Contract spaces for trusted components

Jacques Malenfant, Noël Plouzeau, Jean-Marc Jézéquel

Contract Spaces for Trusted Components

J. Malenfant*, N. Plouzeau and J.-M. Jézéquel
Triskell Project, INRIA/IRISA.

Abstract

In Szypersky's vision, components have explicit required as well as offered interfaces. This allows for the expression of dependencies of offered interfaces upon required ones. As we believe in contracts to develop trusted components, this vision raises the issue of dependencies between offered and required contracts. In this position paper, we propose several constructs to explicit these dependencies: *λ-contracts*, *contract dependency clauses* and *functionally dependent properties*. In our view, these constructs provide for the definition of contract abstractions where offered contracts can be computed (deduced) from satisfied required constraints, therefore producing *contract spaces* rather than single contracts. We discuss the requirements and uses of such a component contract models.

1 Introduction

Among the various ways to provide more trust into components, design by contract is called to play a crucial role. Experience shows that contracts for objects can drastically enhance the overall quality of the produced code, and the confidence we can have over them. But there is more in contracts for components than what appear in contracts for objects. In our view, contracts in the component world do not bind the same entities and make them play the same role as in object-oriented programming.

In object-oriented programming, the encapsulation principle requires that an object masks to its client most of its implementation. A contract therefore directly links the establishment of the preconditions by a caller to the fulfillment of the postconditions by the callee [Mey97]. Any dependency of the callee upon its own providers is appropriatly masked to the client. The callee takes care of them otherwise, i.e. by defining the necessary contracts with its providers so that it can fulfill its duty whenever the caller has done its part of the contract.

We conjecture and argue that this encapsulated vision no longer holds in the component world. In Szypersky's vision [Szy98], components not only publish offered interfaces but they also explicit their required ones. This makes the point that component-based software is not build merely as layers of encapsulated services, but rather as first-class networks of interconnected components where the pieces (components) take engagements only insofar as all their clients *and* providers explicitly fulfill their own ones. These mutual engagements must take over the traditional DbC client/provider relationships in order to have a complete notion of contracts for components. Furthermore, we argue that enlarging the notion of contract will provide for other important aspects in the component world, such as configurability, quality of service and dynamic negociation of contracts.

*Université de Bretagne sud

We also put forward the idea of explicitly defined parameter-based, deductive and functional dependencies among contracts, which can support the expression of families of contracts, *contract spaces*, for components. The fundational arguments are to provide more flexibility and reusability by having trusted components that offers, instead of a single contract on each interface, a space of possible contracts depending on a space of required contracts. At binding-time, the exact required contracts obtained by a component from its suppliers will determine its precise offered contracts.

2 Varying contracts

A good component model has the ability to define adaptable and configurable components. An adaptable component (probably a pleonasm) must be prepared to provide different levels of service, depending on the capability of its underlying run time environment. For instance, a numerical computation operation may produce different results, depending on the kind of floating point unit available on the computer running the component instance.

Consider an object-oriented possible solution for an object operation which computation results depend on the underlying FPU capability:

```
op Calc::compute_this(x:Float)
pre ((x > 1e-20) and good_FPU()) or ((x > 1e-10) and not good_FPU())
post ((x > 1e-20) and good_FPU()) implies Result = f1(x)
     and ((x > 1e-10) and not good_FPU()) implies Result = f2(x)
```

The `good_FPU()` is a crude way for making the component adaptable to its run-time environment by selecting the appropriate algorithm given the available capability. The predicate presumably probes the run time to determine it, and the idea here would be to test whether there is a FPU on the machine, and if so, to what standard is this FPU adhering. In a full fledged component model, such dependencies would be explicited directly by required interfaces. Using this feature, offered contracts can be made dependent upon the required contracts. Using parameterized contracts, i.e. λ-contracts, our example could be stated as:

```
interface Calc (f:FPU)
  op Calc::compute_this(x:Float)
    pre  ((x > 1e-20) and f.good_FPU()) or ((x > 1e-10) and not f.good_FPU())
    post ((x > 1e-20) and f.good_FPU()) implies Result = f1(x)
         and ((x > 1e-10) and not f.good_FPU()) implies Result = f2(x)
end Calc

interface FPU
  op add(...)
  op good_FPU()
end FPU

component Example
  offered port calc:Calc(fpu)
  required port fpu:FPU
end Example
```

