On the Abstraction of Objects, Components and Interfaces

Guy Genilloud
Swiss Federal Institute of Technology of Lausanne (EPFL)
EPFL-ICA, CH-1015 Lausanne, Switzerland
guy.genilloud@epfl.ch

Paper to be presented at the OOPSLA Workshop on Behavior Semantics, Nov. 1, 1999, Denver, Colorado.

Abstract

There is a wide consensus that "an object explicitly embodies an abstraction," but the exact meaning of this statement is not necessarily well understood. This paper argues that the abstraction embodied by an object (or a component) implementation is nothing else than its specification. This implies that abstraction does not appear automatically, it must be carefully constructed and maintained. The paper goes on to show, with the support of an example, that defining contracts for an object's operations does not necessarily result in a complete specification for this object (for example, specification invariants are a different concept from invariants associated to operation contracts). Finally, the paper shows via an example the difference between object specifications and interface specifications.

1 Introduction

The concept of abstraction is often advanced as a benefit of object-orientation, but is poorly understood. Section 2 of this paper argues that the abstraction embodied by an object (or a component) implementation[1] is nothing else than its specification.

However, the notion of an object's specification is itself poorly understood. Many people seem to believe, having read or heard about Bertrand Meyer's historical paper [Meyer92], that an object's specification is simply the set of contracts attached to its methods. For example, it is regularly discussed at the OMG to complement the specification of operations in IDL with pre- and postconditions, for specifying their semantics (i.e., their perceived behaviour). Many delegates favour this practice, which would indeed be useful, but they do not seem to realise the limitations of this specification technique. Section 3 discusses the true nature of a specification. By revisiting a simple directory example originally produced by Clemens Szyperski [Szyperski98], we show that contracts are not sufficient. We also show that the Fusion specification technique succeeds where operation contracts have failed. Incidentally, we find that notifications of state changes are actions which are logically separated from operation activities (even though they may be implemented in some cases as belonging to methods).

The notion of invariant is essential in many specification techniques. Section 4 shows that specification invariants are a different concept from invariants associated to operation contracts. There is an important distinction, and people should become aware of it.

Finally, Section 5 shows the difference between objects and interfaces: their specifications may be dif-

1. For the scope of this paper, we consider a narrow concept of an object implementation: the code in a programming language to be executed by a computer. Thus, we do not consider the case where objects are implemented by humans, etc.

ferent. This distinction is pertinent to Szyperski's directory example.

1.1 Methods vs. operations vs. actions

This paper is founded on the concepts of action, method, and operation. We use those terms with their usual OOP meaning, as reflected in their UML definitions [UML99]:

> *Action*: The specification of an executable statement that forms an abstraction of a computational procedure. An action typically results in a change in the state of the system, and can be realized by sending a message to an object or modifying a link or a value of an attribute.
>
> *Operation*: A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.
>
> *Method:* The implementation of an operation. It specifies the algorithm or procedure associated with an operation.

Regarding the concept of action, we consider the composition of several actions to be an action. For example, a sequence of two actions `<a>` and `<b>` is a *composite action* (that we may denote `<a>;<b>`). The opposite of a composite action is an *atomic action*: an action is *atomic* at a given level of abstraction if it cannot be subdivided at that level of abstraction [X.902]. In our example, `<a>` and `<b>` appear to be atomic actions (we don't know how to subdivide them), but certainly not `<a>;<b>`. However, as we shall see, a composite action such as `<a>;<b>` may well correspond to an atomic action at a higher level of abstraction. An action is an *interaction* if it involves more than one object. For example, both the call and the return actions of an operation are interactions (they involve both the caller and the callee).

For the purpose of this paper, an operation is an action which is initiated by a *call action* and which is immediately followed by a *return action* — we say that an operation *returns* or *terminates* with the occurrence of its return action, but we consider neither the call nor the return action to belong to the operation. The level of abstraction is that of the specification of the object, not its implementation. We will see that an operation is not necessarily an atomic action, even at that higher level of abstraction.

A method is the implementation of an operation (and conversely, this operation is the specification of that method). In other terms, a method is an action (often a sequence of actions) which corresponds to an operation, but at a lower level of abstraction (that of the implementation). The call and the return actions of the method correspond to the call and the return actions of its corresponding operation.

