The Design of GCCL: a Generalized Common Contract Language

Jacques Malenfant, Noël Plouzeau, Jean-Marc Jézéquel

▶ **To cite this version:**

Jacques Malenfant, Noël Plouzeau, Jean-Marc Jézéquel. The Design of GCCL: a Generalized Common Contract Language. [Research Report] RR-4502, INRIA. 2002. inria-00072086

HAL Id: inria-00072086

https://inria.hal.science/inria-00072086

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Design of GCCL: a Generalized Common Contract Language

Jacques Malenfant — Noël Plouzeau — Jean-Marc Jézéquel

N° 4502

Juillet 2002

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4502--FR+ENG

INRIA RENNES

The Design of GCCL: a Generalized Common Contract Language

Jacques Malenfant* , Noël Plouzeau , Jean-Marc Jézéquel

Thème 1 — Réseaux et systèmes
Projet Triskell

Rapport de recherche n° 4502 — Juillet 2002 — 19 pages

Abstract: Following its inception in Eiffel by Meyer and its diffusion to other environments (e.g., the standardisation of OCL as part of UML), Design by Contract now faces a major challenge in component-based software engineering (CBSE). Compositional reasoning about system properties from component ones has been recently asserted by the SEI as the "key technical challenge" of CBSE, and contracts as a "key technical concept to support this vision". To live up to these expectations, DbC must tackle extra-functional properties of components and support Klein's "architectural-based attribute reasoning." Besides adopting the necessary new concepts, a good contract language must provide for abstraction, application, composition and scoping mechanisms in such a way to be used from modeling in UML to programming in standard languages (as seamless extensions to Java, C#, C++, Eiffel, and so on) through execution on traditional operating systems with minimal middleware additions. This paper examines conceptual foundations and design decisions to propose GCCL, a novel open generalized common contract language. GCCL is meant to be compatible with existing DbC (sub)languages, and especially OCL.

Key-words: programming by contract, specifying and verifying and reasoning about programs, languages, measurement, performance.

This work has been supported by the European Contract QCCS.

* également Université de Bretagne sud.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 —Télécopie : +33 2 99 84 71 71

La conception de GCCL : Generalized Common Contract Language

Résumé : Suivant ses débuts en Eiffel par Meyer et sa diffusion à d'autres environnements (comme par exemple la standardisation d'OCL comme partie de la norme UML), la programmation contractuelle fait aujourd'hui face à un défi majeur dans le domaine du génie des logiciels à base de composants (GLBC). Le raisonnement compositionnel sur les propriétés des systèmes à partir des propriétés des ses composants a été identifié par le SEI comme le "défi technologique clé" du GLBC, et les contrats comme un "concept technologique clé pour supporter cette vision." Pour répondre à ces attentes, la programmation contractuelle doit s'attaquer aux propriétés extra-fonctionnelles des composants et supporter le raisonnement sur les attributs basé sur l'architecture développé par Klein. Outre l'adoption des nouveaux concepts devenus nécessaires, un bon langage de contrats doit fournir des mécanismes d'abstraction, d'application, de composition et de portée de telle manière à ce qu'il soit utilisable depuis la modélisation en UML jusqu'à la programmation dans un langage standard (comme extension fluide à Java, C#, C++, Eiffel, etc.) et l'exécution sur des plates-formes standard avec un minimum d'additions au niveau de l'instersticiel. Ce rapport examine le fondations conceptuelles et les choix de conception pour proposer GCCL, un nouveau langage de contrat ouvert, général et commun. GCCL se veut compatible avec les langages de contrats existants, et tout particulièrement OCL.

Mots-clés : programmation contractuelle, spécification, vérification et raisonnement sur les programmes, langages, mesures, performance.

1 Introduction

Following its inception by Meyer in Eiffel [15] and its diffusion to other environments (e.g., the standardisation of OCL as part of UML [19]), Design by Contract™ (DbC) now faces a major challenge in component-based software engineering (CBSE). In a recent feasibility study of CBSE, the Software Engineering Institute of the Carnegie-Mellon University asserts that the "key technical challenge facing CBSE is to ensure that the properties of a system of components can be predicted from the properties of the component themselves" and it identifies contracts as one of the "key technical concept to support this vision" [2]. The overall goal is to provide for a component certification process upon which a real market can be established. This vision pursues ideas already developed in the CBSE commmunity, for example in Szyperski's seminal book [20].

The SEI concurs with Szyperski [20] and Wang [21] in considering a component as an implementation as well as an architectural abstraction, and as such "*an opaque implementation of functionality [(what it does)] subject to third party composition and conformant with a component model [(how it interacts with the outside world)]*" [2]. When executed, components also exhibit observable or measurable quality attributes, such as performance, security, safety, resource consumption, and so on. A lot of applications are subject to stringent requirements on at least some of these quality attributes. Hence, components must be selected not only for their functionality but also for their ability to provide the required quality. They must then be composed in such a way that the overall requirements are provably met.

To date, most component models express dependencies using interfaces that include only syntactical aspects and typing. Experience in composing complex software components has shown that dependencies extend way beyond these aspects. Indeed, to reason about properties of the composed systems from opaque implementations of components, interfaces must provide information about these properties. Component models therefore strive for a way to express a large variety of aspects, including extra-functional ones.

To express these dependencies, we have proposed to make components contract-aware [4], but for DbC to face the challenge, contracts must be extended so to include such extra-functional aspects. This means first of all to include new concepts to "talk" about extra-functional attributes of software and new kinds of constraints over these attributes.

Of course, work has been done in the past to define constraint expressions over specific quality attributes and reason about them. GCCL aims at building over this body of knowledge. But designing a successful general language is not confined to issues of quality attributes and constraints. Key concepts in language design are abstraction, application, composition and scoping mechanisms that provide clear and powerful means to define, place, group, check and reason about constraints at all kinds of binding times (design, coding, testing, deployment, run time, ...). GCCL addresses this broad challenge. Its three major goals are to:

1. provide means for expressing functional and extra-functional requirements and for defining their semantics, as well as for supporting their decomposition onto the architecture designed for applications;
2. provide means to ascertain the correctness of the composition of a set of components; and,
3. provide means to check at run time that components and the underlying run-time system meet their contracted obligations.

The rest of this paper is organized as follows. Section 2 presents the conceptual landscape for GCCL while Section 3 models the general language concepts included in GCCL and explains the key design decisions and semantic features of the language. Section 4 provides programming examples. Related work are then examined and a conclusion summarizes GCCL's current achievements and future work.

2 Conceptual landscape

In this section, we introduce and define the general concepts that surround and contribute to the design of GCCL, indeed focusing on those related to contract languages per se.

2.1 Design by Contract concepts

Following a tradition in axiomatic semantics and formal specifications of programs, DbC associates *contracts* to classes and methods in the form of three specific kinds of *constraints*: *invariants*, *pre-* and *postconditions*. Constraints are boolean expressions relating values of instance variables for invariants, parameters and instance variables for pre-conditions as well as all these plus results of methods for postconditions.

The notion of contract stems from the vision of preconditions as conditions to fulfill by the caller such that the callee will return results satisfying its postconditions, thus establishing mutual obligations between the caller and the callee. Similarly, class invariants impose obligations on methods. Methods beginning their execution in an object state that satisfies the invariant must leave the object in a state satisfying it again upon termination.

Contracts help developers to express explicitly conditions under which portions of code can be guaranteed to run without errors. But also, instrumental to the DbC approach is the run-time checking of contracts. At least until field deployment of the application, but sometimes also during exploitation, constraints are checked during program execution. Contract violations guide maintainers towards bugs, since violation of a precondition leads to blame the caller of the method detecting the violation, while violation of a postcondition or of an invariant leads to blame the called method itself (modulo a correct and complete definition of violation semantics, see [8]).

Less considered but still possible, contracts are assertions in the sense of the more formal axiomatic approaches.[1] Hence, it is possible in many simple cases to prove the correctness of the code with regards to these assertions, i.e. that the code inherently satisfies them. Much of the power of DbC lies in this relationship to more formal methods, with the motto that what you cannot prove formally, you can validate through testing (or running) the application while checking the assertions. In that sense, DbC is often seen as a semi-formal method for enhancing the quality of software.

