How common Specification Frameworks for Software Components deal with Inputs in functional Specifications *

Uwe Keller
uwe.keller@deri.org
Digital Enterprise Research Institute (DERI) Innsbruck, Austria

Version 1.0

December 30, 2004

Abstract

We briefly overview how several existing frameworks for the specification of software components deal with inputs in component specifications. Finally, we propose that capability specifications in WSMO should follow the same principle. Futhermore, we suggest to have a closer look at the considered specification frameworks and evaluate their use for service capability specification in WSMO.

1 Introduction

Software components provide specific and well-defined services to their clients. They are technical means for providing access to services in a well-defined way: clients can use a fixed interface which allows them to interact with the component. By interacting with the component the client is able to consume some specific service. After successful interaction with the component[1] the client should have consumed the intended service and a well-defined state for every party as well as the environment should be reached. Hence, the delivery of a service corresponds to the successful and completed execution of a software component.

*This work is supported by the European Commission under the projects DIP, Knowledge Web, SEKT, SWWS, and Esperonto.

[1] Neglecting errors due to the environment.

A software component does not provide a single service, but instead a class (or set) of distinct and conceptually related (or similiar) services. The precise service to be delivered to clients is determined by the clients themselves: During interaction with the software component they provide certain *input values* to the service which specify the precise details of requested service.

In the simplest case, this is done in a single shot when invoking the component which traditionally can be seen as some sort of procedure call. A classical example for this case are stateless software components, where the client necessarily has to specify all the required input (and thus the specific service to be delivered) at invocation time. Using a more sophisticated and general model one can as well assume some more elaborate interaction between the client and the component during which the precise service to be delivered is determined (or negotiated). The interaction between client and component is conceptually (and technically) realized by means of a series of messages which are exchanged. Here, the single messages might be interdependent and the input values can be spread over a sequence of messages. Obviously, the procedure call model is a (very simple) special case of the message-oriented conversation model.

The latter view is useful in modern object-oriented modelling frameworks like the Unified Modelling Language (UML) and actually important for the modelling of Web Services which can be considered as a specific technology enabling systems of interoperable software components.

There are two particular aspects of a component which are particularly important for a client and thus should be documented in some way:

- What services can actually be provided by the component? (*Functional Specification*)
- How do I have to interact with the component in a proper way such that I will be able to consume an intended service? (*Behavioural Specification*)

The most basic and simple way of documentation is the use of natural language prose. The drawback of this sort of documentation is that its meaning easily is ambiguous and not accessible for machines. If one is interested in unambiguous descriptions which are machine processable then formal languages with a mathematically well-defined semantics have to be used. Prominent examples of such formal languages are for instance logics or process algebras.

In the following, we are interested in specifications of the first type (i.e. functional specification of software components), in particular

the way inputs are used in the formal modelling of the capability of a component.

2 Approaches in existing Frameworks for functional Specifications

There are several specification frameworks which are used in academia and the industry, for instance Eiffel, UML/OCL , VDM, B, Z, Larch/LSL, SDL, RAISE/RSL. In the following we will have a look at some of them and explain how they deal with functional specifications of software components and particularly how input values are represented in these specification.

Functional specification (or capability specifications) of web services in WSMO follow the precondition/postcondition description style that has been invented by Hoare and is the basis of Hoare-style software verification calculi.

Since in WSMO functional specifications are based on pre- and postconditions, we restrict ourselves in the following to specification frameworks that are based on pre- and postconditions. This excludes for instance the well-known formalisms of *Abstract State Machines* (ASMs) as well as the *B Method*.

2.1 Eiffel

Eiffel has been invented by Bertrant Meyer as a systematic approach to object-oriented software construction. It can be considered as a programming language with specific language constructs for the functional specification of methods (that is pre- and postconditions) as well as class invariants. Eiffel embodies the so-called *Design by Contract* principle, where the signature of a interface as well as the functional specification of the interface are considered as a contract between the client using the interface and the provider of the concrete implementation of the interface. None of the parties is allowed to break the specified contract, or more precisely contracts can be checked in a concrete system during runtime and broken contracts are detected and reported. Together with a suitable tool environment and a process recommendation, Eiffel can be considered not only as a programming language but as a formal methodology for software development.

The notion of a contract. In human affairs, contracts are written between two parties when one of them (the supplier) performs

some task for the other (the client). Each party expects some benefits from the contract, and accepts some obligations in return. Usually, what one of the parties sees as an obligation is a benefit for the other. The aim of the contract document is to spell out these benefits and obligations.