It should be noted that UML is extremely vague regarding how an operation is to be specified. The UML meta-model includes two mechanisms for this purpose. The first mechanism is just a `specification` attribute attached to the operation metaclass; this attribute is of type `String` — there is no indication about what to write in that attribute. The second mechanism consists in associating pre- and postcondition constraints to operations. But again, there is little indication about what constraints to write for an operation. In fact, there is no requirement that any constraints be defined for an operation. Moreover, UML is silent on the interplay between the `specification` attribute and the pre- and postcondition constraints.

2 Abstraction as embodied by objects

As observed by Alan Snyder and the OMG (Object Management Group) in the early 90's, there is a wide consensus that "an object explicitly embodies an abstraction" [Snyder93]. However, the exact meaning of this statement is not necessarily well understood.

2.1 Several definitions

The RM-ODP defines abstraction in this way:

> "The process of suppressing irrelevant detail to establish a simplified model, or the result of that process" [X.902].

This definition is compatible with most uses of the term abstraction. It certainly captures the right idea

with respect to the abstraction embodied by an object, but it does not tell the whole story in that specific case.

The UML definition of abstraction is more specific to our concern:

> "The essential characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative to the perspective of the viewer" [UML99].

This definition certainly confirms that each object embodies an abstraction, and that two different object implementations may happen to have the same abstraction (otherwise, every characteristic of an object would be an essential characteristic). Anyone familiar with UML will understand that the essential characteristics include attributes and operations, in particular operations since attributes may be understood to define read and write operations. This interpretation is consistent with the second sentence in the definition. However, we are not fully satisfied with this UML definition, in particular because UML is so vague regarding how operations are to be defined.

In his description of abstraction, Alan Snyder emphasized the difference between *information* and *data*.

> An object explicitly embodies an abstraction that is meaningful to its clients. Although an object may involve data, an object is not just data; the purpose of the data is to represent information associated with the abstraction. [Snyder93]

Abstraction is indeed about separating the concepts of data and information, since data must be interpreted with respect to an implementation, and thus tend to reveal implementation details. However, as observed by Andrew Herbert, the notion of behaviour is also very important:

> An object is more than just information: it provides a **service** to its clients. This service corresponds to the abstraction. Objects do more than just read and write data. The service is carried out by executing code that accesses or manipulates the actual data. The set of services provided by an object is called the **behaviour** of the object. [Herbert93]

In fact, mentioning behaviour is the right idea, but the definition of the behaviour of an object as the set of services it provides is too restrictive — it does not address the fact that some objects may also use other objects services. In this paper, we will use the concept of *behaviour* according to its RM-ODP definition:

> *Behaviour (of an object)*: A collection of actions with a set of constraints on when they may occur.
>
> The specification language in use determines the constraints which may be expressed. Constraints may include for example sequentiality, non-determinism, concurrency or real-time constraints.
>
> A behaviour may include internal actions.
>
> The actions that actually take place are restricted by the environment in which the object is placed. [X.902]

2.2 State vs. data

Many programmers consider an object's state as being its data (including its links to other objects). In fact, it can be said that an object's data represent its state (with respect to a given implementation), but an object's state is slightly different from its data. Two different states may happen to share the very same data — consider for example the state immediately before an object terminates an operation (the object is in a state in which it has the obligation to execute a return action), and the state immediately after that operation was terminated (this obligation was fulfilled, and is not supposed to be repeated). Conversely, two different data may well represent a same state — for example, a set of two elements can be represented by two different lists.

The RM-ODP defines the *state* of an object as "the condition at a given instant in time that determines the set of all sequences of actions in which it can take part" [X.902]. UML defines it as "a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event". Both definitions, while different, are independent of an object's data.

