REIFYING THE SEMANTIC DOMAINS OF COMPONENT CONTRACTS

Jean-Marc Jézéquel
Irisa (INRIA & University of Rennes 1)[*]

Abstract In domains such as automotive or avionics, software cannot any longer be produced as a single chunk, and engineers are contemplating the possibility of componentizing it. A component only exhibits its provided or required interfaces, which must be enriched to take into account extra-functional aspects. This defines multi-level *contracts* between components allowing one to properly wire them. Instead of defining an integrated language only making available a limited set of concepts for modeling extra-functional aspects, we propose to handle open-ended modeling of extra-functional aspects of real-time and embedded systems, based on meta-modeling techniques and Model Driven Engineering (MDE) for reifying their semantics. Then the designer can use off-the-shelf tools to perform various kinds of design time analysis.

1. INTRODUCTION

In domains such as automotive or avionics, products are characterized by high performance, high dependability, outstanding quality demands, and exponentially increasing complexity. Since these real-time and embedded systems are getting ever more software intensive, their software cannot any longer be produced as a single chunk. Automotive or avionics engineers are thus contemplating the possibility of componentizing it, along the lines of Szyperski's [12] ideas, where

> *a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-party.*

In real-time and embedded systems however, we have to take into account many extra-functional aspects, such as timeliness, memory con-

[*]This work has been partially supported by the Artist2 Network of Excellence on Embedded Systems Design (IST-004527).

sumption, power dissipation, reliability, performances, and generally speaking Quality of Service (QoS). These aspects can also be seen as *contracts* [9] between the system, its environment and its users. These contracts must obviously be propagated down to the component level. One of the key desiderata in component-based development for embedded systems is thus the ability to capture both functional and extra-functional properties in component contracts, and to verify and predict corresponding system properties [11].

To master the complexity of modern real-time and embedded systems, engineers must rely on *modeling* for representing several aspects of reality for some purposes, such as thorough quality and stability assessment at early design stages or active treatment of design assumptions to guide system development. However the extra-functional aspects that must be taken into account are so various and domain specific (and even vary very much company-wide inside the same domain) that there is no integrated modeling language encompassing them all.

We propose an alternate way to handle open-ended modeling of extra-functional aspects of real-time and embedded systems, based on meta-modeling techniques and Model Driven Engineering (MDE) [3]. Instead of defining an integrated language only making available a limited set of concepts for modeling extra-functional aspects, we propose to let the designer define as many modeling sub-languages as needed for describing QoS contracts. These sub-languages are defined as executable meta-models: their abstract syntax are defined as plug-in extensions to the UML2.0 meta-model, their static semantics are given with a set of OCL constraints, and their dynamic semantics are expressed with Kermeta [10]. Once the semantic domains of component contracts has been reified this way, the designer can use off-the-shelf tools (such as model-checkers or constraint solvers) to perform various kinds of design time analysis.

The rest of the paper is organized as follows. Section 2 details the notion of *contract* along the four levels defined in [2]. Section 3 presents how they can be integrated at meta-modeling level. Section 4 discusses the problem of composing the components and computing the QoS properties of the assembly. Section 5 discusses how various kinds of design time analysis can then be performed. Section 6 discusses related works, and we finish by some conclusions and perspectives.

2. FOUR LEVELS OF COMPONENT CONTRACTS

The term contract can very generally be taken to mean "component specification" in any form. This specification should tells us what the component does without entering into the details of how.

A contract is in practice taken to be a constraint on a given aspect of the interaction between a component that supplies a service, and a component that consumes this service [9]. Component contracts differ from object contracts in the sense that to supply a service, a component often explicitly requires some other service, with its own contract, from another component. So the expression of a contract on a component-provided interface might depend on another contract from one of the component-required interfaces.

That is also known as the *assume/promise* approach where each component has a black-box model, which explicates assumptions about its environment and state corresponding promises on the service offered by the component to the environment (e.g.; promising state-dependent latencies of component services as a function of assumed service times of invoked services). Similarly, bounds on the occurrence of critical events can be promised only based on failure rates for invoked services, assumptions on failure rates for an execution platform, and knowledge of the ways the component itself can induce and propagates failures.

A now widely accepted classification of different kinds of contracts has been proposed in [2], where a contract hierarchy is defined consisting of four levels.

Level 1 : Syntactic interface, or signature (i.e. types, fields, methods, signals, ports etc., which constitute the interface). The syntactic interface of a component is a list of operations or ports, including their signatures (the types of allowed inputs and outputs), by means of which communication with this component is performed. These Level 1 contracts allow static type checking, that is a verification that there is no possibility of interaction errors (i.e. messages not understood).