A tabular form such as the following (illustrating a contract between an airline and a customer) is often convenient for expressing the terms of such a contract:

	Obligations	Benefits
Client	(Must ensure precondition): Be at the Santa Barbara airport at least 5 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.	(May benefit from postcondition): Reach Chicago.
Supplier	(Must ensure postcondition): Bring customer to Chicago.	(May assume precondition) No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price.

A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.

The same ideas apply to software. Consider a software element E. To achieve its purpose (fulfil its own contract), E uses a certain strategy, which involves a number of subtasks, $t_1, \ldots, t_n$. If subtask t_i is non-trivial, it will be achieved by calling a certain routine R. In other words, E contracts out the subtask to R. Such a situation should be governed by a well-defined roster of obligations and benefits – a contract.

Assume for example that t_i is the task of inserting a certain element into a dictionary (a table where each element is identified by a certain character string used as key) of bounded capacity. The contract will be:

	Obligations	Benefits
Client	(Must ensure precondition): Make sure table is not full and key is a non-empty string.	(May benefit from postcondition): Get updated table where the given element now appears, associated with the given key.
Supplier	(Must ensure postcondition): Record given element in table, associated with given key.	(May assume precondition) No need to do anything if table is full, or key is empty string.

This contract governs the relations between the routine and any potential caller. It contains the most important information that can be given about the routine: what each party in the contract must guarantee for a correct call, and what each party is entitled to in return.

An example of a interface specification. In order to benefit and support the concept of contracts, contracts need to be formalized and explicitely documented. In the spirit of seamlessness (encouraging us to include every relevant information, at all levels, in a single software text), in Eiffel the routine text is equipped with a listing of the appropriate conditions.

Assuming the routine is called `put`, it will look as follows in Eiffel syntax, as part of a generic class `DICTIONARY [ELEMENT]`:

```
class DICTIONARY [ELEMENT]
 ...
     put (x: ELEMENT; key: STRING) is
                        -- Insert x so that it will be
                           retrievable through key.
             require
                        count <= capacity
                        not key.empty
             do
                        ... Some insertion algorithm ...
             ensure
                        has (x)
                                   -- has(x) returns true afterwards
                        item (key) = x
                                   -- x can be retrieved with the key
                                      using the item method
                        count = old count + 1
                                   -- the dictionary has one more
                                      entry afterwards
             end
 ...
    get (key: STRING):ELEMENT is
                        -- Get the element which is retrievable
                           through the given key.
             require
                        not key.empty
                        has_key(key)
             do
                        ... Some retrieval algorithm ...
                        Result := ...
                        ...
             ensure
                        Result = item (key)
                        count = old count
```

```
            end
end
```

Let's have a look at the `put` specification: The require clause introduces an input condition, or precondition; the ensure clause introduces an output condition, or postcondition. Both of these conditions are examples of assertions, or logical conditions (contract clauses) associated with software elements. In the precondition, count is the current number of elements and capacity is the maximum number; in the postcondition, has is the boolean query which tells whether a certain element is present, and item returns the element associated with a certain key. The notation old count refers to the value of count on entry to the routine.

In the `get` contract we have another distinct keyword, namely `Result`. This keyword refers to the object or value which is returned at the end of a successful execution of this method. The keyword is the same as the one which is used in the programming language part in the `do` clause.

The assertion language in Eiffel has been designed in such a way that the (boolean) conditions can be effectively evaluated during runtime. That means that predicates ususally refer to boolean methods. The assertion language seems to be less expressive than assertion languages in other specification frameworks that are not directly based on a programming language and the idea of runtime checking of contracts.

From this prototypical example, we can observe the following:

- Input variables are explicitly declared in the contract (more precisely in the signature part).
- There is an explicit representation of the output value (`Result`) which allows to refer to the return value in the postcondition
- By using the keyword `old`, one can refer to the value of some (boolean) property before the execution of a method. This keyword can only be used in post conditions.

2.2 The Unified Modeling Language and OCL

The Unified Modelling Language is a graphical language for the modeling and description of systems, in particular software systems. It is a collection of around eight different diagramm types which capture different perspectives on the system under consideration like the logical system structure (Class Diagramms), typical usage scenarios (Use Case Diagrams), interactions between the elements the system consitutes of (Interaction Diagrams), the lifecycle of the single elements (State Diagrams), the physical structure of the system (Deployment Diagramms) and so forth. The UML evolved and unified different

preexisting and competing proposals for object-oriented modeling languages for software systems and became the standard object-oriented modeling language in the software engineering domain maintained by the Object Management Group (OMG).