2.2 Quality of service concepts

Quality of service (QoS) has gain much attention in the field of distributed multimedia where it covers essentially the notion of end-to-end guarantees associated with the communication of multimedia contents over networks and their processing on computing nodes [1]. A considerable amount of work has been done in this area to introduce models and architectures supporting such QoS guarantees in order to supersede traditional best-effort IP networks, and to enable mobile computing [6]. This trend has percolated to standard architectures for open distributed processing, namely TINA [11] and more recently CORBA [16]. This narrow vision of QoS is currently generalized to extra-functional properties of distributed systems [9, 3], as QoS becomes subject to research in the middleware community [12].

QoS includes various aspects, which we now define in a very general manner (adapted from [1], [6] and [12], to which we refer the reader for more information). QoS defines *extra-functional characteristics* of a system, affecting the perceived quality of the results. Such characteristics are often organized into *categories*, such as timeliness or reliability.

QoS management is the necessary *supervision* and *control* to ensure that the desired quality of service properties are attained and sustained. It applies both to *continuous* and *discrete interactions*. *Static* management aspects relate to properties and requirements that remain constant throughout some activity. They include:

specification: definition of QoS requirements or capabilities,

negotiation: reaching an agreed specification between all parties,

admission control: comparison between the required QoS and the capability to meet them, and

resource reservation: allocation of resources so to ensure the meeting of the QoS requirements.

Dynamic management aspects relate to responses to changes in the environment. They include:

monitoring: measuring the QoS actually provided,

policing: ensuring that all parties adhere to QoS contracts,

maintenance: modification of system parameters to maintain the QoS,

renegotiation: modification of the QoS contracts, and

[1] The theory-inclined reader will also recognize the continuum between typing and contracting. The Curry-Howard isomorphism acknowledges the similarity between typing rules of programming languages and inference rules of predicate logic. In a sense, pre- and postconditions introduce an extension to the type system of the host programming language by providing the full power of first-order logic to specify the types. Obviously, more powerful type systems in that line could be adopted, but the static verification of types would incur a formidable increase in complexity (and be non-decidable in many cases). DbC does not solve this problem, but rather responds to it by simply defering the verification to the run-time, pretty well like most programming languages that defer at least some of the type verifications to run-time (e.g. latent type systems, as in Scheme, which are indeed quite different from untyped systems).

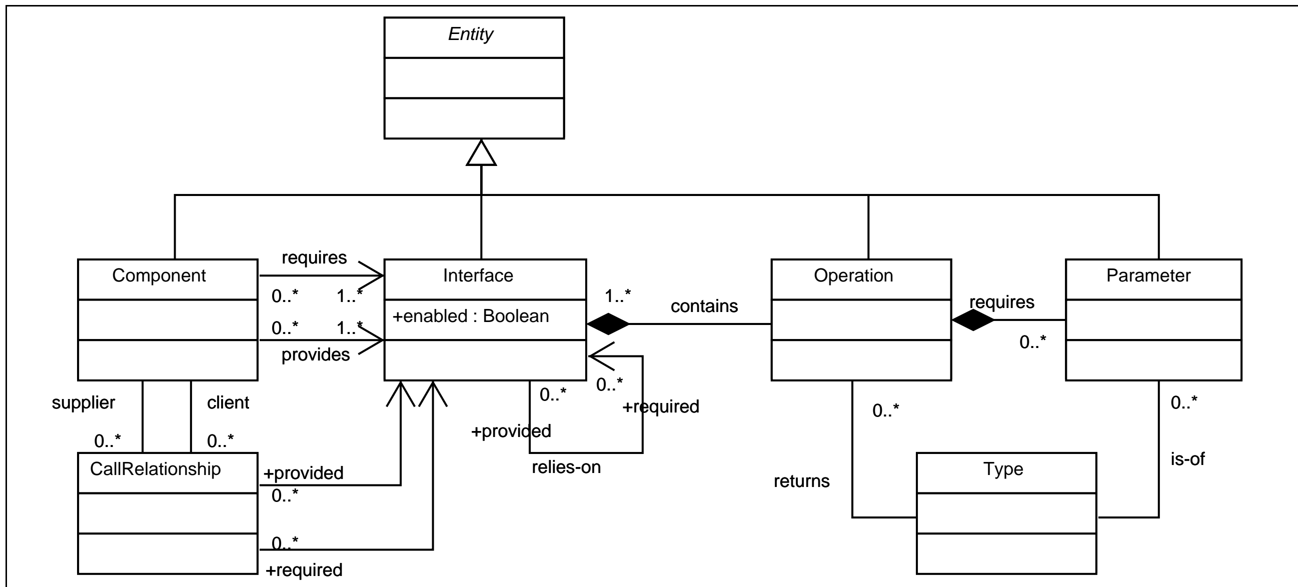

Figure 1: Entity model

adaptation: modifications in applications to maintain the QoS.

Basically, *QoS contracts* are means to specify QoS requirements and capabilities. These specifications can then serve as basis for all the remaining management aspects of QoS, provided that contracts are sufficiently *declarative*. Those are the core aspects that must be dealt with in a QoS contract language, and in this paper we will restrict ourselves to this vision, leaving other management aspects to the underlying QoS architecture built over system and network services. However, a wide variety of *management policies* can be used in the various aspects. Negotiation, for example, can take into account objective functions that differ from user to user. The same kind of variance can be seen in other management aspects where they can be more or less seen as metalevel decisions. Future extensions of GCCL will include such aspects.

An important issue in specification is the ability to establish requirements and capabilities in an application-oriented way [12]. Programmers can deal with metrics applied to programs and user perception, but much less with system-oriented measures. Ideally, QoS architectures ensure the necessary translation between application-oriented constraints and system-oriented measures. Explicit mechanisms to do this mapping are often needed since it is almost impossible to cater for all possible application-oriented intentions. Another important aspect is the ability to use qualitative and quantitative specifications in a seamless fashion. In fact, good specification languages for QoS should allow programmers to refine their requirements and capabilities from high-level application-oriented qualitative constraints to low-level system-oriented quantitative ones. This refinement process ability is indeed central to real scale software design.

2.3 Component concepts

Extra-functional contracts are meant to put obligations on the code or provided services. To do so, they will attach assertions to specific entities. We now introduce the necessary concepts providing a suitable model of program upon which contracts can be defined. The goal of designing a contract language applying to a wide variety of modeling and programming languages suggests to provide it with a reference model and mappings to hosts languages such as UML, Java and C#.

To define this reference model, standards like RM-ODP, TINA, and CORBA, as well as the UML metamodel are natural guides. But in general, a large number of distinct entities can be subject to contracts. GCCL is inspired from these standard reference models, but is currently restricted to contracts put on components. The following considers only the necessary concepts to support this current vision. Future extensions of GCCL will cover more of the potential contractable program entities.

Software systems are built from *program entities*, like classes, methods, formal parameters, statements, variables, and so on and so forth. Program entities have manifestations at run-time that we will therefore call *run-time entities*. Program entities obey a *lifecycle* specific to each kind of entity. The lifecycle of an entity

expresses sequences of events and states through which this entity and its run-time manifestations can evolve over time. For example, objects are created, initialized, repeatedly activated by messages, and then garbage collected when no longer needed. Methods are repeatedly activated, started, executed, suspended, resumed, terminated, returned from.

Components, as we have seen, are opaque implementations of functionality offering services through *provided interfaces* given that other services are furnished to them through *required interfaces* [2]. Interfaces are sets of *operations*, themselves defined by a *name*, a *return type* and a list of *parameters* with their respective types. Components may offer the same logical service using different subsets of its provided interfaces, each one using its own subset of required interfaces. A dependence relationship, called `relies-on` in our entity model, is established between provided interfaces and its required ones. The semantics of this relationship is that a provided interface is enabled if all of its required interface are enabled by a connection to the provided interface of some other component through a call relationship.

Albeit often not explicitly mentioned, components obey to *component models*, which define the set of component types, their interfaces and their allowed patterns of interaction. *Component frameworks* are the run-time systems providing the necessary services to support and enforce component models. Component frameworks have implementations over different *platforms*, and eventually run on specific *hardware*, each of which carrying its own contribution to the overall extra-functional properties of the whole applications.

