leftrightarrow\; \forall a_j \in \{b_1 \ldots b_m\} : \Big(\exists a_i \in \{a_1 \ldots a_n\} : a_i \le b_j\Big).$$ □

This follows from theorem 2 and theorem 3. To prove this we first show that it is true for the base case where each sum contain only one element (i.e. they are simply basic profiles):

$a \le b \;\Leftrightarrow\; a \le b$

We assume that the theorem is true where $x$ is a sum of length $n$ (induction hypothesis). We then observe that this is also true where $x$ is of length $n+1$, i.e. it is substituted with $x+a$ (where $a$ is a basic profile):

$x+a \le y \;\Leftrightarrow\; \exists u \in \{x, a\} : u \le y \;\Leftrightarrow\; x \le y \;\vee\; a \le y$

which obviously follows from theorem 2. This is valid where $y$ is a basic profile. It will also be valid if we assume that y is any sum profile.

Furthermore, we assume that the theorem is also true where $y$ can be a sum of length $n$ (induction hypothesis). If $y$ is substituted with $y+a$ we get:

$x \le y+a \;\Leftrightarrow\; \forall v \in \{y, a\} : x \le v \;\Leftrightarrow\; x \le y \;\wedge\; x \le a$

which obviously follows from theorem 3. This is valid where $x$ is a basic profile. It will also be valid if we assume that $x$ is any sum profile.

## 4.2. Component-sum operator

Basic profiles or sums (of basic profiles) describe properties of a single environment. However, we need to describe open component systems, where each component represents a separate environment with separate properties. For instance, we may need to characterise the client side and the server side of remote bindings separately.

The '⊕' (component-sum) operator is used to state expressions regarding separate environments. Those environments represents separate contexts. The main idea is that to satisfy a component-sum $x \oplus y$ both $x$ and $y$ must be satisfied, but unlike sums, $x$ and $y$ cannot be satisfied by the same profile. The satisfying expression must be a component-sum with separate operands satisfying each $x$ and $y$. To define the semantics of this operator, we start with the definition of composite environments.

***Definition 6 (composite environments)**:* A composite environment σ is a collection of components $\sigma_1, \sigma_2, \ldots \sigma_n$ such that any given sum[3] profile $a$ which is true with σ, is true with at least one of $\sigma_i$. No satisfaction relationship exist between profiles if they are applied to different component environments.

$\sigma(a) \Leftrightarrow \sigma_1(a) \vee \sigma_2(a) \vee \ldots \vee \sigma_n(a)$
$\forall\, i,j \in \{1..n\} : (\, \sigma_i(a) \Rightarrow \sigma_j(b)\,) \Rightarrow i=j\,.$ □

This reflects that component environments are to be treated as separate; a statement about one environment cannot be satisfied by a statement about another. For instance a requirement for processing at the server side cannot be satisfied by processing capacity at the client side.

***Definition 7 (component-sum):*** The '⊕' (component-sum) operator denotes that each operand is applied to different components of the environment. As for sums, for the predicate of a component-sum to be true, both component predicates must be true, but in separate component environments. Observe that order of operands is not significant.

$\sigma(a \oplus b) \Leftrightarrow \exists \sigma_i, \sigma_j \in \{\sigma_1 \ldots \sigma_n\}: \sigma_i(a) \wedge \sigma_j(b)$
*where a and b are sum profiles and* σ *is a composition of* $\sigma_1 \ldots \sigma_n$. □

***Theorem 5 (associativity and commutativity):*** component-sums follows the associative law: $x \oplus (y \oplus z) = (x \oplus y) \oplus z = x \oplus y \oplus z$, and the commutative law: $x \oplus y = y \oplus x$. □

The proof is similar to the proof of associativity and commutativity for sums (theorem 1).

***Theorem 6 (satisfaction of component-sum):*** A profile $x$ satisfies a component-sum y iff $x$ is a component-sum and each operand of $y$ is satisfied by an unique operand of $x$:

Let *Perm[1..n]* be the set of all possible permutations of the numbers *1..n*.

$x \le (b \oplus c) \Leftrightarrow \exists s \in Perm[1..n]: (\, x_{s(1)} \le b \wedge x_{s(2)} \le c\,)$ *where* $x = (a_1 \oplus \ldots \oplus a_n)$. □

To prove this, we first show that $(b \oplus c)$ can not be satisfied by a simple sum profile, by using definition 6 and definition 7 to rewrite the proposition to a conjunction where the second and third part are always false due to the second part of definition 6.

$$a \le (b \oplus c) \Leftrightarrow \left(\sigma_1(a) \vee \sigma_2(a) \Rightarrow \sigma_1(b)\right) \wedge \left(\sigma_1(a) \vee \sigma_2(a) \Rightarrow \sigma_2(c)\right)$$
$$\Leftrightarrow \left(\sigma_1(a) \Rightarrow \sigma_1(b)\right) \wedge \left(\sigma_2(a) \Rightarrow \sigma_1(b)\right) \wedge \left(\sigma_1(a) \Rightarrow \sigma_2(c)\right) \wedge \left(\sigma_2(a) \Rightarrow \sigma_2(c)\right) \Leftrightarrow \mathit{false}$$

---

[3] By a 'sum profile' we mean a sum of basic profiles, or a single basic profile, i.e. a sum of only one element, c.f. the identity law (theorem 9).

Now, replace $a$ with a component-sum with $n$ operands $x = (a_1 \oplus ... \oplus a_n)$. If we can show that

(1) If there exist two component-sum operands of $x$: $a_i$ and $a_j$ such that $(a_i \oplus a_j) \le (b \oplus c)$, then $x \le (b \oplus c)$

(2) $(c \oplus d) \le (a \oplus b) \Leftrightarrow (c \le a \wedge d \le b) \vee (c \le a \wedge d \le b)$

Then it is straightforward to prove the theorem. (1) follows from the fact that if $x$ satisfies a profile expression $y$, $x \oplus a$ also does. Using definition 7 this is equivalent to $(\sigma_1(x) \wedge \sigma_2(a)) \Rightarrow \sigma_1(y)$ which is obviously true if $\sigma_1(x) \Rightarrow \sigma_1(y)$. (2) follows from definition 6 and definition 7:

$$\sigma(c \oplus d) \Rightarrow \sigma(a \oplus b) \Leftrightarrow \left(\sigma_1(c) \wedge \sigma_2(d) \Rightarrow \sigma_A(a) \wedge \sigma_B(b)\right)$$
$$\Leftrightarrow \left(\sigma_1(c) \wedge \sigma_2(d) \Rightarrow \sigma_A(a)\right) \wedge \left(\sigma_1(c) \wedge \sigma_2(d) \Rightarrow \sigma_B(b)\right)$$
$$\Leftrightarrow \left(\left(\sigma_1(c) \Rightarrow \sigma_A(a)\right) \vee \left(\sigma_2(d) \Rightarrow \sigma_A(a)\right)\right) \wedge \left(\left(\sigma_1(c) \Rightarrow \sigma_B(b)\right) \vee \left(\sigma_2(d) \Rightarrow \sigma_B(b)\right)\right)$$

There are two possibilities for matching environment functions, either by setting index numbers: $A=1$ and $B=2$, or $A=2$ and $B=1$ respectively. In each case one of the implications in each of the two disjunctions will be always be false due to the second part of definition 6. Thus the expression can be rewritten as follows:

$$\left(\left(\sigma_1(c) \Rightarrow \sigma_1(a)\right) \wedge \left(\sigma_2(d) \Rightarrow \sigma_2(b)\right)\right) \vee \left(\left(\sigma_2(d) \Rightarrow \sigma_2(a)\right) \wedge \left(\sigma_1(c) \Rightarrow \sigma_1(b)\right)\right)$$
$$\Leftrightarrow \left(c \le a \wedge d \le b\right) \vee \left(d \le a \wedge c \le b\right)$$

***Theorem 7 (satisfaction by component-sum):*** A basic profile (or a simple sum) is satisfied by a component-sum, iff it is satisfied by at least one of the operands of the component sum.

$(a \oplus b) \le c \Leftrightarrow a \le c \vee b \le c$ □

The proof is an application of the definitions and elimination of implications with different environments:

$$\left(a \oplus b\right) \le c \Leftrightarrow \left(\sigma_1(a) \wedge \sigma_2(b)\right) \Rightarrow \sigma(c)$$
$$\Leftrightarrow \sigma_1(a) \Rightarrow \sigma(c) \vee \sigma_2(b) \Rightarrow \sigma(c) \Leftrightarrow \sigma_1(a) \Rightarrow \left(\sigma_1(c) \vee \sigma_2(c)\right) \vee \sigma_2(b) \Rightarrow \left(\sigma_1(c) \vee \sigma_2(c)\right)$$
$$\Leftrightarrow \sigma_1(a) \Rightarrow \sigma_1(c) \vee \sigma_2(b) \Rightarrow \sigma_2(c) \Leftrightarrow a \le c \vee b \le c$$

***Theorem 8 (rule for comparing component-sums)***: A component-sum $x$ satisfies a component-sum $y$ iff every operand of $y$ is satisfied by an unique operand of $x$. Formally (we use the symbol 'Φ' to denote a component-sum):

$$\Phi_{i=1}^{n} a_i \le \Phi_{j=1}^{m} b_j \Leftrightarrow \exists s \in Perm[1..n]: \left(\forall b_j \in \left\{b_1 \ldots b_m\right\}: \left(a_{s(j)} \le b_j\right)\right)$$

where $\Phi_{i=1}^{n} a_i = a_1 \oplus a_2 \oplus ... \oplus a_n$ □

This follows from theorem 6 and theorem 7. The proof is similar to the proof of theorem 4: We first show that it is true for the base case where each component-sum contain only one element (i.e. they are basic profiles or simple sums):

$a \leq b \Leftrightarrow a \leq b$

We assume that the theorem is true where $x$ can be a component-sum of length $n$ (induction hypothesis). We then observe that this is also true for component-sums of length $n+1$, i.e. if $x$ is substituted with $u = x \oplus a$ (where $a$ is a basic profile). Since $u$ here have two elements and $b$ only one, this is equivalent to saying that either the first or second element of $u$ should satisfy $b$.

