

## 0.1 Specifying behavioral compositions

The notion of *behavioral types* may be regarded as the formal underpinning of the notion of *contracts* specifying the interaction between a client and server (object); cf. [Meyer93]. Due to the limited power of the (boolean) assertion language, contracts as supported by Eiffel are more limited in what may be specified than (a general notion of) behavioral types. However, some of the limitations are due, not to limitations on the assertion language, but to the local nature of specifying object behavior by means of contracts. See also [Meyer93].

To conclude this chapter, we will look at an example illustrating the need to specify global invariants. Further we will briefly look at alternative formalisms for specifying the behavior of collections of objects, and in particular we will explore the interpretation of *contracts* as behavioral compositions.

**Global invariants** Invariants specify the constraints on the state of a system that must be met for the system to be consistent. Clearly, as elementary logic teaches us, an inconsistent system is totally unreliable.

Some inconsistencies cannot be detected locally, within the scope of an object, since they may be caused by actions that do not involve the object directly. An example of a situation in which an externally caused inconsistent object state may occur is given in slide ?? (The example is taken from [Meyer93], but rephrased in C++.)

When creating an instance of *A*, the *forward* pointer to an instance of *B* is still empty. Hence, after creation, the invariant of the object is satisfied. Similarly when, after creating an instance of *B*, this instance is attached to the *forward* pointer, and as a consequence the object itself is attached to the *backward* pointer of the instance of *B*. After this, the invariant is still satisfied. However, when a second instance of *A* is created, for which the same instance of *B* is attached to the *forward* pointer, the invariant for this object will hold, but as a result the invariance for the first instance of *A* will become violated. See below.

```
A a1, a2; B b;
a1.attach(b);
a2.attach(b); // violates invariant a1
```

This violation cannot be detected by the object itself, since it is not involved in any activity. Of course, it is possible to check externally for the objects not directly involved whether their invariants are still satisfied. However, the cost of exhaustive checking will in general be prohibitive. Selective checking is feasible only when guided by an adequate specification of the possible interferences between object states.

**Specifying interaction** Elementary logic and set-theory provide a powerful vehicle for specifying the behavior of a system, including the interaction between its components. However, taking into account that many software developers prefer a more operational mode of thinking when dealing with the intricacies of complex interactions, we will briefly look at formalisms that allow a more explicit

**Problem – *dynamic aliasing***

0-1

```

class A {
public:
  A() { forward = 0; }
  attach(B* b) { forward = b; b->attach(this); }
  bool invariant() {
    return !forward —— forward->backward == this;
  }
private:
  B* forward;
};

class B {
public:
  B() { backward = 0; }
  attach(A* a) { backward = a; }
  bool invariant() {
    return !backward —— backward->forward == this;
  }
private:
  A* backward;
};

```

Slide 0-1: Establishing global invariants

specification of the operational aspects of interaction and communication, yet support to some extent to reason about such specifications. See slide ??.

In [HHG90], a notion of *behavioral contracts* is introduced that allows for characterizing the behavior of compositions of objects. Behavioral contracts fit quite naturally in the object oriented paradigm, since they allow both refinement and (type) conformance declarations. See below. Somewhat unclear, yet, is what specification language the *behavioral contracts* formalism is intended to support. On the other hand, from an implementation perspective the interactions captured by behavioral contracts seem to be expressible also within the confines of a class system supporting generic classes and inheritance.

A similar criticism seems to be applicable to the formalism of (role) *scripts* as proposed in [Francez]. Role scripts allow the developer to specify the behavior of a system as a set of roles and the interaction between objects as subscribing to a role. In contrast to behavioral contracts, the script formalism may also be applied to describe the behavior of concurrently active objects. In particular, the script formalism allows for the specification of predefined initialization and termination policies and for the designation of a so-called *critical role set*, specifying the number and kind of participants minimally required for a successful computation.

Also directed towards the specification of concurrent systems is the *multi-*

<b>Contracts – <i>behavioral compositions</i></b> <ul style="list-style-type: none"> <li>• specification, refinement, conformance declarations</li> </ul> <b>Scripts – <i>cooperation by enrollment</i></b> <ul style="list-style-type: none"> <li>• roles, initialization/termination protocols, critical role set</li> </ul> <b>Multiparty interactions – <i>communication primitive</i></b> <ul style="list-style-type: none"> <li>• frozen state, fault-tolerance, weakening synchrony</li> </ul> <b>Joint action systems – <i>action-oriented</i></b> <ul style="list-style-type: none"> <li>• state charts, refinement, superposition</li> </ul>	<div>interaction</div> 0-2
--	----------------------------

Slide 0-2: Specifying interactions

*party interactions* formalism proposed in [Evangelist], which is centered around a (synchronous) communication primitive allowing multiple objects to interact simultaneously. The notion of *frozen state* (which may be understood as an invariance requirement that holds during the interaction) may be useful in particular for the specification of fault-tolerant systems. An interesting research issue in this respect is to what extent the assumption of synchrony may be weakened in favor of efficiency.