The Object Constraint Language (OCL) is an integral part of the UML Standard since UML version 1.1. UML has been equipped with OCL in order to allow modelers to express unambiguously nuances of meaning that the graphical elements of UML can not convey by themselves. An important (and even the first) application for OCL was the Specification of the UML Metamodel itself which was recognized as being ambiguous in the natural language based specification of UML version 1.0. It is a formal language for the specification of functional behaviour of the single elements of a software system as well as global constraints on valid system states. OCL supports preconditions, postconditions for functional specifications of methods and class invariants for the description of global constraints on the system state.

It is important to notice that UML/OCL models are not bound to any specific implementation language. Indeed, they are implementation language independent.

Context of OCL Constraints. OCL expression have no meaning by themselves. In general, they can only be used to constrain the standard semantics of modeling elements of the UML and to detail out the precise meaning of these standard elements in a specific model (if such an derivation from the standard semantics of UML and its elements is needed). Thus, OCL Expression always refer to specific elements in a given UML Model. Such a UML model usually is a class diagram. Other diagram types such as state diagrams are only weakly supported by OCL at present. When a modeler describes a constraint, he must always specify to which element (for example which class) the constraint refers. This element is called the *context* element of a constraint. For some constraints, the specific choice of the context is not relevant from a semantic perspective. Some of the build in stereotypes can be considered as graphical shortcuts of (stereotypical) OCL Constraints, for instance the `<<complete>>` or `<<distinct>>` stereotypes of inheritance relationsships.

OCL Constraints OCL is a typed language which can be consider as some sort of ,,object-oriented predicate logic" whereby quantors can only quantify over finite domains[2]. The readability of the language for the average software engineer had been emphasized during the development of the language. For instance, path-expression which are

[2]In fact, it has been shown by Peter Schmitt that the logic underlying OCL is strictly more expressive over finite domains than First-order predicate logic

common in object-oriented programming and specification languages are supported by OCL as well.

OCL supports as constraints mainly invariants on class diagramms as well as fuctional specifications of methods by preconditions and postconditions. Roughly spoken, all these constraints are basically a boolean OCL expression (the assertion language) combined with a context declaration. In the case of method specifications, the context declaration includes the method signature.

The general template of a functional specification of a methods looks as follows (whereby the method takes parameters $p_1, \ldots, p_N$ with the respective types $T_1, \ldots, T_N$ as input parameters and (possibly) returns a value of type $ResT$):

```
context (cxtName)?
  className::methodName(p1:T1,...,pN:TN)(: ResT)?
     (pre  (constraintName)? : BooleanOCLExpression)?
     (post (constraintName)? : BooleanOCLExpression)?
```

Optional parts of the template are denoted by `( ... )?`

In preconditions, the parameters can be used $p_1, \ldots, p_N$ like (OCL-)variables. The semantics of these variables is identical to universally quantified variables. The quantors of these variables are not specified explicitly but determined implicitly by the declaration of a method signature. The declaration allows a tools that deals with the OCL constraints to distinguish between input values and other properties from class diagrams (like attributes of classes or relations between classes)

In postconditions, the input parameters $p_1, \ldots, p_N$ can be used in the very same way. Moreover there are two additional syntactic constructs which can be used in postconditions only: (i) In order to be able to describe the result value which is returned by a method one can use the literal (or keyword) `result` and (ii) In order to be able to refer to values of properties of the system in the prestate of the method call for each property `prop` (for instance, predicates and attributes, but not OCL variables) one can use the property `prop@pre`.

An example of a interface specification. Lets consider a Role-based Access Control Scenario illustrated in Figure 1: A user works and can take actions in the context of a user session. A user can participate in multiple sessions during each point in time. Each user has a set of roles assigned, these are the roles the user principally can realize. Every session has a single user assigned and a set of roles which the respective user of the session currently is allowed to realize. Each role provides a set of assigned permissions as well as a set of persons which are registered for and allowed to realized this role.

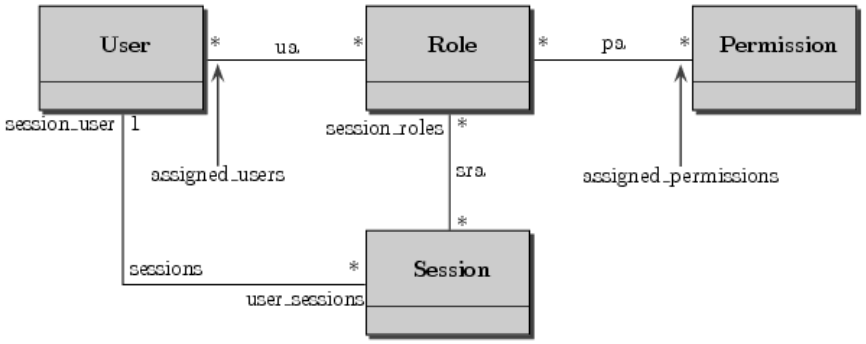

Figure 1: A simple UML diagram for a Role-based Access Control Scenario.