2.3 Abstraction as a specification

The RM-ODP definition of behaviour is well suited for investigating the idea of abstraction as provided by an object[1]. We observe that the *implementation* of an object (i.e., its code specification in a programming language) is a description of its behaviour. That description lacks abstraction in the sense that it reveals implementation details which are not directly nor indirectly observable outside of the object (and thus, which are of no interest to the users of that object). For example, an implementation typically includes sequences of steps (internal actions) which move the object from a well-known state, say A, through intermediary states, to another well-known state of the object, say state B. If none of the intermediary states of the objects is observed by any other activity of the object, it is possible from an external perspective to consider that a single internal action changes the state of the object directly from A to B. This alternative description of the object's behaviour, which thereafter we will call a *specification*, is more abstract than the implementation in the sense that it includes fewer actions and fewer states. Note that a specification may be written after an implementation has been written, and not necessarily before that implementation.

These considerations lead to the following claim: an object or component embodies an abstraction implies that there exist a description of its behaviour (a specification) which is both as precise and more abstract than its implementation[2]. Of course, the specification and the implementation must be *compatible* with one another: whatever observable action is described by one behaviour must be described by the other. Moreover, to every possible state of the specification must correspond at least one state of the implementation (the converse is not necessarily true).

Unfortunately, an object's specification is often (non-intentionally) ambiguous or partially incorrect, when it is provided at all. To understand precisely what an object does and how to use it, programmers must read its implementation and infer its observable behaviour from it. As important and beneficial as it might be, the principle of abstraction remains poorly supported by OO technology and poorly understood by many programmers. For example, the mainstream programming languages offer no support for abstraction beyond the capability to write comments.

The language Eiffel distinguishes itself by allowing programmers to write assertion statements related to preconditions, postconditions, and invariants [Herbert93]. These assertion mechanisms are certainly useful for helping programmers to understand their own code and to debug it (i.e., to ensure that they correctly implement a specification). But they are not entirely suitable for providing a specification, for at least two reasons. Firstly, they need to be written in terms of the data of the object, those very same data which abstraction aims to hide from the users of the object. Secondly, invariant assertions describe a condition on the object's internal state that is required to be true at some point in time in the object execution — this is quite different from a condition that is required to be true for all the states of the object's specification (the true definition of an invariant, as it matters to the users of the object — see Section 4).

3 Specifications are more than operation contracts

There is a common misconception that the behaviour of an object can be expressed simply by stating the pre- and postconditions of its operations. In fact, this is only true for operations that are atomic

1. In UML, behaviour is defined as "the observable effects of an operation or event, including its results". This definition is not useful for discussing the property of abstraction since it does not distinguish between specification and implementation. In UML terminology, a specification and an implementation have the same behaviour. In ODP terminology, they have *compatible* behaviour (other objects cannot observe differences between these two behaviours). See the definition of *behavioural compatibility* in [X.902].

2. In fact, we conceive that some simple objects' implementations are as simple as their specification. However, it is always possible to change an implementation to make it more complex, without changing the corresponding abstraction.

actions (i.e., for operations that change the state of an object directly from one observable state to another observable state). Many operations are not atomic actions, as they invoke other operations during their execution, therefore revealing the existence of extra "observable" states. Moreover, some objects have a part of their behaviour that is not a part of their operations. For example, we will see that notifications of state changes are actions which are logically separated from operation activities (even though they may be implemented in some cases as belonging to methods).

Clemens Szyperski touches on this problem in the chapter 5 of his book [Szyperski98]. We revisit his directory example, making it more object-oriented (we describe methods instead of procedures, and we use interface parameters rather than procedure parameters) and simpler (e.g., only one interface may be notified rather than an arbitrary number of them). None of these changes affect the substance of the issue. To remain close to the original example, we use a pseudo Component Pascal notation which should be readily accessible by most readers.

The example revolves around the methods `AddEntry`, `RemoveEntry`, and `RegisterObserver`, whose contracts are as follows:

```
Method ThisFile (n: Name): Files.File;
(* pre n # "" *)
(* post result = file named n or (result = NIL and no such file) *)

Method AddEntry (n: Name, f:Files.File);
(* pre n # "" and f # NIL *)
(* post ThisFile(n) = f *)

Method RemoveEntry (n: Name);
(* pre n # "" *)
(* post ThisFile(n) = NIL *)

Method RegisterObserver (o: Observer);
(* post notified = o
   -- if notified # NIL, the method o.Notify will be called on AddEntry *)
```

The method `AddEntry` associates a file with a name in the directory. The method `ThisFile` returns the file corresponding to a given name. The method `RemoveEntry` removes a name from the directory. The method `RegisterObserver` allows to associate one interface of type `Observer` with a static variable `notified`. The contract of `RegisterObserver` states informally that this will result in calls to the `Notify` method of the observer, but is rather vague on the exact conditions under which this will be done. Interestingly, the contract of `AddEntry` is silent regarding notifications.