$x \oplus a \leq b \quad \Leftrightarrow \quad x \leq b \vee a \leq b$

This obviously follows from theorem 7. Furthermore, we assume that the theorem is also true where $y$ can be a component-sum of length $m$ (induction hypothesis) and see if this still holds where $y$ is subsituted with $v = y \oplus a$. Here, $x$ must be a component-sum of $n$ elements and there must exist a permutation $s(i)$ of these, such that both $y$ and $a$ are satisfied by a separate element of $v$.

$x \leq y \oplus a \quad \Leftrightarrow \quad x_{s(1)} \leq y \wedge x_{s(2)} \leq a \quad where\ x = (a_1 \oplus ... \oplus a_n)$

Which obviously follows from theorem 6.

## 4.3. General expressions and the normal form

Until now we have defined the semantics of expressions which are either basic profiles, sums of basic profiles or component-sums of sums. In the following define the semantics of expressions which may be compositions of sums and component-sums. Essentially, any profile expression can be represented in a normal form. We can then use theorem 4 and theorem 8 to test conformance between any pair of expressions.

***Definititon 8 (null profile)***: There exist a special basic profile named '*null*' such that for all σ:

$\sigma(null) = true$ □

***Theorem 9 (identity law)***:

$x + null = x$
$x \oplus null = x$ □

Proof: $\sigma(x+null) \Leftrightarrow \sigma(x) \wedge \sigma(null) \Leftrightarrow \sigma(x) \wedge true \Leftrightarrow \sigma(x)$

***Definition 9 (composite expressions):*** Expressions may be constructed by using basic profiles, the sum and component-sum operators according to operator precedence grammar where the '+' operator has precedence over the '⊕' operator:

E ::= E + E | E ⊕ E | (E) | *basic-profile* □

***Definition 10 (distributive law):*** The + operator distributes over the '⊕' operator:

$x + (y \oplus z) = (x+y) \oplus (x+z)$ □

A consequence of this is that for expressions of the form $x+(y \oplus z)$ the profile $x$ applies to both environments of $y$ and $z$.

The rules we have developed above defines the semantics of profile-expressions which are either basic profiles, sums of basic profiles or component-sums of sums of basic profiles. Sums containing component-sums can here be regarded as shortcuts for component-sums containing (shared) requirements. Now we have a complete semantics for general profile expressions.

***Definition 11 (normal form)***: A profile expression is in normal form if and only if it is a component-sum $a_1 \oplus ... \oplus a_m$ of sums $a_i = b_{i_1} + ... + b_{i_n}$ of basic profiles. □

Using the distribution law and the associative law, we can rewrite any expressions containing component-sums, to the normal form, i.e. component-sums of sums of basic profiles. For instance:

$x + (y \oplus (z + (u \oplus v))) = (x+y) \oplus (x + z + u) \oplus (x + z + v)$

Note that basic profiles or simple sums (of basic profiles), or component-sums of simple basic profiles are also in the normal form. In such cases, some component-sums or sums have only one element. This follows from the identity law (theorem 9). We can now conclude that any pair of profile-expressions can be algorithmically tested for conformance, using the rule of theorem 4 and theorem 8 and the set of rules defining conformance between basic profile defined in the concrete model of use.

## 4.4. Conformance testing algorithm

From the conformance rules defined above, we can develop an algorithm to test any pair of profile expressions for conformance. We assume that profile expressions are first transformed into *normal form*. Expressions (in normal form) can then be evaluated against each other, using a mix of (1) a component-sum test, (2) a sum test and the (3) a basic profile test evaluating rules of a basic profile model. In the following we assume that $x$ and $y$ are profile expressions and that the goal is to determine if $x \leq y$.

### *4.4.1.* Component-sum test

If all expressions to be compared are in normal form, we can start by using a test based on theorem 8 for comparing component-sums. A simple and naive algorithm can be formulated like this (the outermost call to the recursive function *isSubR* starts with i=1) :

```
boolean isSubR(Compsum x, Compsum y, int i)
{
    for (each p in x) {
        if (p ≤ y[i]) {
            remove p from x;
            if (isSubR(x, y, i+1))
               return true;
            re-insert p in x; // Backtrack
        }
    }
    return false;
}
```

The worst case complexity of this simple algorithm is $O(N!)$ where $N$ is the size of $x$ or $y$ (depending on which is smallest). Therefore we should look for a better algorithms if the sizes of expressions are not expected to be small. For instance, dynamic programming techniques could be used to eliminate repeated recursions on the same sub-expressions, possibly reducing the complexity significantly. Instead of investigating further how to improve our naive algorithm, we observe that the problem of testing component-sums for conformance corresponds (under certain restrictions) to the problem of determining the existence of a maximum matching in a bipartite graph. Each expression ($x$ and $y$) corresponds to a partition, each component of a component-sum of a node in this graph and conformance relationships (from components of $y$ to components of $x$), to edges. If a maximum matching involves all nodes in $y$, this means that expression x satisfies expression $y$. It is known from literature that this problem can be solved with algorithms of complexity $O(N \cdot E)$ where $N$ is the number of nodes and $E$ is the number of edges. However, in our case, to find the edges, we need first to use the conformance rules to determine conformance for up to every possible pair of nodes which is a $O(N^2)$ problem. This means that it is possible to do the matching with a worst case performance of $O(N^3)$.

#### *4.4.2.* Sum test

This component-sum test again make use of the test for sums of basic profiles which follows from theorem 4. A simple test with a worst case complexity of $O(N^2)$ can (in pseudocode) be formulated like this:

```
boolean isSubR(Sum x, Sum y)
{
    for (each p in y) {
        for (each q in x) {
           if (p≤q)
              return true;
    }
}
return false;
```

#### *4.4.3.* General algorithm

If expressions are in the normal form, a sum cannot contain component-sum components, i.e. $x$ and $y$ can be either component-sums (of sums or basic profiles), sums of basic profiles of just basic profiles. Furthermore, theorem 8 and the corresponding algorithm above allow component-sum components to be either sums or basic profiles. Therefore we can formulate a testing algorithm for any pair of profile expressions in the normal form. This would be a recursive function, using one of three different tests (basic profile, sum or component-sum test respectively) depending on the type of $x$ and $y$. If $x$ and $y$ are of different types, this is resolved as follows.

- A basic profile and a sum: Use the *sum test* where one of the sums contains only one element.
- A basic profile and a component-sum: Use the *component-sum test* where one of the component-sums contains only one element. Note that a basic profile cannot be a subprofile of a component-sum.
- A sum and a component-sum: Use the component-sum test where one of the component-sums contains only one element; the sum. Note that a simple sum cannot be a subprofile of a component-sum with more than one element. The normal form ensures that a sum will always be of basic profiles.

It follows from theorem 9 that sums or component-sums with only one element can exist.

# 5. Definition of profile models

A *profile model* would be defined for an application or application domain. It can be defined as a rule-base, i.e a set of rules each which state a predicate for when there is conformance between two given basic profiles. A full formal analysis of how we can use rule-bases for this purpose, is outside the scope of this report. Here we describe a relatively simple approach to *model definition,* and our experimental prototype which demonstrates it. We show how model definition and rule derivation works using an example.

## 5.1. Foundations

A profile model defines a *partial order*, i.e. a set of binary, reflexive and antisymmetric relations between points in a profile value space. Our profile model definition language is founded on the definitions in section 3. In addition we need to define how conformance rules form *axioms* of a model, and how transitive conformance relationships can be inferred from such axioms.

***Definition 12 (profile type):*** A profile type is a pair consisting of an identifier[4], $a$ and a tuple domain $D = [T_1, .. T_n]$ and where $T_i \in$ Integer. A profile type is identified by $a$. □

***Definition 13 (conformance rule):*** A conformance rule is a predicate $p\ (t_1, t_2)$ defining a conformance relationship between two basic profile types $(a_1, D_1)$ and $(a_2, D_2)$. We use square brackets to apply parameters to a profile type instance.

$\forall t_1 \in D_1, t_2 \in D_2$: $( p(t_1, t_2) \Rightarrow x_1[t_1] \leq x_2[t_2] )$. □

The implication is the minimum requirement for the predicate. It can be viewed as a safety requirement in the sense that the rule set should at least not produce false positives. Note also that in first order logic, the conformance operator '≤' should be understood as a predicate taking two basic profile types ($a_1$ and $a_2$) as arguments.