This model can be used as a starting point to describe a system which supports the administration and implementation of access-control policies based on roles.

Now, lets assume that class `user` has a method `assignUser(r:Role)` which can be used to assign a new role to a instances of class `user`. The method registers the user for this role only if the user does not already have the role and returns the boolean value `true` in this case.

A formal specification of the functionality of this method in OCL could be as follows:

```
context u:User::assignUser(r:Role):boolean
  pre: not(r.assigned_users->includes(u))
     -- the user u (for which the method has been called)
     -- does not already have the given role assigned.

  post: r.assigned_users =
        r.assigned_users@pre->including(u)
     -- After the execution the set of assigned users
     -- is precisely the one we had before the invocation
     -- including the user u (for which the method has
     -- been called)

       and result = true
     -- return the boolean value 'true'.
```

OCL supports in particular collection types (like the set `r.assigned_users` of assigned users to a role `r`) and corresponding standard operations

(like union, intersection, contains, etc). In particular it is possible to iterate through such collections and quantify over such collection.

For instance, if we would have to specify a method which adds a creates user instance for a given name to the system in case that a user with the same name does not already exist, then we could use the following OCL specification:

```
context User::addUser(c:Name)
    pre: User.allInstances->forAll(u1| u1.name <> c)
     -- There is no instance u1 of class User
     -- with the given name c

    post: User.allInstances->exists(u1|
             u1.name = c and
             User.allInstances@pre->excludes(u1) and
             User.allInstance =
             User.allInstance@pre->including(u1)
          )
     -- There an instance u1 of class User
     -- with the given name c
     -- which has not been there before the method call
     -- and the extension of class User has only
     -- changed by this specific instance u1
```

In contrast to Eiffel, the assertion language if OCL has a more abstract flavour and is less oriented towards concrete implementations of the model. In particular associations can be realized in general in various ways in a concrete implemenation of the same UML model. OCL abstracts from these differences and treat associations (and some other modeling elements of UML) as explicit modelling elements which do not have to have directly correspondent elements in a programming language.

3 Conclusion

We have briefly surveyed some of the most-popular and widely-used frameworks for functional specifications of software components in the industry as well as academia.

Besides the differences in the syntactic details and the background of the single specification frameworks, we can extract basically the following general pattern which is independent from the specific assertion language that is used in the specification frameworks:

- There are syntactic constructs to refer to prestate-values, e.g. `old property, property@pre`. These constructs can only be used in post conditions.
- Some syntactic constructs to refer to the output, e.g. `Result, result`

- Input values are declared explicitly and are usually interpreted variables. On a semantic level these variables are universally quantified. Their quantors are not explicitly declared by determined on a meta-level by the semantics of a functional specification based on pre- and postconditions. The scope of these variables
- A component specification is a logically indivisible unit. The functionality description depends on both, pre- and postconditions, the single logical expressions which describe these conditions themselves are meaningsless (outside the functional specification as a coherent unit)

Finally, we suggest to have a closer look at the considered specification frameworks and evaluate their use for service capability specification in WSMO.

We believe that an interesting an important point that still needs clarification is how capability descriptions actually relate to choreography descriptions on an intuitive as well as a formal level. It seems that this aspect has not been discussed extensively in the literature so far.

In formal descriptions of object-oriented systems, invariants represent (besides the functional specifications of the single methods) a major descriptive means of the required properties of the system to be realized from the composition and interaction of instances of the single classes. If WSMO is considered as a formal framework for the description of components of a (possibly dynamic) software architecture, then this should not be a problem. On the other hand, if WSMO is interpreted in a more general way as describing the properties of the overall (dynamic) system, then WSMO certainly must be extended to represent invariants as well.

4 Future Work

We propose to extend this document to a technical report which gives an overview on existing frameworks for the functional and behavioral specification of software components and investigates the applicability of the existing techniques within the Semantic Web Services arena, specifically the Web Service Modelling Framework (WSMO).

Moreover, we propose to definitely investigate the use of UML as a emcompassing and extensible graphical standard notation for systems modelling that can be tailored to specific application domains using a well-defined extension mechanism called *UML Profiles*.

Future versions of this document will include further software specification frameworks like Z, VDM, Larch/LSL, SDL, RAISE/RSL if this is considered to be necessary by the WSMO working group.