Clemens Szyperski acknowledges that there are several points left unclear in the contracts he provides ([Szyperski98], p. 52). Answers to "all questions about the what, when and where" are given in implementations (p. 53). We are mostly concerned here with the AddEntry method:

```
Method AddEntry (n: Name, f:Files.File);
   VAR e: EntryList; u: Notified;
BEGIN
   ASSERT(n # ""); ASSERT(f # NIL);
   e := entries;  (*search for entry under name n*)
   WHILE (e # NIL) & (e.name # n) DO e := e.next END;
   IF e = NIL THEN  (*not found: preprend new entry*);
      NEW(e); e.name := Strings.NewFromString(n); (*create new entry*)
      e.next := entries; entries := e  (*prepend new entry to list*)
   END;
e.file := f; (*fill in file, replaces old binding if entry already existed*)
u := notified;  (*u is the interface to be notified*)
IF u # NIL THEN u.notify(n) END;
END AddEntry;
```

We observe that AddEntry invokes Notify, an external operation, after achieving its contractual post-condition and prior to enabling its return action to occur. However, neither the Notify method nor even its contract are known to the directory — the only sure thing about Notify is that it will eventually return. For reasons of providing a simple example of contract violation, Szyperski proposes a Notify method analog to the following:

```
Method Notify (n: Directory.Name);
BEGIN
   IF n = "Untitled" THEN
      (*oops -- this name is not allowed*);
      myDirectory.RemoveEntry(n) (*gotcha, problem is fixed*)
   END;
END Notify;
```

Note that this method may invoke RemoveEntry as a call-back operation on the directory[1]. It has thus an opportunity to cancel AddEntry's postcondition just before that method returns. It appears that a contract violation occurs whenever a client of the directory attempts to register a file with the name "untitled". Szyperski investigates several strategies for solving this type of problem, but finds none that is totally satisfactory.

3.1 The implementation is considered correct

In this section, we assume that the method AddEntry is correct. Its current specification (the AddEntry operation) is an incompatible or wrong abstraction, which we must correct.

Instead of talking of a contract violation, we will talk of an *incompatibility* between the specification and the implementation. Indeed, it is always possible to fix a problem of contract violation by making the contract less binding, e.g., by stating no postcondition for AddEntry. But this would make operation contracts even further remote from a usable abstraction. To the contrary, our goal is to produce a complete specification that answers "all questions about the what, when and where," so that users of the directory will not need to look at its implementation. Since we do not know the contexts in which the directory component will be used, we will assume no knowledge of the behaviour of the Notify operation to be invoked. As Clemens Szyperski, we assume here a non-concurrent environment[2].

Quite clearly, it is impossible to fully specify AddEntry with solely a pair of pre- and postconditions. The reason is that AddEntry is not an atomic action as it potentially includes a call to an external Notify operation. It follows that RemoveEntry is potentially a sub-action of AddEntry, and its post-conditions may add to (and take precedence over) the normal postconditions of AddEntry. As she ignores whether call-backs will occur or not, a specifier of the directory is in no position to state the final postconditions of AddEntry.

In principle, the users of the directory are in a much better position, since they provide the observer object and they supposedly know the behaviour of its Notify operation. But these users must be given either a valid specification of the directory or its implementation. In the latter case, they are left to derive the specification from the implementation, if they really want to benefit from an abstraction. Chances are that they will do this exercise only mentally and partially, and incorrectly on their first attempt. Objects and components provide the potential for abstraction, but correct abstractions are not

1. Think of what would happen if the Notify method were to systematically invoke AddEntry instead of RemoveEntry — an infinite loop would occur. For keeping our example as simple as possible, we ignore this problem in this paper. Clemens Szyperski provides a solution in his book.
2. Clemens Szyperski attempts not to introduce concurrency unless he has to. He tends to make the implicit or explicit assumption of an environment with a single thread of execution (he seems to equate logical threads of execution with the threads provided by operating systems).

obtained automatically.