### *5.1.1.* Computing derived rule set

The set of axioms should be the minimum needed to define all conformance relationships in the model. Some of those can be inferred directly from some axiom, others can be inferred from a combination of axioms. It is possible to derive an additional set of rules from axioms and it is convenient to do this statically since the axioms does not change at run-time. By precomputing inference which involves more than one axiom rule, conformance checking at run-time becomes equally simple and efficient for any pair of basic profiles. We may view a profile model as a *directed graph* where nodes correspond to profile types and where edges correspond to predicates of axioms. The derived rules corresponds to paths in the graph, and the set of all rules which can be derived corresponds to a *transitive closure* of the graph.

We compute derived rules from axioms by using the principle of transitivity. Given two rules which share a common profile identifier $b$:

$\forall x,y:\ p(x, y) \Rightarrow a[x] \leq b[y]$
$\forall u,v:\ q(u, v) \Rightarrow b[u] \leq c[v]$

we want to derive a rule like this:

$\forall x,v:\ p(x, y) \wedge q(u, v) \Rightarrow a[x] \leq c[v]$.

This derivation is possible if $y$ and $u$ are empty. If not, the new predicate expression would refer to unbound variables $y$ and $u$. Therefore, simply creating a conjunction would not be a general solution. We need to eliminate $y$ and $u$. If we assume that parameters $y$ and $u$ have the same domain and have the same value in all instances, such that they both can be relabelled $y$, we can try to find a predicate $r$ such that:

$\forall x,y,z :\ ( r(x, z) \Rightarrow a[x] \leq b[y] \wedge b[y] \leq c[z] ) \wedge$
$( p(x, y) \wedge q(y, z) \Rightarrow r(x, z) )$

We can then derive a new rule

$\forall x,z:\ r(x, z) \Rightarrow a[x] \leq c[z]$.

### *5.1.2.* Alternative paths and disjunctions

There may be more than one rule for a given pair of profile types. In particular this may be the case after computing the transitive closure, since there may be more than one path between two nodes in the graph representing the rule-base. Alternative rules may correspond to *disjunctions* in the sense that conformance means satisfying at least one of them. They

---

[4]In this section we use the symbols *a, b* and *c* for basic profile types, *x, y, z, u, v* and *t* for parameters, *P*, *Q* and *R* for rules, and *p, q* and *r* for predicates.

should be combined to one rule to facilitate efficient conformance checking at run-time. For instance, from

$p(x,y) \Rightarrow a[x] \leq b[y]$ and

$q(x,y) \Rightarrow a[x] \leq b[y]$.

we would derive

$p(x,y) \vee q(x,y) \Rightarrow a[x] \leq b[y]$.

## 5.2. Rule definition language

Below we define a language for defining profile models as axioms on the form given by definition 13. As a proof of concept and a tool for further exploration, we have made a compiler which takes a set of such axioms and generates conformance testing code for basic profiles. This code is used by the conformance-tester described in section 4.4.

A profile model is specified as a set of *axioms*, each stating a conformance rule. There are two kinds of rules: (1) simple conformance and (2) parametric conformance. Semantically, a simple conformance rule can be seen as a special case of definition 13 where $t_1$ and $t_2$ are empty and $p = true$.

### *5.2.1.* Simple conformance

Simple conformance rules are on the form (EBNF):

```
conformance ::= profile-name  [ "<" profile-name ] + ";"
```

One can state conformance between pairs of profile-names, or one can chain several statements. For instance the statement:

```
a < b < c;
```

is a short form for the following two axioms:

```
a < b;
b < c;
```

### *5.2.2.* Parametric conformance rules

In general, profiles may have parameters, and one may specify rules for how the conformance relationship depends on the value of the parameters.

```
conformance ::= bprofile "<" bprofile  ","  "if"  boolexpr ";"
bprofile    ::= IDENT "[" parameterlist "]"
```

The boolean expressions (*boolexpr*) would specify comparisons of numeric values and/or identifiers. Identifiers referenced in expressions *must* be found in the parameter lists of the profile declaration part. For example:

```
Res[x1, y1] < Res[x2, y2], if x1 >= x2 AND y2 >= 23;
```

Our experimental prototype supports the "<", ">", "<=", ">=" and "=" comparison operators. Only one value type is allowed: integer numbers. The boolean operators: *AND*, *OR* and *NOT* can be used to compose more complex expressions.

## 5.3. Profile model compiler

Our experimental profile model compiler is constructed using Java tools like *jflex*, *cup* and *classgen* [Cup] and consists of the following phases:

1. Transform source text into an abstract syntax tree (lexical and syntax analysis), which can be traversed and transformed using the visitor pattern [Gamma95].
2. Semantic analysis: consistency checking, computing of parameter indices, transforming the AST to list of ASTs (representing rules) and transforming boolean expressions to the *conjunctive normal form*.
3. Compute derived rules representing the transitive closure of the initial rule set.
4. If there are alternative rules for any pair of profile-types, combine by disjunction as suggested above.
5. Generate conformance testing code representing the derived rule set.

### *5.3.1.* Semantic analyser

The semantic analyser check that all variable names used in the boolean expression exists in the *'bprofile'* part (see section 5.2.2), and assigns an index to each of them which corresponds to where it was declared. It would then normalise the predicate expressions by applying the following transformations:

$\neg(x < y) \quad \rightarrow \quad x \geq y$
$\neg(x > y) \quad \rightarrow \quad x \leq y$
$\neg(x \leq y) \quad \rightarrow \quad x > y$
$\neg(x \geq y) \quad \rightarrow \quad x < y$
$\neg(p \vee q) \quad \rightarrow \quad \neg p \wedge \neg q$
$\neg(\neg(p)) \quad \rightarrow \quad p$

Now the AST[5] for the predicate expressions will be in a *conjunctive normal form*, having the following properties: (1) The operands of a AND node can be AND, OR, or COMPARE nodes, (2) the operands of an OR node can only be COMPARE nodes.

### *5.3.2. Computing the transitive closure*

We use an adapted version of a well known algorithm for computing the transitive closure of a directed graph[6]. The transitive closure algorithm inserts new edges where it finds a path of two edges between pairs of nodes. In our case we operate on rules instead of simple edges; we try to create derived rules. Figure 3 illustrates this.

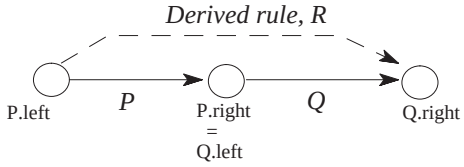


Figure 3. Rule derivation step

---

[5]The abstract syntax tree will have nodes for NOT, AND, OR and COMPARE (which corresponds to the grammar production *expr* → *expr RELOP expr*, where RELOP is a comparison operator).

[6]It is straightforward to use a DFS based algorithm for instance the one in [Sedgewick] p. 178.

Given a pair of rules *P, Q* and where *p* and *q* denotes the predicates of *P* and *Q* respectively, the method for deriving a new rule *R* is as follows:

1. If *P* is a simple rule (*p* just returns *true*), just return *r* = *q*. Similarily return *r* = *p* if *Q* is a simple rule.

2. Let *P.left* denote the left *'bprofile'* node of a rule *P*, Let *P.right* denote the right side etc. If the identifier of *P.right* equals the identifier of *Q.left*, continue. If not, return no match.

3. Combine two expressions *p* and *q* by first constructing a conjunction node: *AND (p, q)*. Then we flatten AND-trees to a list, For instance *AND(AND(p, q), r)* would be transformed to *ANDLIST(p, q, r)*.

4. For any possible pair *(p, q)* of conjuncts of this list, if *p* and *q* originate from different rules, we try to combine them using the subexpression matching rules described below. If matching is successful, the result is added to the resulting expression (by conjunction).

5. The remaining conjuncts (those which were not matched in step 4) are added to the resulting expression. Remaining subexpressions with unbound operands are removed since they are not transitively related to any other subexpression, and thus have no meaning in the derived rule.

6. Construct a new rule node *R* setting *R.left* = *P.left*, *R.right* = *Q.right* and using the predicate rule resulting from step 4. Use the semantic analyser to check that all variables in the predicate are defined in either *R.left* or *R.right*. If true, return *R*, otherwise return no match.

### *5.3.3.* Subxpression combination rules

Assume that we have a conjunction ($p \wedge q$) of subexpressions each which is either an arithmetic comparison expression (e.g. $x < y$), or a disjunction of arithmetic comparison expressions.

1. If *p* and *q* are comparison expressions, and the right operand of *p* is identical with the left operand of *q,* there is a transitive relationship between the two expressions. It is necessary to rewrite as follows when deriving a rule where the operand *y* is unbound:

$$
\begin{aligned}
x < y \ \wedge \ y < z \ &\rightarrow \ x < z \\
x \leq y \ \wedge \ y \leq c \ &\rightarrow \ x \leq z \\
x < y \ \wedge \ y \leq z \ &\rightarrow \ x < z \\
x \leq y \ \wedge \ y < z \ &\rightarrow \ x < z \\
x > y \ \wedge \ y \geq z \ &\rightarrow \ x > z \\
x = y \ \wedge \ y = z \ &\rightarrow \ x = z \\
x = y \ \wedge \ y < z \ &\rightarrow \ x < z \\
x < y \ \wedge \ y = z \ &\rightarrow \ x < z \\
x = y \ \wedge \ y \leq z \ &\rightarrow \ x \leq z \\
x \leq y \ \wedge \ y = z \ &\rightarrow \ x \leq z
\end{aligned}
$$

   etc.

2. If *p* is a comparison expression and *q* is a disjunction, we can use the distributive law to transform it such that we can apply step 1 above, for conjunctions of compare nodes.