Level 2 : Constraints on values of parameters and of persistent state variables, expressed, e.g., by pre- and post-conditions and invariants. Functional properties are used to achieve more than just interoperability. Level 2 contracts are about the actual values of data that are passed between components through the interfaces, whose syntax is specified at Level 1. Typical properties of interest are constraints on their ranges, or on the relation between the parameters of a method call and its return value.

Level 3 : Synchronization between different services and method calls (e.g., expressed as constraints on their temporal ordering). Level 3 contracts are about the actual ordering between different interactions at the component interfaces. They allow one to express explicit control information, which makes the expression of complex, history dependent input/output relations much easier.

Level 4 : Extra-functional properties, such as performance, memory consumption, constraints on response times, throughput, etc..

A quality of a system (e.g.; memory consumption) can in general be considered as a function mapping a given system instance with its full behavior onto some scale. The scale may be qualitative, in particular it may be partially or totally ordered, or the scale can be quantitative (as for memory consumption), in which case the quality is a measure. The problem of realizing systems that have certain guaranteed qualities, also known as their quality of service (QoS), involves the representation of such qualities in design models or languages and techniques to implement and analyze them as properties of implemented system instances.

There exist many QoS contracts languages which allow the designer to specify the extra-functional properties and their constraints on the provided interfaces only. However, few of them allow specifying dependency relationships between the provided and required services of a component (e.g.; the memory consumption of a component may depend on some function of its required interfaces). If we want an open-ended way of letting the designer specify such QoS dimensions, we need to provide a meta-model infrastructure allowing for both:

- the description of the various types of contracts and their dependencies at the modeling level,
- the specification of a mapping function from the syntactic domain of the contract to its semantic domain.

For example, to continue with our simple example of memory consumption, in the case of a simple centralized memory system, the operational semantic of memory can be defined as a class with an integer attribute modeling the amount of memory already consumed, and two operations, *alloc* and *free*, each taking an integer parameter.

3. META-MODELING COMPONENT CONTRACTS

These four levels of contracts should fully describe all the visible properties of components, whatever their actual implementation. Model based engineering is the idea that working with models of such component can be useful to perform a range of engineering tasks, such as prototyping, dimensioning, validation, and even code or test generation. While the levels 1 to 3 are supported in a number of ways in many modeling languages (e.g.; SDL, Lotos, as well as various flavors of automata-based languages) the aspects that must be taken into account at level 4 are so various and domain specific that there can probably be no integrated modeling language encompassing them all.

We thus propose an alternate way to handle open-ended modeling of extra-functional aspects of real-time and embedded systems, based on meta-modeling techniques and Model Driven Engineering (MDE) [3]. Instead of defining an integrated language only making available a limited set of concepts for modeling extra-functional aspects, we propose to let the designer define as many modeling sub-languages as needed for describing QoS contracts.

At the abstract syntax level, these open-ended sub-languages are linked to a component meta-model playing the role of a backbone. We use a subset of the UML2.0 meta-model for describing component-related notions, as follows:

Level 1 The structural part of a component type is defined by a set of port types. Each port type is identified by a name and a set of provided and required UML2 interfaces. Each interface groups a set of services. Composite component types also contain a slot for each sub component they contain.

Level 2 For describing functional aspects, we can just reuse the OCL (Object Constraint Language), hence providing means for describing partial functions or relations by means of invariants, pre- and postconditions.

Level 3 In the description of a primitive component type we include an abstraction of its behavior, based on the UML2.0 State-Chart formalism. Since composite component types must delegate all their ports, they do not contain any behavior.

Level 4 Since our extra-functional contracts would be used on software components with explicit dependency specification, we need means to express a provided contract in terms of required contracts. In the most general case, a component may bind together its provided contracts with its required contracts as an explicit set of equations (i.e. how offered QoS is related to required QoS).

Therefore, the meta-model for Level 4 contracts is made of the following concepts:

- expression of QoS spaces (dimensions, units);
- primitives bindings between these spaces and the levels 1-3 modeling elements (bindings to observable events, conversion from discrete event traces to continuous flows, definition of measures);
- constraint languages on the QoS spaces (defining the operations that can be used in the equations, form of these equations).

The declarative nature of Level 4 contract will make them suitable to various kinds of design-time analysis, including solving them with Constraint Logic Programming techniques [4].

At each level, these meta-models can be enriched with well-formedness rules expressed with the OCL, hence providing some elements of static semantics. Note that all these definitions are made at the component *type* level. UML2.0 indeed rightly distinguishes between component *types* and component *instances*, which are deployed into particular configurations, called *component assemblies*.