3.1.1 Techniques for producing a compatible specification

There are numerous ways to produce a compatible specification for the directory implementation. Most specification languages are capable of expressing it.

One simple example is the Fusion specification technique, which works according to the following principles. The system (in our case, an object or a component) waits on *input events*[1], which it will handle one at a time. This input event is associated with an atomic action, which is specified in terms of pre- and postconditions on the state of the system (thus, an input event, its parameters, and the current state of the system determine one state transition on the system). Following that atomic action, a number of *output events* may be emitted by the system ([Coleman94], sections 2-5.1 and C-3.2).

The key point to notice is that Fusion associates pre- and postconditions to input events (for defining an atomic action), and not to operations. This is illustrated in the following Fusion schema (or atomic action specification) for the `AddEntry` input event:

```
Input event: invocation of AddEntry -- version 1
Description: Adds an entry in the directory.
             Possibly notifies an external object of the addition.

Reads:       notified: Notified, supplied n: Directory.Name,
             supplied f: Files.File
Changes:     entries: Entry
Sends:       "invoker":{return of AddEntry}, notified:{invocation of
Notify}
Assumes:     n # ""
Result:      n was associated to f in the directory -- ThisFile(n) = f
             If notified # NIL, then notified was sent Notify(n).
             Otherwise, return of AddEntry was sent.
```

In the above schema, the `Assumes` clause corresponds to the precondition for the atomic action, while the `Result` clause corresponds to the postcondition. The `Sends` clause lists the potential output events of the atomic action. The `AddEntry` return action is to be sent to `"invoker"` — by this special name, we mean whichever object (or *agent* in Fusion terminology) is invoked by the `AddEntry` operation. This convention is an ad hoc extension to Fusion, which does not handle well operations in the usual OO sense. The most important clause is the `Result` clause. It especially states that either the `AddEntry` return action is performed or a notification is sent, but not both.

In any case, the `AddEntry` operation must be terminated by a return action. When a notification is sent, this obligation is fulfilled by the atomic action associated to the `Notify` return action:

```
Input event: return of Notify
Description: Terminates the original AddEntry operation that was invoked.
Reads:
Changes:
Sends:       "original invoker of AddEntry": {return of AddEntry}
Assumes:
Result:      return of AddEntry was sent.
```

At this point, some readers may wonder why we did not write the following `AddEntry` schema:

---

1. Unfortunately, Fusion refers to input events and their associated atomic action as *system operations*, or *operations* for short. This meaning is not that of operation in the UML or RM-ODP sense, for reasons that our example will make clear. To avoid confusion, we avoid using the term operation in the sense of Fusion.

```
Input event: invocation of AddEntry -- version 2
... -- all other clauses as in version 1
Result:      n was associated to f in the directory -- ThisFile(n) = f
             If notified # NIL, then notified was sent Notify(n).
             Return of AddEntry was sent.
```

The above `Result` clause would make the specification incompatible with the implementation, for two related reasons. Firstly, it contradicts our assumption of a non-concurrent environment, which implies that the environment can accept at most one output event at any time. Secondly, the directory specification would not guarantee to the notified entity an opportunity to call `RemoveEntry` before chances made by `AddEntry` are reported to its invoker. This consideration alone makes the specification incompatible with the implementation, which is correct by hypothesis.

To summarize, the Fusion specification technique with minor ad-hoc extensions is capable of expressing a compatible abstraction of the directory implementation. The key aspect of this capability lies in the fact that Fusion specifies atomic actions in terms of pre- and postconditions, and not operations in their UML or ODP sense. One of the drawbacks of Fusion is that the specifications produced are not structured in terms of operations. Another drawback is that atomic actions are always attached to input events (and not, for example, to internal events) — this is a limitation of Fusion, not a general rule about atomic actions.