   $p \wedge (q_1 \vee q_2 \vee \ldots q_n) \rightarrow (p \wedge q_1) \vee (p \wedge q_2) \ldots \vee (p \wedge q_n)$.

## 5.4. Example

The following example capture delay and display resolution. We illustrate the simple form of parametric profile (rule 2 below), meaning that a smaller number satisfies a higher number. We also illustrate various ways to state resolution in one, two or three dimensions (e.g. for displays) where obviously higher numbers satisfy smaller numbers. We include examples of profile for one dimension (rule 5 and 8), two dimensions (rule 7) and three dimensions (rule 9), as well as rules for comparing profiles of different dimensions (rule 6 and 10).

```
LowDelay < ModerateDelay < AnyDelay;                                    (1)
Delay[x] < Delay[y], if x <= y;                                         (2)
Delay[x] < LowDelay, if x <= 100;                                       (3)
Delay[x] < ModerateDelay, if <= 200;                                    (4)

XRes[x]  < HiRes, if x < 1000;                                          (5)
Res[x1, y1] < XRes[x2], if x1 >= x2;                                    (6)
Res[x1, y1] < Res[x2, y2], if x1 >= x2 AND y1 >= y2;                    (7)

XRes[x] < XRes[y], if x <= y;                                           (8)
3Res[x1, y1, z1] < 3Res[x2, y2, z2],
    if x1 > x2 AND y1 > y2 AND z1 > z2;                                 (9)
3Res[x1, y1, z1] < Res[x2, y2], if x1 > x2 AND y1 > y2;                (10)
```

From the above axioms we can for instance infer that `Delay[10]` satisfies `ModerateDelay` and that `Res[2000, 1000]` satisfies `XRes[500]`. In the following we give some examples of how rules are derived from other rules.

***Example 1:***

From rule 5 and 6 above we can derive

```
Res[x, y] < HiRes, if x < 1000;
```

When explaining how the rules are derived we use a slightly different notation where parameters are relabelled using the profile identifier with an index.

$XRes \leq HiRes \quad \Leftarrow (\mathrm{XRes}_0 > 1000)$
$Res \leq XRes \quad \Leftarrow (\mathrm{Res}_0 \geq \mathrm{XRes}_0)$

The predicates can be be conjoined:

$(\mathrm{XRes}_0 > 1000) \wedge (\mathrm{XRes}_0 > \mathrm{Res}_0)$

The two comparison nodes are transitively related and we can eliminate $XRes_0$ when deriving a rule:

$Res \leq HiRes \quad \Leftarrow (\mathrm{Res}_0 > 1000).$

***Example 2:***

From rule 6 and 10 we can derive

```
3Res[x1, y1, z1] < Res[x2, y2], if (x1 > x2);
```

Rule 6 and 10 are:

$Res \leq XRes \quad \Leftarrow (\mathrm{Res}_0 \geq \mathrm{XRes}_0)$
$3Res \leq Res \quad \Leftarrow (3\mathrm{Res}_0 > \mathrm{Res}_0) \wedge (3\mathrm{Res}_1 > \mathrm{Res}_1)$

The predicates are conjoined:

$(\mathrm{Res}_0 \geq \mathrm{XRes}_0) \wedge (3\mathrm{Res}_0 > \mathrm{Res}_0) \wedge (3\mathrm{Res}_1 > \mathrm{Res}_1)$

We combine the first and the second comparison node by transitivity. The third node still has an unbound variable and is removed. We derive:

$3Res \leq XRes \quad \Leftarrow (3\text{Res}_0 > \text{XRes}_0)$.

***Example 3:***

We can derive a rule for *Delay[x]* ≤ *AnyDelay*. From rule 1, 3 and 4 we see that there are two paths. Either via *LowDelay* or *ModerateDelay*:

$Delay \leq AnyDelay \quad \Leftarrow (\text{Delay}_0 < 100)$
$Delay \leq AnyDelay \quad \Leftarrow (\text{Delay}_0 < 200)$

The second alternative is widest in the sense that it returns true for a larger set of values for *x*, so it should be chosen. If combined by using a disjunction the first case will have no effect since it is fully covered by the second case. Figure 4 below illustrates the relationships as a graph.

$Delay \leq AnyDelay \quad \Leftarrow \quad (\text{Delay}_0 < 100) \ \vee \ (\text{Delay}_0 < 200)$.

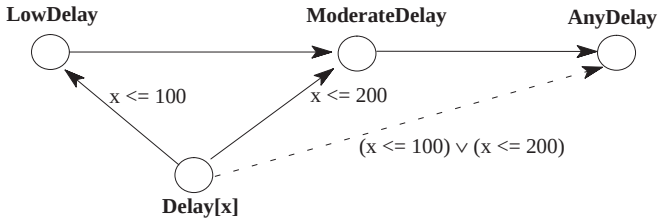


Figure 4. Rule derivation

***Example 4:***

Assume that we remove rule 5 and add some rules stating a resolution requirement in either the *x* or *y* dimension.

```
1Res[x]  < HiRes,   if x > 1000;                    (11)
Res[x,y] < 1Res[n], if x > n OR y > n;              (12)
```

From these two we can conjoin the predicate expressions like we did for the examples above:

$(1\text{Res}_0 > 1000) \ \wedge \ ( (\text{Res}_0 > 1\text{Res}_0) \ \vee \ (\text{Res}_1 > 1\text{Res}_0) )$

If we use the distributive law to transform the expression to a disjunction of two conjunctions and the transitivity combination rules for each of those, we get a derived rule:

$Res \leq HiRes \quad \Leftarrow \quad (\text{Res}_0 > 1000) \ \vee \ ( \text{Res}_1 > 1000)$ .

# 6. Evaluation and analysis

In this section we analyse the profile model with respect consistency and completeness in concrete model definitions, to interoperability between autonomous applications, and how composition can be adressed using the model. But first we briefly discuss the performance issues of evaluating profile expression.

## 6.1. Performance

The complexity of evaluating profile expressions against each other to test for conformance is discussed in section 4.4. The matching of component-sums is equivalent to a bipartite graph matching problem. Therefore it is possible to implement this with a worst case performance of $O(M^3)$ where M denotes the number of components of the component-sum in the normal form of the expressions[7]. Our naive but simple implementation has an exponential worst case complexity. The matching of sums is $O(M^2)$ worst case (where $M$ is the number of components of the sum).

It is generally hard to analyse the practical performance of expression matching independently of applications. First performance is highly influenced by the structure of the expressions to be compared, not only their length. Furthermore, this analysis is based on that M denotes the length of both expressions. Often, the lengths are different. The performance is also influenced on how often there exist conformance relationships between basic profiles used, i.e. the size and the structure of the profile conformance rulebase is also relevant.

We have implemented a prototype policy trader, using the algorithm as described above, which is a tool for experimenting on different application scenarios. The prototype is implemented in Java and contains an expression parser, conformance testing code as well as a trading service interface. To get an indication of how comparison behaves, we run the comparison algorithm on examples believed to be realistic. In the following case we have two service profile expressions and two environments to match. We assume that there is a parametric rule for *'Lat'* (which represents latency) and that we have a rule saying *that 'Storage'* satisfies *'LimitedStorage'*:

$SP_1$="(Channel+Lat[30])⊕(Server+LimitedStorage+Lat[10])"

$SP_2$="(Channel+Lat[39])⊕(Server+LimitedStorage+Lat[1])"

```
E1 = "Server+((Channel+Lat[35])⊕(Storage+Lat[10])
        ⊕(LimitedStorage+(Lat[0.5]))"

E2 = "Server+((Channel+Lat[30])⊕(Storage+Lat[10])
         ⊕(LimitedStorage+Lat[0.8]))"
```

The number of basic profile comparisons lies between 9 and 16 in this example (the number basic components is 5 and 7 in the expressions to compare, respectively). If we use the component-sum operator to add a component to the service profiles, the number of basic comparisons rise to 25, at most. This and other experiments indicate that the number of basic comparisons would typically be lower than $M^2$ (where $M$ is the number of basic components in the longest expression), though theoretically the worst case complexity would lie between $N^2$ and $N^3$ depending on the expression structure. This analysis suggest that if the length of profile expressions are kept within manageable bounds, the comparison cost would be reasonably small.

---

[7]Algorithms with complexity $O(EV)$ may at first sight seem to be better, but finding all edges (conformance relationships) is a $O(N^2)$ problem. So, in sum it is a $O(N^3)$.

| Test | | | Result | Comparisons |
|---|---|---|---|---|
| E1 | ≤ | SP1 | FALSE | 10 |
| E1 | ≤ | SP2 | TRUE | 16 |
| E2 | ≤ | SP1 | TRUE | 9 |
| E2 | ≤ | SP2 | TRUE | 16 |

Table 1. Basic comparisons of expressions

We have been able to produce a worst case behaviour for component-sums. Our experiments with application cases indicate however, that the probability of such scenarios is small. The average performance is expected to be significantly better than the worst case performance. We also observe that a critical operation in our algorithms is the comparison of basic profiles, which is expected to be cheap (see section 5). We believe that in most cases, the length of profile expressions would be reasonably low, and probably lower than the number of QoS characteristics in parameter-based negotiation schemes, since our declared conformance approach allows profiles which abstract over detailed characteristics. It is therefore likely that scalability issues would be related to the number of candidate policies to search in the trading process rather than the length of profile expressions.