4. COMPONENT COMPOSITION

At the syntactic level, component composition is easy: the designer just has to wire required interfaces to provided interfaces of component into a particular component assembly. The overall system can even be closed if we are able to provide a model of the environment, which is formally seen as just another component. Things are getting a little bit more interesting at the semantic level, which should answer the following question "what is the global behavior of the assembly on each of the 4 levels that have been defined above?". At the behavioral level, that's not too difficult a question, because the global behavior of a component assembly can often be described as the parallel composition of the component state-charts, with some synchronization on input/outputs. However the question is more complex for level 4. Ideally, for any components (C_x, C_y), and for any interesting property P of the components, the composition operator o should have the following property: $P(C_x o C_y) = P(C_x) o P(C_y)$

This ideal composition operator might however be difficult to define for two different kinds of reasons.

- real system level composition operators are non trivial. This might be overcome by modeling these composition operators as components relying on a few set of primitive operators. In practice however, it is tedious to reverse engineer complex component frameworks such as .NET or real-time buses.

- the mere nature of the composition meaning depends on the property of interest. For instance, while memory consumption MC is clearly additive among components ($MC(C_x o C_y) = MC(C_x) + MC(C_y)$), this is seldom the case for other quality attributes (e.g; reliability). Furthermore, if we want to allow open-ended level 4 contracts, we need the designer to express the meaning of composition on new quality attributes.

We thus prefer to keep all aspects separate in the semantics, or more precisely to define one semantic domain for each aspect, and then let the designer explicitly define the meaning of the composition with a set of projections o_i of the composition operator on each aspect: $\forall i \in aspect, P_i(C_x o_i C_y) = P_i(C_x) o_i P_i(C_y)$

We propose an operational way of handling this projection, which is based on the reification of the semantic domain of each QoS dimension. As we have seen before with our simple example of memory consumption, the operational semantic of memory can be defined as an integer that can be incremented or decremented as memory is allocated or freed. Then we can use an Aspect-Oriented Programming (AOP) kind of approach to "weave" this memory consumption aspect into the global behavior of the component assembly which is given by the parallel composition of component state-charts. Each time a component service with a memory consumption related contract is called, we get a side effect which is a call to the relevant operation on the reification of the memory QoS dimension. In AOP terms, the contract plays the role of a point-cut, while the state-chart transition holding the service call is the join point.

We then need a tool for (1) describing the operational semantics of extra-functional aspects and (2) implementing this weaving at modeling time, for model analysis purposes. We have developed Kermeta exactly for this kind of problems.

Kermeta [10] is an open source meta-modeling language developed by the Triskell team at IRISA. It has been designed as an extension to the EMOF 2.0 to be the core of a meta-modeling platform. Kermeta extends EMOF with an action language that allows specifying semantics and behavior of meta-models. The action language is imperative and object-oriented. It is used to provide an implementation of operations defined in meta-models. As a result the Kermeta language can, not only be used for the definition of meta-models but also for implementing their semantics, constraints and transformations, as well as weave extra-functional aspects [8] into base models (e.g. memory consumption within the component model).

5. APPLICATIONS

Since the projection of a component assembly onto any QoS dimension can be seen at the semantic level as a system of non-linear constraints that must be satisfied, we can foresee several ways of exploiting this information at design time, with (1) Constraint Logic Programming (CLP) techniques (2) Model Checking and (3) Simulation.

For any QoS dimension, the constraints attached to a component can be translated into a specific CLP-compliant language, using a model transformation techniques as described in [5]. Then for a single component, we can use a CLP(R) based constraint solver to get ranges for admissible values for the QoS properties of the component. It might also allow an early detection of incompatibilities among component with respect to QoS properties. Similarly, since the component assembly is being seen a a complex system of constraints, it can be solved in two directions, either bottom-up or top-down. Knowing the value ranges for QoS properties of the deployment platform, the system of non-linear constraints can be solved bottom-up to obtain end-to-end QoS value ranges. Conversely, based on wanted operational value ranges for QoS properties of the component assembly, the system of non-linear constraints can be solved top-down to obtain dimensioning information for the deployment platform.

The idea of using model-checking techniques is to run an exhaustive co-simulation of the various semantic domains. Since the semantic domains of the QoS dimensions have been reified and woven into the global behavior of the component assembly, QoS dimensions such as memory consumption are now part of the global state of the system as seen by a model checker. Since the model checker needs no knowledge at all of this fact, we can easily reuse off-the-shelf model checkers and obtain results on e.g.; the exact bounds on memory consumption of the component assembly. However, it is clear that if our model checking tool relies on enumeration techniques for the state space exploration, we are bound for trouble if our QoS semantic domains are based on unbounded integers or even worse, floating point real numbers. This limitation can be somehow overcome either by bounding QoS semantic domains, or by using recent progress on symbolic model checking. In the latter case, it would however limit our level 4 contracts to constraint expressed with simple arithmetics.