3.1.2 Fusion vs. the RM-ODP

The Fusion specification technique may be seen as a simple but limited instantiation of the RM-ODP *information language*. This ODP language is intended for writing the specifications of systems, subsystems or smaller components, without revealing their implementation. It features *information objects* (instead of data) for expressing the states of components. It features also a notion of dynamic schemas for specifying the allowable state changes of the component, or in other words, the allowable joint state changes of one or more of its information objects (one atomic action may produce state changes in several information objects).

The state of a Fusion system is described in terms of analysis objects — these analysis objects correspond to the information objects of ODP. A Fusion input event schema (corresponding to an ODP dynamic schema) leads to one system state transition — this state transition may consist in the change of the states of several analysis objects (and possibly in the creation or deletion of other analysis objects). Therefore, Fusion allows an atomic action to simultaneously change the states of an arbitrary number of objects (this is true a the analysis abstraction level; it is no longer true at the abstraction levels of design and implementation).

An essential difference between Fusion and the RM-ODP is that Fusion insists that analysis objects map one to one onto design objects. This may be sound software engineering advice, but as an implementation constraint, it is unwarranted. The RM-ODP has no analog requirement or recipe (for example, there is no requirement that simple mappings exist between information objects and computational or engineering objects). Unlike Fusion, the RM-ODP fully acknowledges the fact that an object-oriented specification may be implemented without recourse to OO technology.

3.2 The initial specification is considered correct

In this section, we consider `AddEntry`'s original contract to be a partial but correct specification: the stated contractual postconditions hold in the specification. The `AddEntry` method appears thus to be incorrect. We must change this implementation so as to eliminate the potential for a contract violation.

But firstly, we must transform `AddEntry`'s original contract into a complete operation specification. We must indicate that a `Notify` operation might be invoked, and when it is to be invoked. As it turns out, we can simply use the 2nd version of the AddEntry schema, the very same schema that we rejected in the previous section:

```
Input event: invocation of AddEntry -- version 2
... -- all other clauses as in version 1
Assumes:     n # ""
Result:      n was associated to f in the directory -- ThisFile(n) = f
             If notified # NIL, then notified was sent Notify(n).
             Return of AddEntry was sent.
```

The reader should compare this schema with the contract and with the method given in Section 3. Unlike the contract, the specification makes an explicit mention of the Notify output event. Unlike the method, the specification has the `Notify` invocation and the `AddEntry` return action occur in parallel. Even if true parallelism semantics are excluded in favour of interleaving (i.e., even if the notification invocation may in fact precede the return action), the `AddEntry` return action is performed before any input event is accepted by the directory. Thus, the contract holds when the operation returns, as was required.

Concurrency is thus an intrinsic property of the specification: the directory may still be internally sequential, but parallel activities occur in its environment. Concurrency simply cannot be avoided if the contractual postcondition is to hold when `AddEntry` returns. Either the notification occurs in parallel with the `AddEntry` return action, or it occurs later. In both cases, the `Notify` operation exists as a separate activity from the `AddEntry` operation.

But now that concurrency has to be assumed by the directory clients (as is the rule in distributed systems), the implementation may no longer be considered to violate the original contract! Indeed, the behaviour as implemented (`RemoveEntry` occurs as a part of the `AddEntry` operation) is compatible with the behaviour as specified (`RemoveEntry` occurs after the completion of `AddEntry`). In the implementation scenario, the notified object is quite sure to remove the new entry before any other object uses it. In the specification scenario, it is not so sure, but the possibility nevertheless exists[1]. In both cases, the directory clients have to take into account that other clients can undo their operations on the directory.

If this is not acceptable, we have no choice but to further change the specification (and the implementation) of the directory. For example, it may be decided that only the creator of an entry is allowed to remove this entry, or a very privileged user. Or untrusted objects may not be given access to the update operations of the directory. Or yet, concurrency control may be introduced. As we have seen, the activity induced by the notification is to be considered a separate activity from that of the AddEntry operation — it would thus be denied access to the `RemoveEntry` operation. What we have shown by producing a correct specification is that, if there is a design problem with the directory, it is at the level of its specification rather than at the level of its implementation.