Applications are built through *explicit binding* of components, i.e. *call relationships* connecting required interfaces of some components to *conformant* provided interfaces of others. Conformance in current component models often reduces to syntactic conformance of signatures and types, while the component model itself implicitly caters for many dependencies by imposing its own rules. Explicit binding is another very important concept in our context, as QoS is often recognized as an aspect of binding [12, 5]. Connecting or binding components can occur at several different points in their lifetime: application design time, coding time, deployment time or even run-time (when applications are dynamically reconfigured), within which one can also distinguish bindings that last throughout the execution, from bindings that can be redefined occasionally and from bindings that can be done only for one call and then vanish. Binding times are crucial to the semantics of contract enforcement.

The figure 1 presents the GCCL entity model as a UML class diagram for entities upon which contracts can currently be defined. A component provides one or more interfaces, and also requires one or more interfaces. Each provided interface may rely on a subset of the component required interfaces. An interface contains one or more operations, which in turn requires parameters. Components participate in call relationships (`Call-Relationship`) by being either the supplier or client of some operation(s). A call relationship therefore links a required interface of some client component to a provided interface of some supplier component.

This model of contractable entities is abstract since it does not depend on any particular target language. To connect it to a given context, we would need to define bindings from our abstract notions to concrete ones in this target context. For instance, we could embed this model into the UML meta-model, using UML extension mechanisms (i.e. Stereotypes) to connect it to existing meta-model classes (actually, we chose the class names from the UML Core to make this connection particularly easy in the case of UML).

2.4 Constraint programming concepts

To provide means for ascertaining the correctness of component compositions with regards to extra-functional requirements, a contract language must define a *conformance* relation between provided and required contracts. Conformance is an important general concept since it enables the comparison of contracts based on constraint satisfaction rather than merely on equality between specific values [10].

Constraint satisfaction (over integers or reals) has been studied for decades in the mathematical programming and operation research community. More recently, the constraint programming paradigm has been developed with the goal to solve programming problems by merely stating *what* constraints solutions must satisfy, and to leave to the underlying language implementation the *how* these solutions can be computed.

Constraint programming languages have two major parts: a set of programming constructs allowing programmers to define and add constraints to so called *constraint stores*, and *constraint solvers* to simplify and find solutions statisfying the set of contraints [13].

Constraint solvers are defined for specific *domains of computation* and expressible *constraints*. For example, if the domain is the real numbers and constraints can use the usual arithmetic operators $+$, $\times$ and comparison operators $=$, $<$ and $\leq$, then we obtain the domain of arithmetic over real numbers, noted $\mathbb{R}$. If we omit the operator $\times$, then we get the domain of linear arithmetic over real numbers, noted $\mathbb{R}_{Lin}$, and if we omit the operators $<$ and $\leq$, we obtain the domain of linear equations over real numbers noted $\mathbb{R}_{LinEqn}$. Constraint

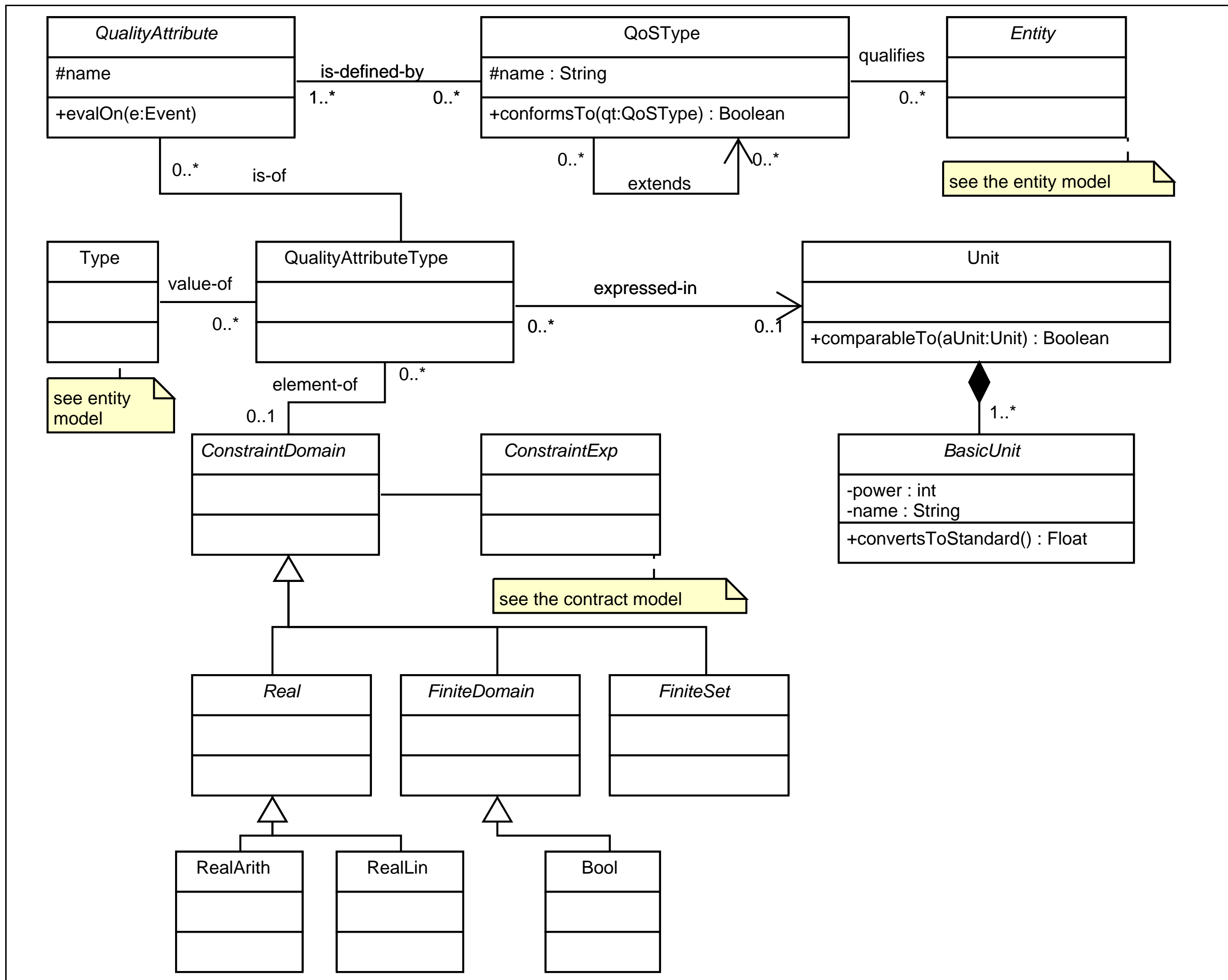

Figure 2: Attribute model

solvers have been developed for several domains to date, including linear and quadratic arithmetic for reals, terms and rational trees, finite domains, semi-rings (including booleans, and finite sets with $\in$ constraints), etc.

The constraint solver uses an axiomatization of (some) properties of the domain and limitations on the class of expressible constraints to implement the two following tests:

- *consistency* or *satisfiability*, i.e. the fact that a set of constraints has at least one feasible solution;
- *implication* or *entailment*, i.e. the fact that a constraint is implied by another one and thus can be neglected.

Together with more specific operations on constraints, solvers proceed by simplifying the set of constraints (deleting implied constraints and combining constraints) and then by enumerating the solutions or reporting unsatisfiability.

Constraint domains and solvers provide an excellent general foundation for the selection of assertions over extra-functional properties expressible in contracts and for the definition of the semantics of conformance between contracts. Moreover, constraint simplification can also be used to synthesize overall requirements of systems as yet to be satisfied constraints or to provide bounds for the negociation of appropriate levels of QoS. This can be done at binding-time by propagating available resource levels towards contracts between components or by retro-propagating contract requirements onto the hardware in order to dimension the overall hardware/software system or to perform admission control and resource allocation.

3 GCCL design

Given the above conceptual landscape, we now list the key concepts supported by GCCL.

3.1 Introduction of quality attributes

DbC involves equiping a program with assertions that can be either statically proved or dynamically monitored. To support that, concepts of QoS must be applied to program entities and an appropriate assertion language must be defined to provide a model for QoS specification. We now introduce such a model.

For specification purposes, *quality attributes* are attached to program entities; here, quality attributes must be understood in a very general manner. In the context of extra-functional specifications, quality attributes naturally include items like response time, size, availability, and so on. They correspond to QoS characteristics in traditional QoS vocabulary. Extra-functional quality attributes are used to specify quality of service and resource needs for components and applications. But, since we aim at extending more traditional DbC languages, we also consider values of parameters as implicit quality attributes.