## 6.2. Model consistency and completeness

When defining concrete models as described in section 5, it is possible to mistakenly define models which do not work as indended. One might make models with inconsistencies, or one may fail to completely cover the range of conformance relationships needed for the application in question. Therefore, additional guidelines and tools for checking can help to avoid or discover such problems.

In this section we propose a consistency criterion which let us detect some problems by systematic analysis. We look at some relevant cases, to discover what we can expect to be typical patterns of rule-derivation or sources of problems. We also observe that it can be useful to extend the model definition language with higher order constructs like equivalence.

### *6.2.1. Completeness*

A model is *complete* if all possible conformance relationships may be inferred from the ruleset. i.e. the implication in defintion 13 is also an equivalence. Definition 13 means that models need not be complete, but one should strive to define models which are *sufficiently* complete, mening that the conformance relationships needed by the applications are covered.

### *6.2.2. Consistency*

*Consistency* means that we would not infer contradicting results from a set of axioms. However, two rules returning different truth values for a given set of parameters are not necessarily inconsistent rules. Instead we base the notion of consistency on which *range* of profile values evaluate to true. To help finding inconsistencies we propose a simple rule which can be applied to the axioms of a model.

***Proposition (consistency):*** The predicate $p$ of any axiom, $a \leq b$, would be *implied by* any derived predicates $q$, for a ≤ b. We say that *p covers q* in the sense that the set of values which satisfy $q$ is a *subset* of the values which satisfy $p$.

$\forall x: q(x) \Rightarrow p(x)$. □

A derived rule will involve another profile type, $c$ and a rule $c \leq b$ such that all profile values which satisfy $c$ also satisfy $b$. If we assume we have a complete and consistent model, the set of profile values which (when applied to $a$ and $b$) satisfy $a \leq c \ \wedge \ c \leq b$, is a subset of the values which satisfy $a \leq b$ directly. This follows from the definition of conformance relationship (definition 3) and the definition of transitivity:

$( \sigma(a) \Rightarrow \sigma(c) \ \wedge \ \sigma(c) \Rightarrow \sigma(b) ) \ \Rightarrow \ ( \sigma(a) \Rightarrow \sigma(b) )$

We cannot prove consistency since our models are not complete. However, we can prove that inconsistencies exist. Given the argument above about consistent models, and definition 13, it can be proven that if the above proposition does not hold for a set of rules, the model is inconsistent: It is straightforward to prove that the following is true (*p,q,r,x,y* and *z* are logic expressions):

$( \neg( p \wedge q \ \Rightarrow r) \ \wedge x \Rightarrow p \wedge y \Rightarrow q \wedge z \Rightarrow r ) \ \Rightarrow \ \neg(x \wedge y \Rightarrow z).$

By definition 13 and transitivity we have

$p \Rightarrow \ a \leq b, q \Rightarrow (a \leq c \wedge c \leq b) \Rightarrow a \leq b.$

From this we can show that

$\neg( q \Rightarrow p) \ \Rightarrow \ \neg(a \leq c \wedge c \leq b \ \Rightarrow \ a \leq b).$

### *6.2.3. Examples*

In the following we discuss some cases where this rule may help us detect problems. We also consider cyclic rulesets and two patterns for rules which goes in the opposite direction: Symmetry (to address completeness) and equivalence (to realise synonym profiles). We also consider the pattern of parallel paths.

***Example 1 (inconsistent rules):*** Consider the following rules:

```
LowDelay < ModerateDelay;                          (1)
Delay[x] < LowDelay, if x <= 100;                  (2)
Delay[x] < ModerateDelay, if x <= 50;              (3)
```

Since *LowDelay* ≤ *ModerateDelay*, the requirement for a *Delay* profile to satisfy *LowDelay* should be at least as strict as for *ModerateDelay*. There is an inconsistency here in the sense that rule 3 does not cover the rule derived from rule 1 and 2. In this case, the result of combining alternative rules by disjunction is that rule 3 will have no effect. Figure 5 illustrates this by representing profile types as nodes and predicates as edges. We show derived rules by using dashed lines.

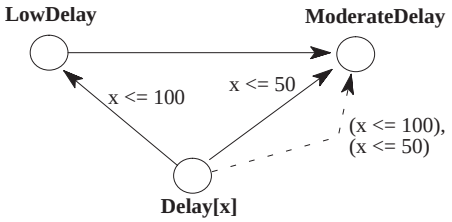


Figure 5: Derived rule

***Example 2 (cycle):*** Consider the following rules:

```
LowDelay < ModerateDelay;                          (1)
Delay[x] < LowDelay, if x <= 100;                  (2)
ModerateDelay < Delay[x], if  x > 200;             (3)
```

This results in a *cycle* in the conformance graph. A cycle is not necessarily an error, but may be problematic because of the derived rules it produces.

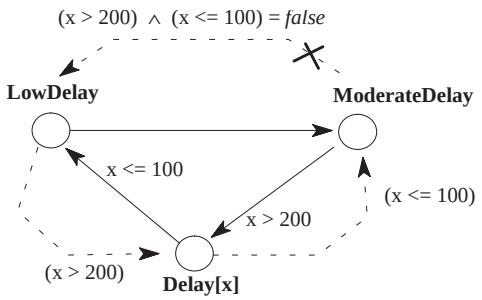


Figure 6: Conformance cycle

No new rule will be derived for *ModerateDelay* ≤ *Lowdelay* (conformance is *false*) since *x* is not bound by any of those profile types. In this case there should not be any conformance in this direction anyway. The derived rules for *Lowdelay* ≤ *Delay* and *Delay* ≤ *ModerateDelay*, are not wrong, but imprecise in the sense that they do not cover all cases where we expect conformance. Here, we would expect conformance to be true for excactly the set of values for which conformance is false in the opposite direction.

These observations suggest that cycles may lead to derivations which are not intuitively foreseen, and that some of these do not necessarily are as complete as we may wish. They may still be useful, but somewhat complicated to analyse by a programmer. Therefore, we believe that a profile model compiler should warn or inform about cycles, except when introduced by symmetric rules in opposite directions (directly between two nodes), or by defining equivalence. We illustrate the usefulness of this case in example 3 and example5 below.

***Example 3 (symmetry):*** Consider the following rules:

```
LowDelay < ModerateDelay;                          (1)
Delay[x1] < Delay[x2], if x1 <= x2;                (2)
Delay[x] < LowDelay, if x <= 100;                  (3)
LowDelay < Delay[x], if x > 100;                   (4)
ModerateDelay < Delay[x], if x > 200;              (5)
Delay[x] < ModerateDelay, if x <= 200;             (6)
```

Here we want to represent the fact that *LowDelay* satisfies *Delay[x]* if *x* is smaller than 100 (rule 2), while also *Delay[x]* satisfies *LowDelay* if *x* is bigger than 100 (rule 3 and 4). This realises a more complete connection between the two profile types than the previous example. A similar rule is made to connect *Delay* with *ModerateDelay* (rule 5 and 6). Those rules are consistent with the rule for two *Delay[x]* profiles in the sense that derived rules for *Delay* ≤ *ModerateDelay* and *Delay* ≤ *Delay* are covered by axioms

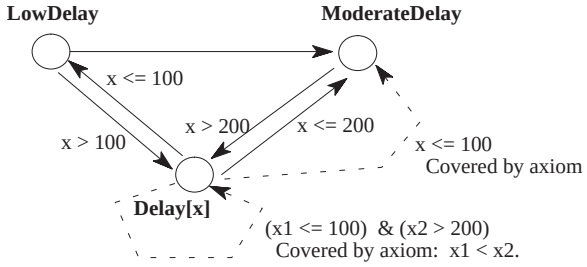


Figure 7. Symmetric rules

***Example 4 (Conflicting rules):*** Assume that we make a mistake in example 3 above and define conformance between *ModerateDelay* ≤ *LowDe*lay in the wrong direction. Now, there will be a derived rule for *Delay* ≤ *LowDelay* which is *not* covered by the axiom. In fact the comparison rules goes in the opposite direction. Observe that the derived rule for *Delay* ≤

*Delay* is not covered by the axiom as well. We have two indications that the model has inconsistencies.

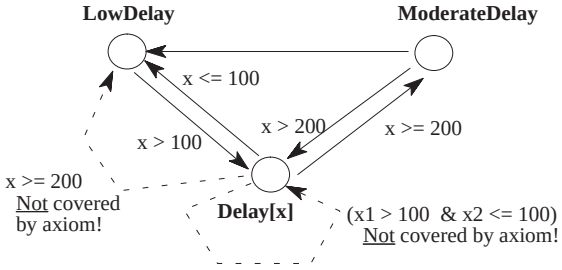


Figure 8. Conflicting rules

***Example 5 (Equivalence):*** Consider the following rules:

```
LowDelay < LowLatency < LowDelay;                (1)
Delay[x] < Latency[y];                           (2)
Latency[x] < Delay[y];                           (3)
Delay[x] < LowDelay, if x <= 100;                (4)
LowDelay < Delay[x], if x > 100;                 (5)
```

In some cases we may want to define *equivalence* between pairs of profiles, meaning that there is a conformance relationship in both directions at the same time. We limit the discussion to the cases where there is equivalence for all possible parameter values. In the figure below we indicate equivalence by thicker, double arrowed lines. It illustrates how equivalence and rule derivation effectively mirrors the conformance rules to *synonym* profile types.