Understanding how to make good specifications, or how to make valid abstractions of existing implementations, is an important skill for designing object-based systems. Writing a correct specification does not transform a bad design into a good one, but it allows one to see the issues, and to understand which guarantees are really offered by an object, under what assumptions. As we have seen, the problems denounced by Clemens Szyperski have more to do with wrong specifications and poor design than with poor implementations. When the problems are understood, solutions can be found and discussed, often even before tackling an implementation.

1. The implementation presented here is correct as long as the activities of the directory client and of the notified object are independent from each other. In the implementation scenario, a deadlock may occur if the notified object attempts to synchronize with the directory client, after the `AddEntry` operation has returned. There is only one thread of execution in the implementation, for two logical threads in the specification.

4 Operation contract invariants vs. specification invariants

Clemens Szyperski touches on another problem in his book ([Szyperski98], chapter 5). He explains that call-backs and concurrent method executions can "hit" unspecified states, yielding incoherent results. He talks of the temporary "violation of an invariant."

The examples given by Clemens Szyperski are too extensive and too subtle to be discussed in this paper. However, Szyperski's readers should bear in mind that he thinks of invariants as viewed from Bertrand Meyer's contract theory [Meyer92], i.e., as general clauses which apply to the contractual pre- and postconditions of all operations. Such *operation contract invariants* say nothing about what happens in an operation between its invocation and its return. It is thus OK for an implementation to violate these invariants as long as they are re-established when an operation returns. As Clemens Szyperski demonstrates, operation contract invariants are not adequate to describe an abstraction of an object.

Specification invariants have a different meaning in specifications: they are predicates characterizing all the states of the specification. In other words, specification invariants are never violated — otherwise, the specification is inconsistent, i.e. worthless. Note that this statement applies to the specification, i.e., to the abstract description of the object behaviour, and not to the implementation. To know whether an implementation satisfies its specification invariants, one must determine whether the implementation is compatible with the specification. It is indeed possible for an implementation to be compatible with an implementation, and yet to appear to violate its invariants, even when operations return. But this does not matter to users of the implementation — what matters to them is that, as external observers, they are never in a position to detect any invariant violations, whether temporary or not. If Clemens Szyperski had defined specification invariants rather than operation contract invariants, he would not have been able to talk simultaneously of correct implementations and temporary violations of invariants.

The RM-ODP information language puts an heavy emphasis on (specification) invariants. Dynamic schemas are complemented by *invariant schemas*, which are sets of predicates that constrain the possible states and state changes of information objects. In other words, invariant schemas are general clauses which apply to the postconditions of all atomic actions (not operations), and to the initial state of an object. By definition, these predicates are therefore true for all the states of the specification. Again, this is typically not the case in the implementation, because an implementation features more actions and more states than its specification.

5 Specification of interfaces

The behaviour of an interface is not exactly that of a component, if that component has multiple interfaces rather than just one. This is usually the case, even for objects, since we consider both client and server interfaces.

The RM-ODP defines an interface in this way:

> "An abstraction of the behaviour of an object that consists of a subset of the interactions of that object together with a set of constraints on when they may occur. Each interaction of an object belongs to a unique interface. Thus the interfaces of an object form a partition of the interactions of that object.
>
> NOTE: An interface constitutes the part of an object behaviour that is obtained by considering only the interactions of that interface and by hiding all other interactions. Hiding interactions of other interfaces will generally introduce non-determinism as far as the interface being considered is concerned." [X.902].

The definition speaks of interaction instances, not interaction types — a same operation may be defined in several interfaces. The note attached to the definition gives a procedure for deriving an interface from an object.