QoS minded quality attributes have quality attribute types composed of a type, i.e. the set of values that the quality attribute can take, and an optional measurement units for numerical quality attributes. Values for quality attributes in GCCL can be element of constraint domains, which currently include real arithmetic, boolean constraints, and user-defined finite and finite set domains. GCCL can be extended to include other kinds of constraint domains.

For each quality attribute associated to some program entity, *values* can be qualified by *measurement units*. The semantics of measures or values takes into account these *measurement units* attached to them, especially when comparing values of different quality attribute types. Measurement units include basic units, such as seconds, bits, etc., as well as compound units, such as bits/seconds.

We do not define here a complete list of basic units; such a list is produced from international standards, like ISO Quantities and units standards. Measures qualified by different units can be normalized into a standard units space; afterwards they can be compared, added, etc. More precisely, our model defines a unit as an object composed of basic unit objects, each unit object having a `power` attribute expressing the power dimension. Each basic unit object is able to convert itself into a standard unit along the same dimension. For example, an acceleration expressed in feet per square second is made of a `Foot` object with a value of 1 for its power attribute, and a `Second` object with a value of -2 for its power attribute (as feet/second^2 is the same as feet $\cdot$ second^{-2}).

The figure 2 proposes a UML class diagram summarizing this quality attribute model and its relationships to the previously presented entity model (the two diagrams link through the class `Entity`). A `QualityAttribute` has a `name` and has a certain `QualityAttributeType`. This `QualityAttributeType` says:

1. that the values taken by the quality attribute are of some `Type`, defining the set of values that the quality attribute can take;
2. optionally, that these values can be element of a `ConstraintDomain`, defining properties of constraints that can be expressed on them; and
3. again optionally, that a measurement `Unit` is qualifying the values.

As we will see later (§3.3), a `ConstraintDomain` has specific admitted expressions, i.e. a `ConstraintExp` that can appear in contracts and that respects the restriction imposed by the constraint domain in order to apply a specific contraint solver to these constraints. Currently, these domains include `Real`, `FiniteDomain` and `FiniteSet`. The domain `Real` is used in for quantitative QoS specifications and is further refined in domains `RealArith` for real arithmetic, and `RealLin` for real linear arithmetic. Finite domains and finite set domains deal with user-defined sets of values expressing levels of qualitative QoS. For example, new finite and finite set domains can be defined to deal with levels of security. A primitive concrete finite domain `Bool` is also defined.

3.2 Underlying event model

To define when and how measurements must be made, to express constraints over these measured values, but also to provide constraints with a precise semantics for the run-time checks of constraints, we have to relate quality attribute measures and constraint checks to what happens at run-time. For some quality attributes, measures can be done at some specific time instant, for others they will be done for some elapsed time. To define precisely when and how each measure is taken by the underlying run-time system and QoS architecture,

the GCCL programmer refers to the lifecycle of entities and a model of their execution as events governing their evolution through states in this lifecycle. These concepts are introduced by the GCCL event model.

The GCCL event model adopts, adapts and generalizes the following foundational model due to Blair and Stéfani in their definition of QL, a logic designed to express QoS requirements in distributed multimedia systems [5].

Foundational event model

Let $(T, \leq, +)$ represent the time domain, where:

- T is an infinite set of instants with a minimal element $-\infty$ and a maximal element ∞,
- $\leq$ is a total order relation on instants, and
- $+$ is a binary addition relation on T.

Lifecycles of program entities are defined by sets of *event types*, representing types of state transitions that can be observed during the execution. Individual event types are denoted ε_i and the finite set of all event types $\Pi = \{\varepsilon_1, \varepsilon_2, \varepsilon_3, ...\}$ is the union of all events in the lifecycles of all program entities subject to contract.

Events are run-time manifestations of event types, and therefore represent specific occurrences of state transitions actually observed during the execution. Individual events are denoted e_i and $E = \{e_1, e_2, e_3, ...\}$ is the set of events that occurred during a particular run of the application. We denote $e_i : \varepsilon_j$ the fact that the event e_i is of event type ε_j.

Event histories are discrete sequences of events of the same event type that actually occurred during the current execution. They are denoted h_i and $H = \{h_1, h_2, h_3, ...\}$ is the set of all histories in the current execution. Histories can be interpreted as functions relating integers to events, i.e. $h : \mathbb{N} \rightarrow E$, providing a means to access the nth occurrence of a specific type of events.

The event model assumes the availability of a dating function, denoted $\mathcal{T} : E \rightarrow T$, relating an event to an instant at which the event occurred, i.e. its *date*. Let $\phi : \Pi \rightarrow H$ be a function returning the history of events associated to an event type, then the date of the nth occurrence of an event of the event type ε_i is given by:

$t = \mathcal{T}(\phi(\varepsilon_i), n)$

GCCL events and event types

In GCCL, lifecycles can be expressed over a set of event types and states for each program entity. A lifecycle also includes a transition relation defining how the type of program entity evolves over the time. Such transitions are triggered by event types. At run-time, events are observed when entities pass from one state to another.

The set Π of event types is therefore the set of all possible event types appearing in the lifecycle of each type of program entities, individualized over each specific program entity of that type in the program. The designation scheme for a particular event type occurring for a specific program entity extends the dot notation by linking the name of the program entity (optionally dotted itself) with the name of the event type. Whenever unambiguous by the context, designators can omit useless prefixes of dotted names.

For example, consider the event type `activation` appearing in the lifecycle of operations in interfaces, then for each operation `oper` in each interface `inter` called `inter.oper`, an event type `inter.oper.activation` appears in Π. Within the interface `inter`, `oper.activation` suffices to designate this event type.

The primitive function `dateOf` defined on event types takes an integer and returns the date of the corresponding event. Events in GCCL are instances of event types, themselves causally related by the lifecycle of an entity. When computing the value of some quality attributes or when expressing some constraints between quality attribute values computed at different points in time, it is important to be able to link events by a causal precedence relationship.

For example, when computing and constraining the response time of an operation `o` using the `dateOf` function, we need to relate a termination event for `o` to the activation event that initiated the execution which termination is being observed. One cannot simply talk about 'a previous activation event', since in concurrent execution, several activation events may occur before the termination of the first activation. Causal precedence relationships among event types derive from transition paths in the lifecycle of entities and can be computed as events trigger state changes at run-time.

GCCL defines a function `precededBy` on events taking an event type ε and returning the event $e : \varepsilon$ that precedes and somehow caused the current event, if such an event exists. For example, applying `precededBy` on a termination event e_i of some operation o and for the event type `activation` of o returns the event e_j that started the execution of o resulting in the `termination` event e_i. In fact, both events relate to the same run-time manifestation of the program entity o. More precisely, dates of events totally ordered for each event type, together with the order of event types in lifecycle of entities induce a partial order structure for events.