Without loss of generality, we have considered a syntax where interfaces and contracts are defined in a single syntactic construct. First, note that the contracted interface `Calc` has a parameter `f` of type `FPU` which refers to another contracted interface. This is an illustration of what we call a λ-contract. Second, the contract explictly bases the pre- and postconditions on the properties (`good_FPU()`) of the provider. Such properties attached to providers would have to be specified using some form of typing to be included in a good contract language.

3 Logical deduction of contracts

λ-contracts exploit the explicit required interface to make an offered contract depend upon a properety of the provider of a required interface. In the above example, however, it is used to define a unique contract that blurs together all possible alternatives. A more declarative way to express dependencies among contracts is to define several λ-contracts on required and offered interfaces and then to state, using *contract dependency clauses*, what contracts are obtained on offered interfaces given the ones fulfilled on the required interfaces. If we consider our FPU example again, we would write:

```
interface Calc(f:FPU)
  op Calc::compute_this(x:Float)
  with contract High_precision given f with contract Good_FPU
    pre  (x > 1e-20)
    post Result = f1(x)
  or contract Low_precision given f with contract No_FPU
    pre  (x > 1e-10)
    post Result = f2(x)
end Calc

interface FPU
  op FPU::add(...)
  with contract Good_FPU
    pre  good_fpu()
  or contract No_FPU
    pre  not good_fpu()
end FPU
```

Such a form of logical expression of dependencies would work hand in hand with a deduction engine to propagate at binding-time fulfilled required contracts to offered ones. Backtracking to choose among several possible choices could be turned into a form of run time "negociation" of contracts.

4 Functional variation

We are also interested in quality of service contracts, where functional and non-functional properties are considered equally important, and therefore contracts are extended to deal with both kinds of properties. QoS properties are known to be either qualitative (secure/non-secure, safe/unsafe, fault-tolerant/non-fault-tolerant, etc.) or quantitative (number of transactions per second, number of megabytes transmitted per second, etc.). In case of quantitative

QoS properties, another form of dependency appears when a parameter of an offered contract depends upon a parameter in a required contract.

Consider a component that offers an interface to send blocks of data over the network. Blocks of data are cut into packets to be sent using a required interface to network services. Obviously in this case, a QoS contract put on the offered interface in terms of the number of data blocks sent per second is directly proportional the number of packets transmitted per second (provided that the processing power and the memory are sufficient to take care of the necessary breaking of blocks into packets, which we neglect here for simplicity). We could write:

```
interface BlockTransmitter(p:PacketTransmitter)
op BlockTransmitter::send(b:Block)
with contract BlockRate given p with contract PacketRate
   dimension blockRate > (p.packetRate * Packet.size) / Block.size
end Calc

interface PacketTransmitter(ns:NetworkServices)
op PacketTransmitter::send(p:Packet)
with contract PacketRate given ns with contract Bandwith
   dimension packetRate = ns.bandwith() / Packet.size
end FPU
```

where `bandwith()` returns the currently available (allocatable) network bandwith. Dimensions used in these contracts are derived from a concept introduced in QML [FK98]. The contract `BlockRate` is actually transforming (using an arithmetic expression) a required contract dimension value into an offered contract one that is meaningful to its clients.