Simulation can be seen as a non-exhaustive exploration of the component assembly semantic domain, again including the reification of the QoS dimensions. Then, given a sets of operational profiles, we can get interesting statistics on typical distributions of end-to-end QoS values.

We actually see these various techniques as complementary. Once the designer has described her component assembly, then a set of complementary analysis can be performed depending on the characteristics of the application, as well as the availability of tools.

6. RELATED WORKS

In the Component-Based Software Engineering community, the concept of predictability [6] is getting more and more attention, and is now underlined as a real need (see for instance Predictable Assembly from Certifiable Components (PACC) initiative promoted at the SEI [7]). At modeling level, the Object Management Group (OMG) has developed its own UML profiles for QoS and for schedulability, performance and time specification, and it is still working in this domain with the MARTE RFP. In these works however, the semantic domain of extra-functional properties is either hard-coded or implicit. The interest of our approach is to make their semantics both open-ended and explicit at the meta-model level.

In most of these approaches, a QoS property is specified as a constant: they do not allow the specification of QoS properties dependency relationships. In contrast, Reussner proposes parameterized contracts [11]: the set of available services provided by a component depends on its required services that the context can provide. We actually follow the same line, just making it more flexible an open-ended with executable meta-modeling techniques.

The Metropolis meta-model [1] also allows capturing extra-functional aspects of design by so-called quantity managers, and provides means for declarative specification of extra-functional constraints through its constraints logic. Our approach follows the same line, but starts from a different context where component contracts are explicitly modeled to allow assume/promise reasoning.

7. CONCLUSION

In the Component-Based Software Engineering community, the concept of predictability is getting more and more attention: how component technology can be extended to achieve predictable assembly, enabling runtime behavior to be predicted from the properties of components. We have proposed to handle open-ended modeling of extra-functional aspects of real-time and embedded systems, based on meta-modeling techniques and Model Driven Engineering. We let the designer define as many modeling sub-languages as needed for describing QoS contracts. These sub-languages are defined as executable meta-models: their abstract syntax are defined as plug-in extensions to the UML2.0 meta-model, their static semantics are given with a set of OCL constraints, and their dynamic semantics is expressed with Kermeta in order to be woven into the base behavioral model of the component assembly. Once the semantic domains of component contracts has been reified this way, the designer can use off-the-shelf tools (such as model-checkers or constraint solvers) to perform various kinds of design time analysis.

REFERENCES

[1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Paserone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36(4), April 2003.

[2] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 13(7), July 1999.

[3] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benot Langlois, and Damien Pollet. Reflective model driven engineering. In G. Booch P. Stevens, J. Whittle, editor, *Proceedings of UML 2003*, volume 2863 of *LNCS*, pages 175–189, San Francisco, October 2003. Springer.

[4] Olivier Defour, Jean-Marc Jézéquel, and Nol Plouzeau. Applying CLP to predict extra-functional properties of component-based models. In J. S. de Boer, editor, *Proceedings of Logic Programming: 20th International Conference, ICLP 2004*, number 3132 in LNCS. Springer Heidelberg, September 2004.

[5] Olivier Defour, Jean-Marc Jézéquel, and Nol Plouzeau. Extra-functional contract support in components. In *Proc. of International Symposium on Component-based Software Engineering (CBSE7)*, May 2004.

[6] Aagedal J.O. *Quality of service support in development of distributed systems.* PhD thesis, University of Oslo, Dept. Informatics, March 2001.

[7] Wallnau K. Volume iii: A technology for predictable assembly from certifiable compo-nents. Technical Report CMU/SEI-2003-TR-009, SEI, 2003.

[8] Jacques Klein, Loic Hélouet, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *proceedings of the 5th International Conference on Aspect-Oriente d Software Development (AOSD'06)*, Bonn, Germany, March 2006. ACM.

[9] B. Meyer. Applying design by contract. *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, October 1992.

[10] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[11] Reusnerr R.H., Schmidt H.W., and Poernomo I.H. Reliability prediction for component-based software architecture. *Journal of Systems and Software*, 66:241–252, 2003.

[12] C. Szyperski. *Component software, beyond object-oriented programming.* Addison-Wesley, 2nd edition, 2002.