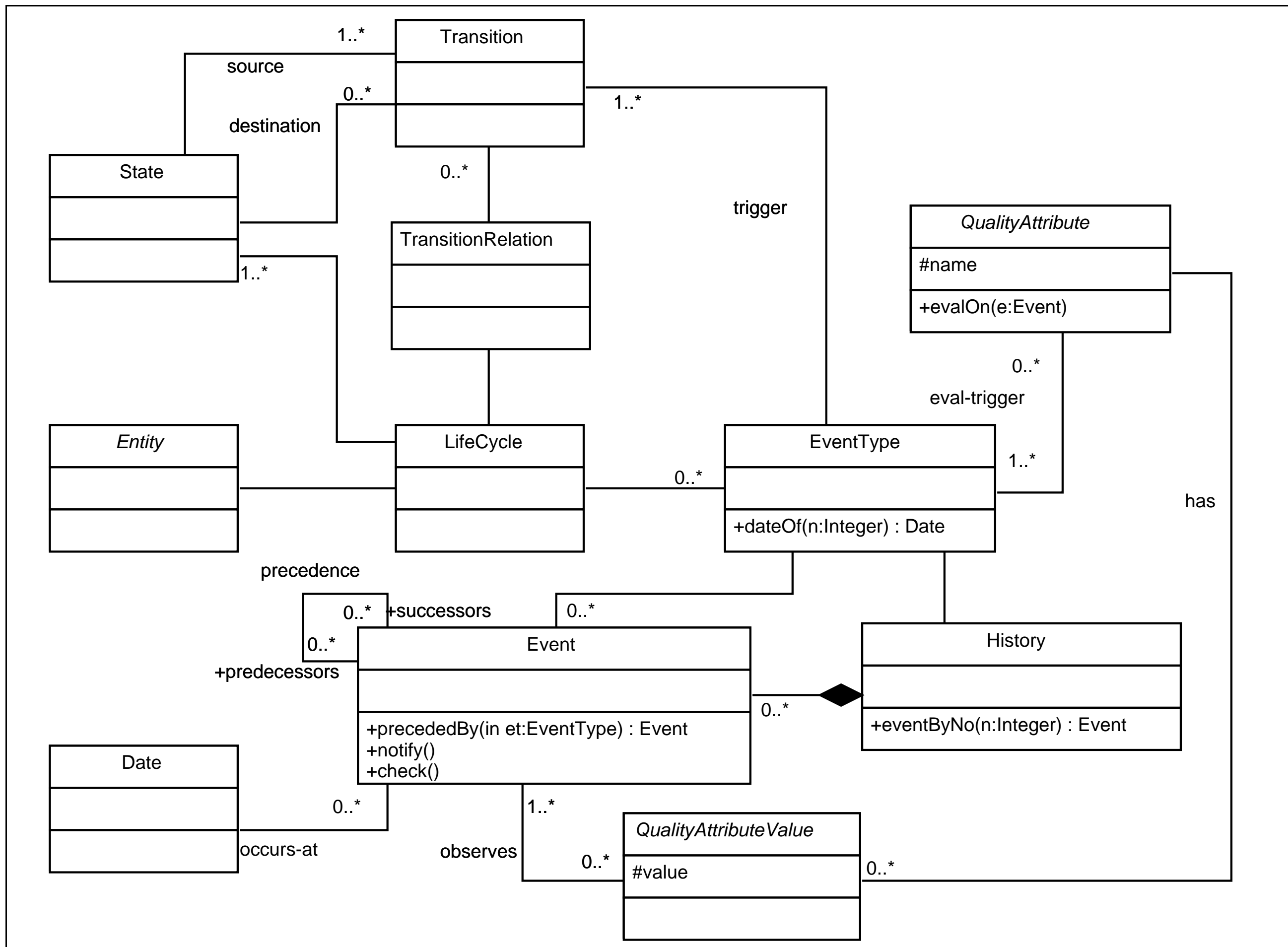

Figure 3: Event model

Events play two additional roles in GCCL. Quality attribute values (`QualityAttributeValue`) are obtained either through measurements done within the run-time system or by computing them using other quality attribute values or low-level system services. In GCCL, values for quality attributes are associated to events. Computation procedures are attached to quality attribute and they produce values when events, which event type is a trigger for the quality attribute (`eval-trigger` association), occurs. This is implemented by a simple notification pattern. The second additional role concerns constraint checking as we will see shortly.

Notice here that computation procedures for quality attributes can be used to implement the translation from low-level system-oriented QoS quantitative values into high-level qualitative QoS values, and therefore to abstract the end-user from such low-level concerns. Such an abstraction mechanism is seen as an important requirement for contract languages, since the translation from high-level application-oriented QoS attributes are easier to apprehend then low-level system-oriented measures (see §2.2).

The figure 3 proposes a UML class diagram summarizing this event model and its relationships to the entity and quality attribute models.

3.3 Contracts and constraints

Imposing QoS contracts means *constraining* measures or values of quality attributes attached to program entities. This is done by applying *constraints* to quality attributes of program entities. *Satisfaction* of a constraint can be *verified* at run-time using measures or values for the quality attributes. It can also be *validated* prior run-time through testing. Satisfaction of a required constraint C_1 can sometimes be *proved* once and for all when it is implied (or entailed) by another constraint C_2. C_1 is then said to *conform* to the constraint C_2. Constraints can refer to individual measures or values, but also to sets of values, through *statistical properties* of these sets (mean, variance, etc.).

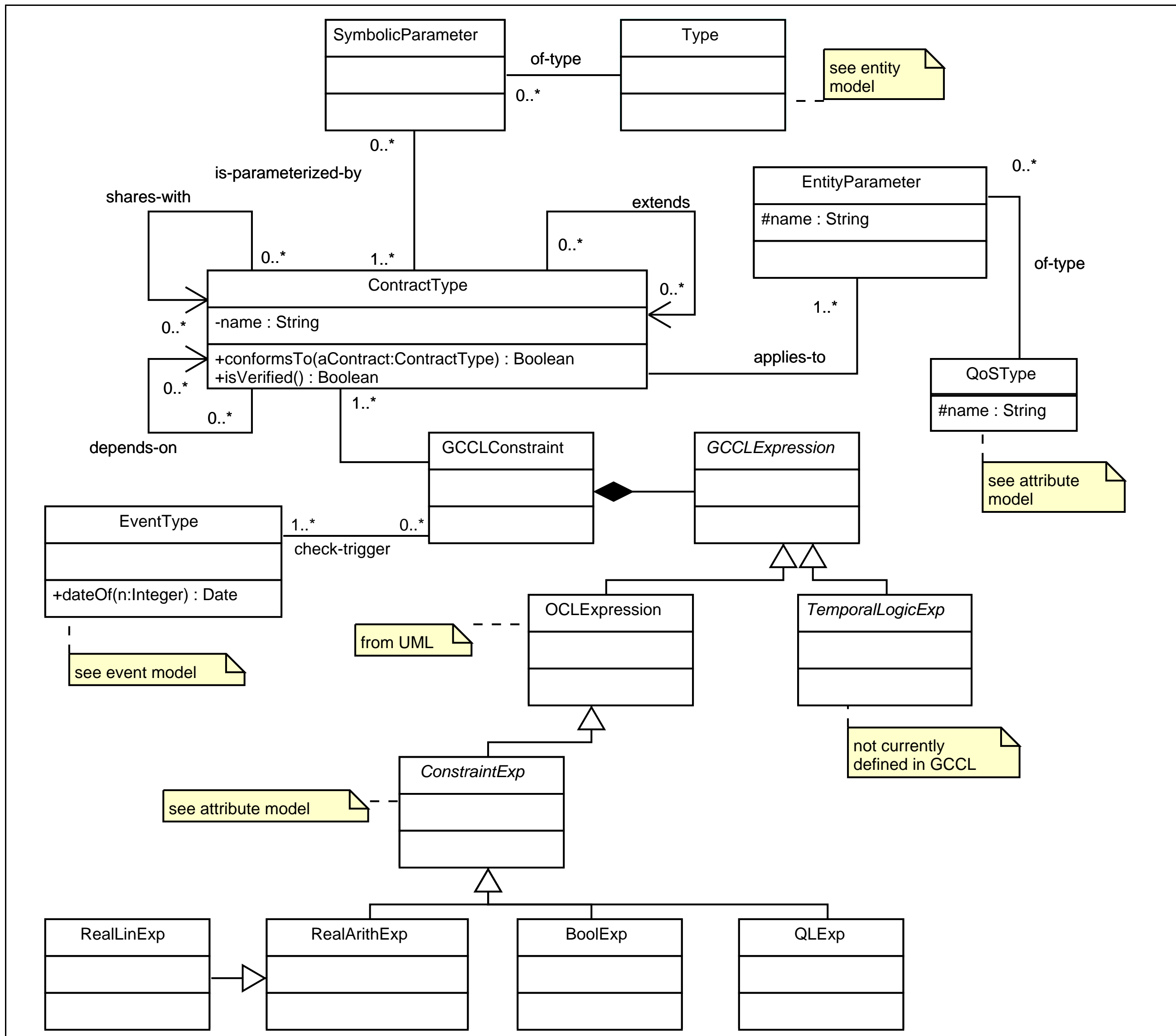

Figure 4: Contract model

A *constraint expression* is an assertion defined over quality attributes of program entities. Currently in GCCL, constraints can be written using OCL-like syntactic constructs [22]. For traditional invariants, pre- and postconditions, expressions use logical and comparison operators as well as logical quantifiers ($\exists$, $\forall$). All quality attribute can appear in such general constraints. When no constraint domain can be associated with the quality attribute type, the quality attribute can appear in these general expressions for which conformance is defined as logical implication and therefore needs the full power of logical proofs.

For QoS minded quality attributes, constraint expressions can relate to constraint domains. The advantage of using specific constraint domains and therefore restricted constraint expressions from that constraint domain lies in the use of constraint programming concepts to define the conformance relationship between constraints and then contracts.