A rather different orientation towards specifying the interaction between collections of concurrently active objects is embodied by the *joint action systems* approach described in [Kurki]. Instead of relying on the direct communication between objects, *joint action systems* proceed from the assumption that there exists some global decision procedure that decides which actions (and interactions) are appropriate.

<b>Joint action systems</b> action <i>service()</i> by client c; server s is when c.requesting && s.free do <body>	0-3
---	-----

Slide 0-3: Specifying actions – example

An example of an *action* specification is given in slide ???. Whether the *service* is performed depends upon the state of both the client and the server object selected by the action manager. [Kurki] characterize their approach as *action-oriented* to stress the importance of specifying actions in an independent manner (as entities separate from classes and objects). An interesting feature of the *joint action systems* approach is that the behavior of individual objects is specified by means of *state charts*, a visual specification formalism based on [Harel87]. The

specification formalism adopted gives rise to interesting variants on the object-oriented repertoire, such as inheritance and refinement by superposition. From a pragmatic viewpoint, the assumption of a global manager seems to impose high demands on system resources. Yet, as a specification technique, the concept of *actions* may turn out to be surprisingly powerful.

In summary, this brief survey of specification formalisms demonstrates that there is a wide variety of potentially useful constructs that all bear some relevance to object-oriented modeling, and as such may enrich the repertoire of (object-oriented) system developers.

**Contracts as protocols of interaction** Contracts as supported by Eiffel and Annotated C++ are a very powerful means of characterizing the interaction between a server object and a client object. However, with software becoming increasingly complex, what we need is a mechanism to characterize the behavior of collections or compositions of objects as embodied in the notion of *behavioral contracts* as introduced in [HHG90].

A *contract* (in the extended sense) lists the objects that participate in the task and characterizes the dependencies and constraints imposed on their mutual interaction. For example, the contract *model-view*, shown below (in a slightly different notation than the original presentation in [HHG90]), introduces the object *model* and a collection of *view* objects. Also, it characterizes the minimal assumptions with respect to the functionality these objects must support and it gives an abstract characterization of the effect of each of the supported operations.

<pre> contract model-view; V <math>\dot{=}</math> {   subject : model supports [     state : V;     value( val : V ) <math>\mapsto</math> [state = val]; notify();     notify() <math>\mapsto</math> <math>\forall v \in \text{views} \bullet v.\text{update}()</math>;     attach( v : view ) <math>\mapsto v \in \text{views}</math>;     detach( v : view ) <math>\mapsto v \notin \text{views}</math>;   ]   views : set view <math>\dot{=}</math> where view supports [     update() <math>\mapsto</math> [view reflects state];     subject( m : model ) <math>\mapsto</math> subject = m;   ]   invariant:     <math>\forall v \in \text{views} \bullet [v \text{ reflects subject.state}]</math>   instantiation:     <math>\forall v \in \text{views} \bullet \text{subject.attach}(v) \ \&amp; \ v.\text{subject}(\text{subject});</math>     subject.notify(); }</pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block;"><i>MV(C)</i></div>	0-4
---	---	-----

Slide 0-4: The Model-View contract

To indicate the type of variables, the notation  $v : \text{type}$  is used expressing that

variable  $v$  is typed as *type*. The object *subject* of type *model* has an instance variable *state* of type  $V$  that represents (in an abstract fashion) the value of the *model* object. Methods are defined using the notation

• `method`  $\mapsto$  `action`

Actions may consist either of other method calls or conditions that are considered to be satisfied after calling the method. Quantification as for example in

•  $\forall v \in \text{views}$  • `v.update()`

is used to express that the method *update()* is to be called for all elements in *views*.

The *model-view* contract specifies in more formal terms the MV part of the MVC paradigm discussed in section ???. Recall, that the idea of a *model-view* pair is to distinguish between the actual information (which is contained in the *model* object) and the presentation of that information, which is taken care of by possibly multiple *view* objects.

The actual protocol of interaction between a *model* and its *view* objects is quite straightforward. Each *view* object may be considered as a handler that must minimally have a method to install a model and a method *update* which is invoked, as the result of the *model* object calling *notify*, whenever the information contained in the model changes. The effect of calling *notify()* is abstractly characterized as a universal quantification over the collection of *view* object. Calling *notify()* for *subject* results in calling *update()* for each *view*. The meaning of *update()* is abstractly represented as

• `update()`  $\mapsto$  [`view reflects state`];

which tells us that the *state* of the *subject* is adequately reflected by the *view* object.

The invariant clause of the *model-view* contract states that every change of the (state of the) *model* will be reflected by each *view*. The instantiation clause describes, in a rather operational way, how to initialize each object participating in the contract.

In order to instantiate such a contract, we need to define appropriate classes realizing the abstract entities participating in the contract, and further we need to define how these classes are related to their abstract counterparts in the contract by means of what we may call, following [HHG90], *conformance declarations*. Conformance declarations specify, in other words, how concrete classes embody an abstract role, in the same sense as in the realization of a partial type by means of inheritance.