This subtle topic deserves to be illustrated in an example. We go back to the AddEntry operation of the directory in a non-concurrent environment. We assume that all the interactions with the observer (i.e.,

the `RegisterObserver` and the `Notify` operations) are excluded from the client interface. Conversely, we assume that the observer, if any, has no access to the `AddEntry` operation. With these assumptions, the Fusion schema pertaining to AddEntry is then:

```
Input event: invocation of AddEntry -- version 3 (client interface)
Description: Attempts to add an entry in the directory.

Reads:       notified: Notified, supplied n: Directory.Name,
             supplied f: Files.File
Changes:     entries: Entry
Sends:       "invoker":{return of AddEntry}
Assumes:     n # ""
Result:      n was associated to f in the directory -- ThisFile(n) = f
             However, some entries, including n, may have been removed!
             Return of AddEntry was sent.
```

The atomic action always enables its corresponding return action: it corresponds to the operation. However, the operation specification has become non-deterministic. We are in no position of stating the effective post-conditions of the operation, as we cannot mention the potential interactions with an observer[1].

To make a further twist in this discussion, we now specify the same interface, but assuming this time that an observer is always associated to the directory, and that this observer systematically removes entries with the name "`Untitled`". With these assumptions, the behaviour of the AddEntry operation is now fully deterministic:

```
Input event: invocation of AddEntry -- version 4 (client interface)
... -- all other clauses as in version 1
Result:      If n # "Untitled", then n was associated to f in the directory
             Otherwise, n is no longer an entry in the directory.
             Return of AddEntry was sent.
```

The non-determinism of the interface has gone. The reason is that the interface has become an interface on a composite (or aggregate) object, that of the directory and its observer. There is therefore no longer any variable part in the specification.

In general, it is relatively easy to determine the behaviour (the specification) of an interface from the behaviour of its object. The converse task is much more difficult: it can be impossible to determine the behaviour of an object, even when the behaviours of all its interfaces are known.

According to the above RM-ODP definition, if an object has exactly one interface, then this object and its interface are one and the same thing (look at the procedure for deriving an interface: if there is only one object, there are no interactions to hide). The converse is true as well: an object and its interface are identical when that object has exactly one interface. But importantly, the operations invoked by an object count as interactions, and thus must belong to the unique interface (this is unlike the usual practice in object modelling). This observation confirms our claim that specifying the operations supported by an object is in general insufficient for fully specifying the behaviour of that object.

6 Summary and Conclusion

In this paper, we gave a precise meaning to notion of abstraction as it is embodied by an object implementation. An object's abstraction is the specification of that object — abstraction does not appear automatically, it must be carefully constructed. We also showed more precisely what should be under-

1. Some formal description notations, such as LOTOS, are capable to express this kind of non-determinism in a more clear and fully precise way.

stood by a specification, and we gave examples. In particular, contrarily to many people' beliefs, the set of operation contracts of an object do not necessarily provide a specification for this object. We also showed that an interface and an object are different concepts (often confused); in particular, their specifications can be different.

Along the theoretical considerations, we made the case that knowing how to make good specifications, or how to make valid abstractions of existing implementations, is an important skill for designing object-based systems. Indeed, specifications are simpler than implementations. When specifications are valid, they allow one to better grasp the problems of a component or of a system design. And when the problems are understood, solutions can be found and discussed, often even before tackling an implementation.

7 Bibliography

[Coleman94] D. Coleman, Patrick Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*. Englewood Cliffs: Prentice Hall, 1994.

[Herbert93] A. J. Herbert, "Distributing Objects," Architecture Projects Management Ltd., Cambridge (UK), ANSA Technical Report TR.18.01, Feb 18, 1993.

[Meyer92] B. Meyer, "Applying "Design by Contract"," *Computer (IEEE)*, vol. 25, pp. 40-51, 1992.

[Snyder93] A. Snyder, "The Essence of Objects: Concepts and Terms," *IEEE Software*, pp. 31--42, Jan. 1993.

[Szyperski98] C. Szyperski, *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[UML99] OMG, "OMG UML Specification v. 1.3," 1999.

[X.901] ISO/IEC and ITU-T, "Open Distributed Processing - Basic Reference Model - Part 1: Overview and Guide to Use," Standard 10746-1, Recommendation X.901. 1996.

[X.902] ISO/IEC and ITU-T, "Open Distributed Processing - Basic Reference Model - Part 2: Foundations," Standard 10746-2, Recommendation X.902. 1995.

[X.903] ISO/IEC and ITU-T, "Open Distributed Processing - Basic Reference Model - Part 3: Architecture," Standard 10746-3, Recommendation X.903. 1995.