GCCL can also be extended to deal with restricted constraint expressions that do not relate to such domains, by defining specific conformance relationships between them. In our model, we explicitly mention QL expressions, from the logic of Blair and Stéfani for which a tractable theory of conformance has been defined [5]. QL introduce no new syntactic constructs to those already existing in OCL. It rather restricts them to:

- integer and date constants;
- variables over event types, integers and delays (differences of dates);

- functions restricted to integer addition, integer multiplication by constants, delay addition, dating function;
- predicates restricted to equality over integers, event types, events and delays, as well as inequalities ($\leq$, $\geq$) over integers and delays;
- logical connectors ($\wedge$, $\vee$, $\rightarrow$, $\neg$) and quantifiers ($\forall$, $\exists$).

Figure 4 summarizes the current contract and constraint model for GCCL. Each constraint type has a constraint expression which syntax is currently the OCL syntax (`OCLExpression`). As we have seen, constraint domains restrict the kinds of constraint expressions that can be dealt with by the constraint solver. Hence, constraint expressions are further refined into real arithmetic expressions (`RealArithExp`), real linear expressions (`RealLinExp`) and boolean expressions (`BoolExp`). Other concrete constraint domains will also have their own specific constraint expressions. As already mentioned, other constraint types can also be defined. For example, GCCL can include QL expressions [5] (`QLExp`), which are restricted OCL expressions, but also other kinds of temporal logic expressions which are not expressible in OCL because of the lack of temporal logic operators. Conformance of constraints that does not refer to constraint domains per se must be defined specifically (for example, QL). Conformance between contract types and contracts is defined in terms of conformance between constraints.

Dealing with environmental parameters

In real situations, constraint expressions in a specification must often refer to values than broadly speaking come from the environment of the application. This include specific values delays, limits or more generally requirements imposed on the application to perform its duty. For example, this can be the frequency at which some external process generates values imposing a real-time limit on the processing of these values. Environmental parameters also include levels of resources specific to the current mix of run-time system and hardware over which the components execute.

Instead of early committing to specific values for these parameters, constraint expressions would rather use symbolic variables that should be bound lately and propagated in contracts before they can be dynamically checked. Informally, we want to express in GCCL things like: the response time of a method in a provided interface will be $f(P)$, where P is a symbolic variable representing the processing power of the underlying processor and f an appropriate function estimated by the component provider. Another example is to express that if the response time for some (required) operation op_1 is less than a certain value in milliseconds represented by the symbolic variable M, then the response time for some other (provided) operation op_2 is less than $2M\ msec$. For instance, this could express the fact that the throughput offered by a multimedia component in number of video frames per second depends upon the throughput of the network interface required by the component to transmit these video frames.

From a theoretical point of view, these symbolic variables should be interpreted as universally quantified variables in constraints. Conformance therefore takes this into account by ascertaining that constraints and contracts are conformant for all possible values of these symbolic variables, or generate constraints that these values must observe to make sure that the conformity holds.

Constraint checking and events

Different types of constraints must be checked at different instants during the execution. Traditional invariants, pre- and postconditions are usually checked upon method activation and return. Evidence shows that invariant should also be checked when control leaves and returns to the object [8], to deal correctly with call-backs. In a concurrent programming context, the only places where an invariant could be broken are critical sections.

Defining precisely the semantics of constraint checking is known as a hard problem [8]. Eiffel has a very precise set of rules to deal with invariant, pre- and postconditions that go much farther than simply generate code for the check on entries and exits of methods, as some preprocessors for DbC in other languages do.

Thanks to the generality of its event model, GCCL offers a flexible and precise way to define the semantics of dynamic checking for new *constraint types*. Constraint check is triggered by the occurrence of events of predefined event types. With a sufficiently fine-grained event model, it becomes possible to specify as precisely as one wants when each constraint must be checked.

Contracts and contract types

Contracts are sets of constraints involving a number of program entities to which are attached specific quality attributes. Because similar contracts can apply in several contexts, contract abstractions, hereby named *contract types*, are contracts parameterized by the constrained program entities.

```
public interface Retrievable {
    public void insert(Key k, Value v) ;
    public Element lookFor(Key k);
}

public interface Collection {
    public void insertAt(Element e, int index) ;
    public int size() ;
    public Element get(int index) ;
}
```

Figure 5: Interfaces for the response time example

Contract types are parameterized by symbolic variables, which represent dependencies over values that will be known later (when binding these variables, eventually at deployment or at run-time, but before checking the constraints). Because such symbolic variables must often be shared among constraints appearing in several contracts, contract types are parameterized by such symbolic variables and a contract type that wants to use a symbolic variable defined in another contract type becomes associated to that other contract. Contract types are therefore also parameterized by other contract types to express this kind of relationships, and to syntactically give the possibility to use in a contract type C the symbolic parameters defined in the contract type C'.[2]

Another totally different kind of dependency arise between contracts when different contracts associated to provided interfaces rely on differents constracts associated to required interfaces. Informally, we want to be able to express things like: if the component has access to the network with a certain level of security (secure sockets, for example), then it can provide a secure transaction mechanism, but if the contract on the required interface does not offer any security mechanism, then the transaction mechanism will be unsecure. Contract types therefore define dependencies between contract. These dependencies are used in the semantics of GCCL, which says that if some required contracts are satisfied, then all the provided contracts that depends upon those satisfied required contracts must be statisfied by the component.

Contract type inheritance

Contract types can be defined as extensions of existing contract types. The new contract type then inherits from the extended ones their entity parameters, their relative contract types, their symbolic parameters, their dependent contracts and finally their constraints. Extension must be understood as a refinement relationship. The new contract type must conform to the extended ones, i.e. whenever contracts of the new contract type are satisfied, contracts of the extended contract types would also be satisfied. New constraints can therefore only strenghten constraints appearing in the extended contract types.

The figure 4 includes the current contract model for GCCL. A `ContractType` has entity parameters, defined by a name and a `QoSType`, and symbolic parameters (`SymbolicParameter`). Also, a `ContractType` can extend other contract types, share symbolic parameters with other contract types and depend upon other contract types. *QoS types* are introduced to support contract types as contract abstractions, by categorizing program entities given the set of quality attributes attached to them, as shown in Figure 2. QoS types serve two purposes: identify the quality attributes that can be constrained for each of the parameters in a contract abstraction and verify the conformance of program entities to contract actual parameters when applying the contract type. Similarly to contract types, QoS types can be defined as extensions of existing QoS types; the new QoS type inherits all of the quality attributes defined in the extended ones.

4 Examples

An important requirement for GCCL is the ability to include it within several modeling and programming languages. Its design must therefore avoid specific concrete syntax, and be rather based on an abstract syntax that can be targeted to specific host languages. The four preceding models define an abstract syntax for GCCL. Targeting includes the definition of a mapping from these models to a concrete syntax matching the host

[2] Another way to deal with such dependencies would be to put in one contract provided and required constraints, as in Blair's and Stéfani's approach [5]. We believe that our contract dependency mechanism will prove to be more flexible, modular and extensible.

```
lifecycle operationLifecycle {
    states      dormant, executing ;
    event-types activation, termination ;
    with from dormant on activation to executing ;
         from executing on termination to dormant ;
}

constraint-type pre on methodLifecycle checked-upon activation ;
constraint-type post on methodLifecycle checked-upon activation ;
constraint-type invariant on methodLifecycle checked-upon activation, termination ;
```

Figure 6: Definition of operation lifecycle and the three basic constraint types

language, and the definition of a mapping between designation schemes and scoping rules of GCCL and those of the host language. In this section, we adopt a Java-like syntax as a mock-up for such a targeting of GCCL.

4.1 Functional part

Our simplified example assumes that an application includes a component `C1` that gathers key-value pairs and then provide a retrieval service for values given the associated key. Keys are assumed to be comparable. `C1` provides this service by publishing the provided interface `Retrievable` appearing in Figure 5. To store the elements, this component requires two instances of a collection component providing three services: inserting, returning the size of the collection and getting an element from an index ranging between 1 and the size of the collection. The required interface is called `Collection` and it also appears in Figure 5.