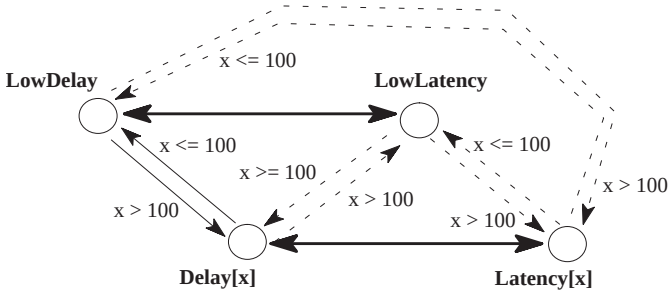


Figure 9. Equivalence

***Example 6 (parallel paths):*** Consider the following rules:

```
XRes[x] < HiRes, if x >= 1000;           (1)
YRes[y] < HiRes, if y >= 800;            (2)
Res[x,y] < XRes[x], if x1 >= x2;         (3)
Res[x,y] < YRes[y], if y1 >= y2;         (4)
```

Rules 3 and 4 define the relationship between one-dimensional and two-dimensional profiles describing (display) resolution. But they will also define a relationship between *Res* and *HiRes*, i.e. it represent a composition of two rules into one. If we follow the simple rule of composing alternative rules to disjunctions, we get

```
Res[x,y] < HiRes, if (x >= 1000) or (y >= 700);
```

Figure 10 illustrates how rules are composed. We have two parallel derived rules, which do not cover each other in any way. In this case they cannot, since they are based on different parameters in the *Res* profile. Intuitively, it looks like it is possible to define conflicting parallel rules, but to be able to detect or prove the existence of such conflicts we need to add further constraints to the model. This may be a topic for further investigation.

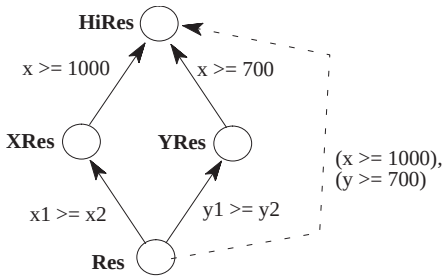


Figure 10. Parallel rules

Another possible problem of this pattern may occur if we actually wanted to compose the two rules (between *XRes*/*YRes* and *HiRes*) by *conjunction* instead of disjunction. Note that we may express conjunctive composition in an expression by using the '+' operator (c.f. section 4.1). To build conjunctive composition into the model we may be tempted to define a rule explicitly between *Res* and *HiRes*, using conjunction. However the explicit rule would then clearly be inconsistent with the two derived rules (a conjunction does not cover a disjunction or any of the individual conjuncts). In addition, it is somewhat redundant. A possible resolution is to recognise this particular pattern, and simply override any derived rules, or we may add a keyword to the profile definition language to explicitly request such overriding.

#### *6.2.4. Discussion*

It is possible to develop a notion of consistency for profile models. We have seen that there is a set of inconsistencies which can be detected and reported by a profile model compiler, to the extent it can detect if one predicate imply another.

Cycles in the conformance graph are legal, but may lead to derived rules which are not easily foreseen by the model designer and which may be imprecise. Therefore cycles are likely to be problematic, except when introduced by (1) *symmetric rules*, i.e. rules which goes in the opposite direction and where one rule is a negation of the other, or (2) *equivalence*, i.e. rules which evaluate to true in both directions at the same time. It is useful to restrict eqivalence to simple rules (always true). It is possible for a compiler to warn about cycles which do not follow those constraints. It would also be useful to extend the profile definition language with higher order constructs like an equivalence operator and an operator to define symmetric rules automatically.

Example 5 shows that there exists a type of inconsistency which we may want to resolve by letting the axiom override (disable) the derived rules. We observe that the pattern of parallel paths can be a source of some problems and it may be helpful for a profile model programmer, if a compiler could detect and report this case.

### 6.3. Interoperability

Components participating in a binding may be heterogeneous, and not necessarily designed for one single purpose. Still, there is a need to express and convey QoS and resource information among the participants. In this context, interoperability is about establishing a common understanding across different subdomains and component applications, of profile expressions to be exchanged during negotiation, and consequently, of the resulting contracts. We may distinguish between two levels of interoperability: (1) Interoperability among components in a single application sharing a single profile model. (2) Interoperability among applications using different profile models.

We assume that participants have agreeed on a shared profile model which defines the syntax and semantics of profile expressions to be exchanged. However, the semantics of profile identifiers and parameters are not completely defined, since a profile model is limited to conformance relationships. Each participant need to interpret profiles in terms of measurable characteristics or platform properties, and they should do it in a consistent way. For instance, the parameter in a profile *Delay[x]* may mean the maximum end-to-end execution

time for empty operation invocations measured in milliseconds. It is obviously a problem if the other participant understand it as a mean estimate, or using microseconds as the unit. It is also important that programmers of policies or platform components have precise information about the meaning of basic profiles, in addition to the profile model itself, to be able to implement those components correctly. The consequence of making the wrong assumptions could be that the wrong policies are selected. This could lead to failure of policies or that they violate the contract.

### *6.3.1. Integrating models*

Applications may want to interoperate and negotiate, even if they use different profile models locally. This problem is comparable with the problem of integrating schemas for federated databases [ShLa90] or different datasources described by ontologies [Wache01] (for instance in the context of the semantic web).

We can distingush between (1) *loose coupling strategies* where mappings between applications to cooperate are defined in an ad hoc way and (2) *tight coupling strategies* where one would statically define one or more federated (global) schemas, based on local schemas. [Wache01] also classify the approaches to mapping based on ontologies as follows

1. Defining *one shared ontology* (or merging local ontologies into one) for all datasources to interoperate. This approach limits the diversity of local applications.
2. *Multiple ontologies* which is somewhat similar to a loose coupling strategy, but makes it more difficult to compare local ontologies.
3. *Hybrid approaches* allowing local ontologies which share a common vocabulary. Local ontologies may be defined in terms of this vocabulary which support easier comparison and mapping. However, with this approach it is harder to integrate existing ontologies which are not based on the common vocabulary.

In the following discussion we assume that all local models to be integrated follows the model defined in this report. This simplify the problem compared to a more general case. We also assume a tight coupling strategy and that the integration is done by merging relevant parts of profile models (termed local models) into one global model. We may also define new profiles at the global level to generalise over local concepts. We may provide *mappings* such that expressions in terms of one local model can be understood in terms of global model (or another local model). Mappings should preserve authonomy in the sense that existing negotiations within a single application need not be affected by the integration. Each application would use their own profile models in platform implementations and policy implementations.

In our case, relationships between local and global profile types can be defined as *conformance rules* (see section 5). This means that a policy trading service (see section 2) would automatically perform the necessary mappings if it knows the complete integrated model (including the mapping rules). Alternatively, one may add run-time interception and translation of expressions which cross application boundaries.

### *6.3.2. Heterogeneity and conflict types*

The problems arising from data heterogeneity are well known within e.g. the federated database community [ShLa90]. In our context we are mostly concerned about semantic heterogeneity [Kim91]. Integration of models will need to resolve conflicts. According to a taxonomy of [Goh97], data heterogeneity leads to three main types of semantic conflicts: naming conflicts, scaling conflicts and confounding conflicts. Because of the strong limitations of exchanged data types in our model, the schematic and intensional conflicts (except generalization conflicts), are not directly relevant for us. The taxonomy is shown in the figure below.

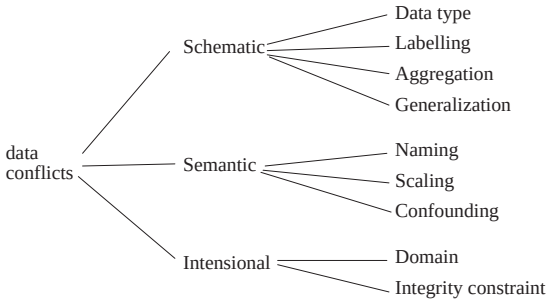


Figure 11: Conflict taxonomy

**Naming conflicts** stem from differing naming of values, typically synonyms and homonyms. In our model, this is also similar to what [Goh97] calls *labelling conflicts* since the labels of profile types also denote define possible profile identifier values. Synonyms are handled by defining equivalence (see section 6.2.3). Homonyms can be resolved for instance by prefixing names, such that they are recognised as distinct.

**Scaling conflicts** stem from the use of different units for values. For instance, using different currencies, or expressing delay in milliseconds in one place and microseconds in another. Our models do not state units and scaling explicitly. It is possible to resolve such conflicts by defining rules which do the necessary conversions. However, this will require that the language in section 5 is extended with arithmetic operators. For example if delay is given in both milliseconds and microseconds:

```
Delay[x] < us_Delay[y], if  x <= y * 1000;
us_Delay[x] < Delay[y], if  x * 1000 <= y;
```

Given the idea of extending with an equivalence operator (section 6.1) we may add scaling. From such a rule, proper conformance rules may be derived. For instance:

```
Delay[x] = us_Delay[x * 1000];
```

**Confounding conflicts** occur when information items seem to have the same meaning but have not, because they are defined in different contexts. For instance it is a conflict if a *Delay* parameter denote a maximum delay in one instance and a mean delay in another. It is not trivial to map between various interpretations for this type of conflict. Note that profiles like e.g. *Delay* may be used in different contexts, and a ruleset for *Delay* can have different contextual interpretations. Recall that the component-sum operator (section 4) can be used to express constraints in different contexts. It is possible to use the sum operator to relate a constraint to a context. For instance:

```
( Mean + Delay[40] ) ⊕ ( Maximum + Delay[100] )
```

state requirements for both mean and maximum delay (the *Mean* and *Maximum* profiles represents contexts for delay expressions). Expressions in different contexts are not comparable unless we define an additional rule defining that one context is subsumed by another. For instance we may also specify a rule *Maximum* ≤ *Mean* meaning that a maximum delay offer will satisfy a mean delay requirement, from which we can infer that *(Maximum + Delay[40])* satisfy *(Mean + Delay[40])*. Note however that this example reveals a dangerous trap. Consider a parametric profile *Bandwidth[x]* instead, where the higher parameter value satisfies a lower. Here it will be wrong to use it with the *Maximum* ≤ *Mean* contexts, since they were meant for use with a parametric profile where the smaller value would satisfy a higher. In this case it would be better to relate the context profiles to the more abstract notion of conformance relationships. For instance to define: *Best* ≤ *Mean* ≤ *Worst*.