Expressing this kind of dependencies again allows for more flexibility and reusability. One cannot be satisfied by a solution where contracts would be decided statically, once and for all. A good component model must allow for expressing the level of offered contracts as a function of the level of required ones. Not only is this good software engineering, it would enable the form of compositional reasoning about non-functional properties of component-based systems from their constituent components [BBB+00], but also to do performance estimation by propagating constraints bottom-up (compute throughput from available resources) or system dimensioning by propagating them top-down (compute the level of resource needed to sustain a given level of throughput). Figure 1 illustrates these.

5 Conclusion: a glimpse of the new order

Contracts for trusted components impose a shift from the traditional object-oriented vision of contracts. In this traditional vision, the emphasis is put on the caller/callee relationship, where the former establishes preconditions in order for the latter to fulfill its contracted postconditions. Implicit in this scheme is the idea that the sole responsible for establising the necessary conditions for the requirements of a service is the caller. The service implementation takes care of other requirements by establishing its own contract with its own service providers, masking them to its client.

In the component world where contracts are defined on both offered and required interfaces, the need for an explicit statement of dependencies between these contracts appear.

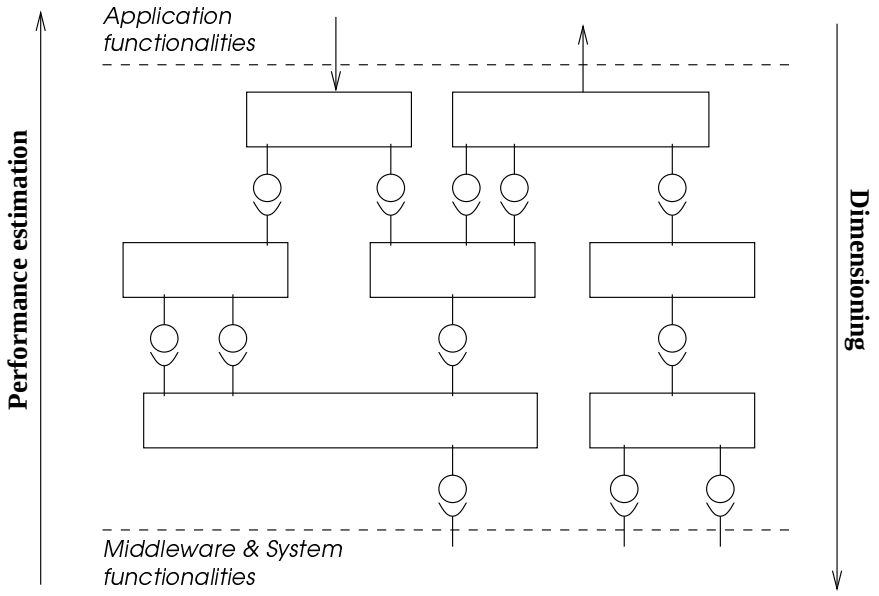

Figure 1: Uses of contract spaces for dimensioning and performance estimation.

Not only because they are part of the specification of the component, but more importantly because it allows for the definition of *contract spaces*, i.e. family of contracts where the level of engagement obtained by a component from its providers can be – often functionally – translated into a level of engagement it can offer to its client.

This capability of λ-contract, contract dependency clauses and functionally dependent properties to adapt to the run-time level of available resources gives more flexibility and accounts for what is called QoS negociation in traditional, network-oriented QoS. It also paves the way to compositional reasoning about non-functional properties of component-based system, such as performance estimation (offered throughput for a given level of available resources) and dimensioning (necessary level of resource to sustain a given throughput).

In this position paper, we have presented exmaples using a loose mockup syntax. The definition of a full-fledged contract language is part of our on-going research at IRISA.

References

[BBB+00] 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, May 2000.

[FK98] S. Frølund and J. Koistinen. Quality-of-Service Specification in Distributed Object Systems. *Distributed System Engineering*, 5:179–202, 1998.

[Mey97] B. Meyer. *Object Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[Szy98] C. Szyperski. *Component Software – Beyond Object-Oriented Programming.* Addison-Wesley, 1998.