For that purpose, a procurement team is required to find on the market a component providing the needed service. This team finds two off-the-shelf components `C2` and `C3`, both providing the interface `Collection`. After a careful examination of the commercial documentation for the two components, the procurement team ascertain that they are both effectively implementing the desired functionality. So, they are coming back to the development team for a decision, since there is no reason to choose one or the other besides the difference in cost.

4.2 Extra-functional properties

Developers of `C1` in fact have implemented the `Retrievable` interface by keeping keys sorted in one collection and by storing the corresponding value at the same index in the second collection. To look for an element, they get the index of the key using a binary search on the first collection and then they use this index to retrieve the corresponding element in the second collection.

Now, consider that `C2` implements the collection using a vector while `C3` uses a linear list. Elementary knowledge on algorithms tells us that using `C3` is just plain wrong. The basic assumption that makes a binary search interesting, i.e. log time, is that accessing an element from an index takes constant time (besides the fact that computing the arithmetic mean of two indexes is also done in constant time). We all know that if accessing elements from indexes takes linear time, then binary search will turn $n \log n$ and a simple sequential search becomes much better.[3]

But how can we apply this elementary knowledge from our first courses in computer science if components `C2` and `C3` are "*opaque implementation of functionality*"? Of course, in this oversimplified example, many different solutions appear feasible: documenting the choice of implementation (but how?), doing performance measurements and discovering the behavior of the two components (which assumes that you can test the two components before buying them), ...

[3] As incredible as this might appear, this scenario is in fact inspired from a true story, when one of the author was implementing an intensive computing application. This application was relying on a library for piecewise linear approximations of functions from reals to reals implemented by a summer intern. We discovered only after several weeks, while tracking inefficiencies, that the summer intern thought it would be wise to implement the function locating the piece containing the value for which the function had to be evaluated using a binary search, neglecting the fact that he used a linear list to store the pieces... The modification of the algorithm to a linear search offered a gain of one order of magnitude in execution time of the application! We welcome readers to send us other examples of such "incredible" situations.

```
qostype responsiveMethod { attribute responseTime : float in seconds ; }

qostype countableInvocations { attribute numberOfInvocations : float ; }

entity Retrievable.lookFor, Collection.get : responsiveMethod
                                              with-lifecycle operationLifecycle
{
    compute responseTime
            when e = termination
            using e.dateOf() - precededBy(e, activation).dateOf() ;
}

entity Collection.insert : countableInvocations with-lifecycle operationLifecycle
{
    compute numberOfInvocations
            when e = activation
            using precededBy(e, activation).Collection.insert.numberOfInvocations + 1 ;
}

contract-type constantResponseTime(m : responsiveMethod)
    symbolic-parameters float k1 ;
{
    post : m.responseTime < k1 ;
}

contract-type logResponseTime(m : responsiveMethod, c : countableInvocations)
    symbolic-parameters float k2, k3 ;
{
    post : m.responseTime < k2 * Math.log(c.numberOfInvocations) + k3 ;
}

contract constantTimeGet is constantResponseTime(Collection.get) ;

contract logLookFor is logResponseTime(Retrievable.lookFor, Collection.insert)
    depends-upon constantTimeGet ;
```

Figure 7: Response time example

We propose that extra-functional contracts can be used to specify these properties, to reason about the correctness of composing the two components, and eventually to check at run-time that the properties holds for each call to `lookFor`.

In GCCL, when targeting to the host language, lifecycles of the different entities would be defined once and for all as in Figure 6. The three basic types of constraints would also presumably be predefined once and for all. Programmers would then define the necessary QoS types needed to express the desired constraints. In this example, we need an attribute `responseTime` associated to operations. Because contract types don't have access to component implementations, we need another attribute to count the number of invocations of the `insert` operation to estimate the size of the collection. Figure 7 show the QoS type declarations for `responsiveMethod`, defining a `responseTime` quality attribute of type `float` expressed in seconds, and `countableInvocations`, defining a `numberOfInvocations`.

Next, entities `Retrievable.lookFor` and `Collection.get` are declared with QoS type `responsiveMethod` with the lifecycle of operations. This declaration also says that the attribute `responseTime` is computed when an event `e` of type `termination` (standing for `Retrievable.lookFor.termination` or `Collection.get.termination` depending on the context of the generated computing procedure) occurs, and this computation subtracts the date of the event of type `activation` that precedes the event `e` from the date of `e`.

The entity Collection.insert is then declared to be of QoS type countableInvocations. This attribute is computed at each activation by retrieving the number of invocations of the preceding activation event and adding one.

Two contract types are then declared. The contract type constantResponseTime constrains the response time of its entity parameter m to be less than a symbolic (constant) parameter k1. The contract type logResponseTime constrains it to be less than $k_2 \log(\texttt{c.numberOfInvocations}) + k_3$ for some symbolic parameters k_2, k_3, where c is a countableInvocations entity parameter.

The component C1 creates two contracts. The contract constantTimeGet is obtained by applying the contract type constantResponseTime to the entity Collection.get. The contract logLookFor is obtained by applying the contract type logResponseTime to the entities Retrievable.lookFor and Collection.insert, while specifying that this contract depends upon the constantTimeGet contract.

Components C2 and C3 publish contracts, the first to declare that its get provided operation has a constant time behavior, while the second to declare that its own get operation has a linear time behavior. Reasoning about the conformance of these contracts with the contract constantTimeGet of C1 will say that the contract of C2 is conformant, but not the one of C3. If this conformance cannot be decided statically, tests will exhibit the conformance or the non-conformance of the required and provided contracts as contracts in DbC would exhibit the failure of a postcondition.

4.3 Assessment

With this example, we want to stress the fact that extra-functional contracts will prove useful in component documentation and validation, but also in supporting component composition by statically trying to ascertain the correctness of the call relationships established between components. In this example, from a simple but realistic scenario in the future component-based software development processes, we have strived to show that neglecting extra-functional properties can lead to failing to meet "untold requirements" (such as time and space complexities), while using extra-functional contracts have the potential to pinpoint problems as soon as possible from component composition time to run-time.

In this example, we have shown how the response time requirements can be specified using GCCL. This specification can be used as documentation, but also to support automatic reasoning about the composition of components. Finally, provided that estimations are made for the different environmental parameters, contracts can be dynamically checked to reveal invalid assumptions about the correctness of component composition. Without this, tracking this kind of problems will not only be a time-consuming process, but also a major challenge to the quality of off-the-shelf component-based software.

Of course, this incurs some complexity in defining, compiling, possibly proving the conformance of contracts, as well as estimating the different environmental parameters and dynamically checking the contracts. But, we concur with the SEI and a large portion of the CBSE community that this kind of technology is needed to enable the emergence of a real component market, and if its complexity is the price to pay, then we have to face it. GCCL's goal is to help and support the programmer in expressing the requirements and capabilities of components, by tackling the inherent complexity of this process. The current GCCL proposal, presented in this paper, is one of the first that tries to address the full complexity of extra-functional components. From this already well-balanced design, more experiment will certainly suggest improvements but GCCL offers the necessary foundations to include such improvements.

5 Related work

Few full-fledged contract languages for QoS have been designed to date. Most QoS architectures rely on API to specify the requirements or the capabilities of software and systems. Typically, requirements and capabilities are expressed as precise individual target values, or sometimes as ranges of values. Some contract languages introduce specifications as contraints on required and provided QoS. QML [9] and QIDL [3] are notable examples of such languages, yet they are still quite limited.

To our knowledge, QML [9] represents one of the most comprehensive efforts to define a language for the specification of quality of service contracts, especially when it comes to language issues. In QML, QoS is modeled using contract types, contracts and profiles. Different aspects of QoS are considered as independent dimensions. A contract type defines a set of dimensions with their type of values. Contracts are instances of contrat types defining constraints on the different dimensions. Profiles bind contracts to entities in interfaces. QML is a modeling language, and therefore focus on the conformance relation between contracts (both at compile time

or at run-time, see [10]). Frølund and Koistinen provided the inspiring insight that contract conformance is a constraint satisfaction problem.