### *6.3.3. Model consistency*

In section 6.2 above we discussed consistency with respect to conformance rules and transitivity. The types of conflicts discussed there can also arise from heterogeneity in conformance rules. It can be viewed as a *generalisation conflict* (c.f. [Goh97]) in the sense that

profile expressions subsumes each other by conformance relationships. Integrating models may create new paths of transitive conformance, and conflicting derived rules if conflicts between models are not properly resolved. The following example illustrates how inconsistency can arise from wrongly assuming that two profile types are equivalent (we assume that we can define equivalence using the '=' operator).

```
Delay[x] < LowDelay, if x <= 100;              (model 1)
Latency[x] < GoodLatency, if x <= 50;          (model 2)
Delay[x] = Latency[x];                         (mapping 1)
LowDelay = GoodLatency;                        (mapping 2)
```

The error here is that *GoodLatency* is assumed to be equivalent with *LowDelay*. It is not. This problem can be detected using the consistency test from section 6.2, i.e. the derived rule for $Delay \leq LowDelay$ should not be covered by the axiom (from model 1). This problem is easily resolved by changing mapping rule 2 to a one way conformance:

```
LowDelay < GoodLatency;                        (mapping 2)
```

## 6.4. Composition

According to section 2 we want to support composition of profile expressions. The profile model can capture basic conjunctive composition using the sum operator and composition of separate contexts using the component-sum operator. In this section we look into the pragmatics of expressing composition using these operators, and discuss what are the possibilities and limitations of our model in addressing composition.

First, it is important to observe that it is not within the scope of profile expressions themselves, to define *how* components are combined and what a composition actually results in, but rather *constraints* on how policies can combine the resources. Our approach of policy trading (c.f. section 2.1) imply that the *effect* of composition, i.e. the *relationship* between the expectation (service profile) and the obligation (user profile), is encapsulated into each policy. It is the concern of policy implementers how available resources should be used in combination to reach a goal. Given the same environmental properties, making different choices wrt. protocols and other parts of policy implementations could result in different interactions between the resources. This could again lead to different resulting QoS.

This is somewhat different from CQML [Aagedal01] where the result of composition may be specified per QoS characteristic. Three types of composition are considered: *'parallel-or'*, *'parallel-and'* and *'sequential'* and an application specific model may define functions defining characteristic specific meaning for each of them. As an example, Aagedal considers (as an example), the start-up time for a composition of two components. The total start-up time could either be the quickest one if the policy is to select the best one (parallel-or composition), the latest one if both are needed (parallel-and composition), or a sum of the start-up times, if the second one cannot be started before the first is ready (sequential composition) For instance the sequential composition of startup times would be the sum of the component startup times.

In the following we discuss how the patterns of composition in [Aagedal01] (*parallel-or*, *parallel-and* and *sequential)* can be captured by our profile model. We also discuss how to address nested composition. But first we briefly discuss some pragmatics in describing entities.

### *6.4.1. Entity descriptors*

Environmental expressions (in normal form) will typically be component sums where each element is a sum of some entity name, defining the context, plus a set of constraints to be associated with it. We refer to such subexpressions as *'entity descriptors*'. Note that the

term *'entity'* is not a syntactic category of our expression language; it is rather a way to describe pragmatics of structuring expressions.

$entity + property_1 + ... + property_n$

Expressions may be composed from a set of entity descriptors. We should normally use the component-sum operator to separate entity descriptors, like this:

$(entity_1 + properties) \oplus ... \oplus (entity_n + properties)$

The entity name may be skipped if there are no ambiguity with respect to what entity the property refer to, for instance when the expressions describe only one entity. Furthermore, expressions can be sums of entity names only, meaning that all those entities are (or is required to be) available, for instance *"network + CPU + memory"*. The existence of entities may be viewed as properties, but observe that the choice of this form must be consistent among the negotiating participants[8]. For instance there will be no conformance relationship between $"A + B + C"$ and $"A \oplus B \oplus C"$. If other properties are to be associated with any of the entities, they must be separated by using a component-sum, like in following expression:

$(Network + HighBW) \oplus CPU \oplus (Memory + HighCapacity)$

In contrast, the expression

$Network + HighBW + CPU + Memory + HighCapacity$

would not express the same fact. In this example we cannot longer tell which entity properties like *'HighBW'* are associated with, i.e. what context they appear in.

### *6.4.2.* Parallel-or composition

If there exist no dependencies between (possibly equivalent) components, the composition can be classified as parallel. *Parallel-or* composition corresponds to the case where one of the components is selected for use.

In the context of policy trading where SP denotes the service profile of a potential policy and where E denotes the description of the environment, consider the example of combining properties of communication channels (assume that $SuperHighBW \leq HighBW$).

$SP = Channel + SuperHighBW, \quad E = Channel + (HighBW \oplus SuperHighBW)$

The policy requires one *SuperHighBW* channel. The environment offers two channels, one of them satisfy the requirement, which is sufficient to result in a match. Note that if a policy implementation for example uses one channel, and the environment supports two or more channels, the implementation must be able to select a suitable channel from the available ones.

### *6.4.3.* Parallel-and or sequential composition

*Parallel-and* composition mean that components are unrelated but all are used in combination. For the example of startup time, this may mean that the longest time must be used as the result, since the components can startup in parallel, but we need all to finish. For certain capacity characteristics, parallel-and may mean that the result is the sum of the component capacities, e.g. for processors which can run in parallel without need for synchronisation. *Sequential* composition mean that there are dependencies between constituent

---

[8]This particular case could be viewed as a possible conflict which need to be resolved to provide interoperability (c.f. section 6.3).

components, for instance that one cannot run before the other has finished. For the case of startup time this would typically mean to return the sum of component times.

Consider the following expressions.

*SP = Channel + (HighBW ⊕ HighBW),  E = Channel + (SuperHighBW ⊕ HighBW)*

The policy expresses the need for the satisfaction of two separate channel entities by using the component-sum operator, and only environments with at least two channels will satisfy it. This is the way to express parallel-and or sequential composition. Observe that the expression pattern is the same for those two cases. In essence, policies express the need for multiple resources. How these are combined (parallel, sequential or something in between) is an implementation issue. However, environment descriptors may need to include additional constraints on how components can be combined by policy implementations, for instance they may explicitly disallow parallel composition. Such constraints may be specific to component types or application domains. The only way to express them, is to define basic profile types denoting composition constraints or location, and to use these in profile expressions. Figure 12 illustrates how we may define a profile model for channel entity profiles:

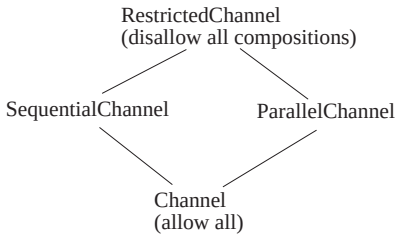


Figure 12. Constraints on composition

For instance, a pair of channels which can only be combined sequentially may be described like this:

*E = SequentialChannel + (SuperHighBW ⊕ HighBW)*

Observe that a profile which denotes a restriction of composition compared with another profile, would be a *superprofile* of that other profile, since a requirement for a more capable channel is a stronger constraint on the environment. A requirement for a resource which can be used in any way, is a stronger requirement than a resource with restrictions on the use.

It may also be necessary to state the *roles* of each component or the context they are expected to be used in, as additional constraints. Consider the following expression:

*E = (SeqSourceChannel+ SuperHighBW) ⊕ (SeqSinkChannel + HighBW)*

Here the basic profile types *'SeqSourceChannel'* and *'SeqSinkChannel'* describes channels which could only be used with the source and the sink of some stream, when used in sequential composition.