However, QML as a tool to model QoS is rather simplistic. Dimensions in contracts are useful to define the necessary domain ontology to talk about QoS, but they are only defined in contract types. Because there is no hierarchy of contract types, several contract types using the same dimension must redefine it independently. A second limitation of QML appears in the separate definition of contracts and interfaces. Frølund and Koistinen rightly insist that it should be possible to associate different QoS contracts to the same interface, yet the profile mechanism they provide does not allow programmers to state constraints that depend upon entities in interfaces. One cannot state in QML that the response time of a sort method is a function of the size of the collection to be sorted, which is a natural thing one would like to specify. Finally, the constraint language of QML is very limited, since one can only express basic constraints. Hence, one can say that:

```
responseTime < 10 msec
```

but one cannot say that:

```
responseTime < (100 + 10*sizeOf(l)) msec
```

where `l` would be the first parameter of the method for example. Using these strong restrictions, QML is able to give a full definition of conformance between contracts.

Becker and Geihs QIDL [3] as well as Février, Najm and Stéfani contracts for ODP [7] insist on contract checking based on observers attached to bindings. Contracts in these two proposals are therefore seen in an operational way, and therefore less amenable to symbolic reasoning. To follow these ideas, we could equip our `CallRelationship` and `Interface` objects with means to trap exchanges between components and check contracts at that time. We believe that the GCCL event-triggered constraint checks is much more flexible and powerful to deal with a large spectrum of different types of constraints. On the other hand, these authors offer interesting paths for the definition of a more formal semantics for GCCL.

In Corba IDL [16] and TINA ODL [11], provision is made for expressing explicit QoS values or ranges of values as QoS parameters in messaging services. ODL [18] is an extension of the OMG IDL language defined by the TINA Consortium to support features that were not covered at the time by IDL. Among them, quality of service was an important issue. However, QoS in ODL is restricted to multimedia end-to-end guarantees. As in other similar approaches, the expressive power of their languages is much more limited than our approach based on constraints. Recent additions to CORBA IDL for QoS are quite similar in nature to those originally proposed by ODL [16].

QuO [17] is a framework for QoS Objects in the context of distributed computing where the focus is put on adaptation of applications to changes of QoS from the environment. Despite its broader perspective, QuO uses Quality of service Description languages (QDL) that, as QIDL and above contracts for ODP, are highly operational in nature. The goal is to compile a specification of adaptation behaviors into observers that will do the reaction at run-time. QDL allows to define QoS regions, i.e. subspaces of values for QoS attributes, and attach to transitions among these regions adaptation behaviors. Compared to GCCL, QDL is quite limited in its ability to express QoS constraints, although the framework addresses goals that are complementary to those of GCCL.

6 Conclusions

Following a long period when quality of service was essentially viewed as a property of distributed multimedia systems, extra-functional properties of software attract more and more attention as the software industry struggles to achieve its revolution towards standardized, off-the-shelf components. Applications in the fields of distributed, embedded and mobile computing must satisfy stringent requirements on extra-functional properties such as performance, response time, safety and security. To address these issues, components must be build in such a way that their extra-functional properties are specified, published and manipulable in such a way that overall properties of systems can be predicted from the properties of the components from which they are built.

Contracts and design by contracts are viewed as essential techniques and concepts to address these issues, yet the current technology has to be extended towards extra-functional properties. In this paper, we have undertaken a broad attempt to respond to this challenge by designing GCCL, a general common contract language. GCCL is general in the sense that it is based on an open-ended model of contractable entities, quality attributes and constraints, which can be extended to deal with all kinds of extra-functional requirements. GCCL

is a common language in the sense that it is meant to be compatible with existing DbC languages, such as OCL, but also because it does not commit itself to any host language. Its foundational models as well as its concrete syntax can be adapted to a large spectrum of languages, from modeling languages such as UML, to programming languages such as Java, C#, Eiffel and so on.

GCCL proposes several innovations compared to existing DbC and QoS specification languages. GCCL is based on a model of contractable entities and an event model defined upon the lifecycle of entities, i.e. sequence of events governing their evolution from state to state at run-time. The entity model is then augmented with a quality attribute model associating attributes to contractable entities, forming QoS types. The event model allows us to define precisely the dynamic semantics for the computation of quality attribute values, which is triggered by the occurrence of events in the lifecycle of these entities. Constraints are imposed on these quality attributes. Constraint types also use events to define the semantics of dynamic checking of constraints since events occuring in the evolution of program entities trigger the check of constraints.

These innovations are not only crucial for the definition of a precise semantics for GCCL extra-functional contracts, but can also serve as a basis for the definition of a semantics of existing contract language, such as OCL.

GCCL also introduces several new powerful language concepts that are useful to tackle the full complexity of extra-functional contracts. Conformance for quantitative attributes use constraint programming concepts to provide GCCL with a static semantics to prove the correctness of component assemblies by establishing the conformance of required and provided contracts, whenever it is possible. Constraint programming techniques are also used to deal with the propagation of constraints from or towards (to dimension it) the underlying run-time systems. Contract types are a powerful abstraction mechanism to express the parameterization of contracts over constrained entities, but also to share environmental parameters expressing requirements external to the application and express dependencies between sets of provided and required contracts.

Currently, GCCL is restricted to contracts for components, which express constraints put on quality attributes of entities related to components: component themselves, interfaces, operations in interfaces and their parameters. Future extensions of GCCL will cover more of the potential contractable entities. A lot of work has still to be done to fully exploit the possibilities of constraint programming in the static semantics of GCCL and its use in Component-Based Software Engineering (static checking, test, dimensioning, ...). A full formal semantics for both static and dynamic aspects of GCCL is currently under definition. Future work also include extensions to GCCL in order to provide programmers with the capability to control more of the traditional QoS management aspects. For example, we would like to extend GCCL in order to define and require specific management policies for negotiation, renegotiation, adaptation and maintenance.

References

[1] C. Aurrecoechea, A. Campbell, and L. Hauw. A Review of Quality of Service Architectures. *ACM/Springer Verlag Multimedia Systems Journal*, 6(3):138–151, May 1998.

[2] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Technical Concepts of Component-Based Software Engineering, Volume II. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie-Mellon University, May 2000.

[3] C. Becker and K. Geihs. Generic QoS Specifications for CORBA. In *Proceedings of Kommunikation in Verteilten Systemen (KIVS'99)*, 1999.

[4] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.

[5] G. Blair and J.-B. Stéfani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1998.

[6] D. Chalmers and M. Sloman. A Survey of Quality of Service in Mobile Computing Environments. *IEEE Communications Surveys*, pages 2–10, Second Quarter 1999.

[7] A. Février, E. Najm, and J.-B. Stéfani. Contracts for ODP. In *Proceedings of the Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software — Towards a mathematical transformation-based development*, volume 1231 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, May 1997.

[8] R. Findler and M. Felleisen. Contract Soundness for Object-Oriented Languages. *Proceedings of OOSPLA 2001, ACM Sigplan Notices*, 36(11):1–15, November 2001.

[9] S. Frølund and J. Koistinen. QML: A Language for Quality of Service Specification. Technical Report HPL-98-10, Software Technology Laboratory, Hewlett-Packard, February 1998.

[10] S. Frølund and J. Koistinen. Quality of Service Aware Distributed Object Systems. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems, COOTS'99*, May 1999.

[11] T. Hamada, S. Hogg, J. Rajahalme, C. Licciardi, L. Kristiansen, and P. Hansen. Service Quality in TINA: Quality of Service Trading in Open Network Architecture. *IEEE Communications Magazine*, 36(8):122–130, August 1998.

[12] O. Hanssen. FlexiNet - QoS Investigation. Technical Report APM.1977.01.00, APM, June 1997.

[13] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, May/July 1994.

[14] M. Klein and R. Kazman. Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie-Mellon University, October 1999.

[15] B. Meyer. *Object Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[16] Object Management Group. *The Common Object Request Broker: Architecture and Specification, v2.6*, December 2001.

[17] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to Specify QOS Aware Distributed (QuO) Application Configuration. In *Proceedings of the Third International Symposium on Object-Oriented Real-Time Systems, ISORC'00.* IEEE, March 2000.

[18] A. Parhar, editor. *TINA Object Definition Language Manual v2.3.* TINA-Consortium, July 1996.

[19] S. Si Alhir. *UML in a nutshell.* O'Reilly & associates, 1998.

[20] C. Szyperski. *Component Software – Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[21] G. Wang and A. MacLean. Software Components in Contexts and Service Negociations. In *Proceedings of the 1999 International Workshop on Component-Based Software Engineering*, pages 83–88. Software Engineering Institute, Carnegie-Mellon University, 1999.

[22] J. Warmer and A. Kleppe. *The Object Constraint Language — Precise Modeling with UML.* Addison-Wesley, 1999.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399