### *6.4.4.* Nested composition

An entity descriptor (c.f. section 6.4.1) may be a property of another entity. For instance we could have a component-sum of two entity descriptors describing the client and the server respectively. Each of those could contain a set of sub-entities to describe local services and resources at each side. Given an entity $e$ with properties $p_1$, $p_2$, ... $p_n$, if at least one property $p_i$ is a entity descriptor $f$ with its own properties $q_1$, $q_2$, ..., $q_m$, it is necessary to separate this part from the other properties of $e$ by using the component-sum operator. Expressions should be on the form:

$e + ( (p_1 + ... + p_n) \oplus (f + q_1 + ... + q_m) )$

For instance if we have a channel with the property *'HighBW'*, which also have a special sub-channel with the property *'SuperHighBW'*, it cannot simply be expressed:

*Channel + HighBW + (SubChannel + SuperHighBW)*

According to the associative law the parentheses does not mean anything, and the sum *'HighBW + SuperHighBW'* is actually equivalent to just saying *'SuperHighBW'*[9]. Thus, this expression does not capture that the outer and inner component have different bandwidth constraints. Instead we need to express this as a component-sum of the outer and the inner components:

*Channel + ( HighBW ⊕ (SubChannel + SuperHighBW) )*

for which the normal form is:

*(Channel + HighBW) ⊕ (Channel + SubChannel + SuperHighBW)*

### *6.4.5. Limitation of nested composition*

Consider the example:

*E = (Server + ((Disk + LowLatency) ⊕ HighPerformanceCPU)) ⊕*
*(Client + LowPerformanceCPU)*

It may seem like expressions can express nested (or hierarchical) composition. In this example a disk and a CPU component appear in the context of a server. But observe that the nesting hierarchy is flattened when transforming to the normal form. However we do not lose all nesting information this way. For instance *'(a+(b⊕c)) ⊕ d'* is equivalent to *'(a+b) ⊕ (a+c) ⊕ d'*. If any constraints are associated with a containing context, any subcontexts would inherit those constraints. This follows from the distributive law. An identification of the context (i.e. an entity name) should also be viewed as nothing more than a constraint. One could view normalisation as writing the expression as a component-sum of leaf constraints along with (the constraints of) the contexts they appear in. The top level context would appear repeatedly for each operand.

This means that our model captures nested composition only in a limited sense, i.e. the component-sum separation cannot actually be nested. To see this clearer, consider the following example:

*SP = (Server + ( (Disk+LowLatency) ⊕ HighPerformanceCPU)*
*E = (ServerA + Disk + LowLatency) ⊕ (ServerB + HighPerformanceCPU)*

In *SP*, the profile *'Server'* seems to occur once as a shared context for the two sides of the component sum and it may look like the S*P* expression describe one single server instance. This is not necessarily true; it may be two as well. If *'ServerA'* and *'ServerB'* are subprofiles of *'Server'*, *E* will satisfy *SP*. Strictly speaking, we cannot tell from this expression if *'ServerA'* and *'ServerB'* describe separate containing server instances or just two components (disk, CPU) sharing the same server instance.

We may want to express that the separate components describe the same instance or separate instances. With our current profile model the only way to do this is to use basic profiles which identify particular instances. This is obviously not a flexible or scalable solution, since the instances must be known a priori.

---

[9]If we assume that $SuperHighBW \leq HighBW \leq NormalBW$.

We see from this that our model has significant limitations in expressing nested composition. This should be a case for further work in extending the profile model. A possible approach is to introduce labels to indicate instances. If we for instance want to describe an environment with two separate environment and where there should be separate instances of the entity *'Server'* associated with each component.

$SP_1$ = *(x:Server + some-cpu)* ⊕ *(y:Server + some-storage)*
$SP_2$ = *x:Server + (some-cpu* ⊕ *some-storage)*
*E = (a:Server + some-cpu)* ⊕ *(b:Server + some-storage)*

Here, *E* would satisfy $SP_1$ but not $SP_2$. If we say *'x:Server (some-cpu* ⊕ *some-storage)'* we clearly state that the CPU and the storage must be associated the same server instance. Note that we suspect that this approach may increase the computational complexity of conformance checking and should be carefully evaluated before making any conclusion.

# 7. Concluding remarks

In this report we define a language for dynamic QoS expressions which can be evaluated at run-time for conformance. We define how expressions can be constructed from atomic QoS statements termed *'basic profiles'* using composition operators. Two such operators are defined: The *sum* ( '+' ) which corresponds to simple conjunction and *component-sum* ('⊕') which assume that the operands denote properties of separate environments and therefore must be satisfied separately. To define the semantics of expressions using both operators, we define a distributive law and a normal form. Based on those rules, as well as conformance rules for pairs of component-sums or sums, algorithms for conformance checking any pair of expressions can be developed.

Concrete models define the basic profile space and explicitly establishes conformance relationships between basic profiles. They are typically defined for specific application domains and can be defined as *rule-bases*. These are essentially sets of axioms from which we can infer conformance between any pair of basic profiles. From the axiom set we may derive a full rule set, covering any pair of profile-types for which there may be conformance. Such a rule set can be directly mapped to executable code which allows efficient conformance checking at run-time. As a proof of concept we implemented a profile model compiler (which also has been useful in analysing consistency and performance issues). The compiler performs a basic semantic check of rules, compute derived ruleset by using a transitive closure algorithm and outputs conformance checking code.

Our analysis shows that there exist some types of inconsistencies which are detectable by a profile model compiler. It follows from the model that an axiom should *cover* (its predicate should be implied by predicates of) any derived rules between the same pair of profile types. Furthermore, some problems can be detected if additional constraints are introduced. We also observe that the pattern of parallel paths can be a source of some consistency problems. In that case it would be useful if an axiom could could be set to override any parallel derived rules. Cycles in a conformance graph may lead to derived rules which are not easily foreseen by the model designer.

The problem of supporting interoperability between applications using different profile models is comparable with the problem of integrating datasources described by different ontologies. In analysing interoperability we see that certain conflict types are relevant when integrating profile models from different sources. Domain specific models define profile names and semantics only. Therefore, conflict resolution is mostly limited to semantic conflicts. The sources of problems comes from differences in how expressions are interpreted locally (scaling and confounding conflicts), and in how names and rules are defined (naming, and generalisation conflicts). The latter is partly detectable. In our context it is possible to define mappings and possibly merge models by defining additional conformance rules. In particular we can use equivalence to handle synonyms.

When discussing composition, it is important to note that profile expressions is used to express requirements or constraints on composition rather than what composition results in. It is up to a policy implementation how available resources are combined. We observe that we may need to add profile types to represent constraints on how resources can be composed. We also observe that the ability to express nested composition is limited because of the distributive law. A particular problem which results from this is that it is complicated to express properties which apply to different instances of a profile type.

## 7.1. Cases for further work

Our analysis suggests that we investigate some extensions to the model. Some are higher order constructs which can be defined in terms of the core model. Others may be more fundamental. We believe that the following issues should be cases for further investigation:

- An equivalence operator '=' which is straightforward to define in terms of two conformance rules. We have shown some cases where this is obviously useful.
- A symmetry operator where one conformance rule can be defined in relation to another meaning that there is conformance excactly for the parameter values where it is not conformance in the opposite direction.
- Overriding of any derived rules (in the case of parallel paths).
- Arithmetic operators in rules to support scaling.
- The core profile model is designed with conformance checking and composition in mind. This is however not necesssarily easy to read by humans and we may need some composition pragmatics in addition to the model. We may benefit from defining some higher order syntax, for instance to simplify expressing entity descriptors with constraints like discussed in section 6.3.1.
- Variables (labels) in expressions to distinguish between separate instances like discussed in section 6.3.7.

# References


Abadi93 M. Abadi, L. Lamport, *Conjoining Specifications*, *Digital Systems Research Center Report 118*, Palo Alto.

Abadi94 M. Abadi, L. Lamport, Open Systems in TLA, *Proceedings of ACM Symposium on Principles of Distributed Computing*, August 1994, p. 81-90.

Aagedal01 J.Ø. Aagedal, *Quality of Service Support in Development of Distributed Systems*, Ph.D. Thesis, University Oslo, 2001.

Bearman93 M.Y. Bearman, ODP Trader, In Proceedings of ICODP'93, Berlin 1993, pp. 13-16, 1993.

CUP *CUP: LR Parser Generator in Java*. http://www2.cs.tum.edu/projects/cup/

Frølund98a S. Frølund, J. Koistinen, Quality-of-Service Specification in Distributed Object Systems, *Distributed Systems Engineering Journal. Vol. 5. 4, pp. 179-202*.

Frølund98b S. Frølund, J. Koistinen, Quality of Service Aware Distributed Object Systems, *Hewlett Packard Software Technology lab. report: HPL-98-142*.

Gamma95 Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison Wesley 1995.

Goh97 C.H. Goh., *Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Sources*, Ph.D. thesis, MIT, 1997.

Hanssen98 Ø. Hanssen, F. Eliassen, Towards a QoS aware Binding Model, In *Proceedings of SYBEN '98*, Zurich, Spie press, 1998.

Hanssen99 Ø. Hanssen, F. Eliassen, A Framework for Policy Bindings, In *Proceedings of Distributed Object and Applications '99*, IEEE press, 1999.

Hanssen00 Ø. Hanssen, F. Eliassen, Policy Trading, In *Proceedings of Distributed Objects and Applications '00*, IEEE press, 2000.

ISO97 *ODP Trading Function,* Report: ITU-T X.950 - ISO/IEC 13235.

ISO95 *QoS - Basic Framework*, Report: ISO/IEC JTC/SC21, N9309, 1995.

Kim91 W. Kim, J. Seo, Classifying schematic and data heterogeneity in multidatabase systems, *IEEE Computer, 24(12)*: 12-18, 1991.

Wache01 H. Wache. T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, S. Hübner, Ontology-Based Integration of Information - A Survey of Existing Approaches, In *Proceedings of IJCAI-01 Workshop, Ontologies and Information*, pp. 108-117, Sharing, 1997.

Sedgewick03 R. Sedgewick, *Algorithms in Java*, third edition. Part 5., Pearson 2003.

ShLa90 A. Sheth, J. Larson, Federated Database Systems for managing Heterogeneous and Autonomous Databases, *ACM Computing Surveys, Vol. 22, No. 3*, September 1990.