A Reflective Approach for Describing Cooperation between Objects

E. Tramontana, R. de Lemos

TECHNICAL REPORT SERIES

No. 663
March, 1999
A Reflective Approach for Describing Cooperation between Objects

Emiliano Tramontana and Rogério de Lemos

Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU, UK
{emiliano.tramontana, r.delemos}@newcastle.ac.uk

Abstract

Conventional object—oriented model lacks the means to represent collaborative behaviour between objects. In this paper an approach is described for extending the representation of objects to express cooperations between them. The motivation for defining a cooperative object—oriented approach is to provide support for developing adaptive software systems: the degree of adaptability of a software system depends on the flexibility of objects changing their pattern of collaboration. The cooperative object—oriented approach is described in terms of a reflective model, in which the collaborative activity is captured by the metaobjects thus obtaining a clear separation between the objects and their respective cooperations. The role of the metaobject protocol is to provide the means for associating a set of metaobjects to an object and for reconfiguring dynamically the metaobjects. In this paper, we present the basic structure of a metaobject and define the services that the metaobject has to provide in order to implement a cooperation.

Keywords: collaboration, adaptive software, software architectures, reflection, reusability

1. Introduction

Comparing with a conventional object—oriented design, collaboration—based designs are able to provide a better support to achieve high—levels of design reuse /VanHilst 96, Smaragdakis 98b/. The basis for this argument is that high degree of reusability is difficult to achieve if reuse libraries are populated either with feature—driven variations of components, or with components which are highly complex (complexity is derived from the need to support generic interfaces for providing a wide range of services) /Biggerstaff 94/. In collaboration—based designs, software systems are represented as a composition of independently—definable collaborations: collaborations are known as a set of objects together with a set of activities that determine how objects interact, and that part of an object which prescribes the activity of the object, within a collaboration, is called the object’s role /Smaragdakis 98b/. In terms of software evolution, collaboration—based designs are able to achieve higher degrees of reusability because libraries would have fewer and lower complexity components, while the feature—driven services would be provided by combining components using collaborations. Moreover, reusability would not be restricted to components because collaborations could be equally reused.
However, the main concern of this paper is not with software evolution (or adaptability at design—time /Bosch 98, Lieberherr 96/), but instead with adaptability at run—time — the ability of a software system to adapt itself to changes that occur in its environment, while providing the required service defined as goals that have to be achieved by the system. The aim of this work is to establish a software architecture which is able to support dynamic configuration of a software system. The motivation for selecting a collaboration—based design is similar to that of software evolution: the degree of (run—time) adaptability of a software system depends on the flexibility of objects changing their pattern of collaboration. Instead of having a software system based on components which individually are able to provide a wide range of services, the approach being pursued relies on the ability of several components to reconfigure their collaborations, to provide the required service of the system. In such approach components remain unchanged, while the roles played by components may change according with the collaborations that have to be established between the components.

In collaboration—based designs, a role is a notion that is common to both objects and collaborations because they may be viewed as a collection of roles /VanHilst 96/. In terms of software evolution, the notion of a role is important since it provides the basis to support component changes: an object which takes part in a collaboration can be replaced by another object if the corresponding object is able to play the same role. However, when dealing with run—time adaptability objects and collaborations are more adequate as abstractions for representing alternative configurations of software components. For instances, if we consider an embedded system where the number of resources (in terms of software components) is limited, the only means to change the system behaviour (without changing its components) is to modify the pattern of collaboration between the components. However, there are some applications where the notion of collaboration is not sufficient to represent collaborative behaviour which aims to achieve a common purpose. In complex concurrent applications it is also necessary to capture the notion of coordination, to support, for example, error handling between multiple interacting objects /Randell 97, Xu 95/.

The software architecture for supporting run—time adaptability should include an architectural element which is able to represent the dynamic composition of software components. The modelling abstraction cooperative action (CO action) was introduced in order to coordinate collaborations between objects, which can either be cooperative or competitive /de Lemos 98/. In a cooperative object—oriented design, at any point in time, an object can be involved in more than one cooperation (defined by its participants and collaborative activity), and during run—time objects are able to reconfigure their interactions by selecting the appropriate cooperations depending on the required behaviour of the system. For the purpose of this work, it is assumed that a restricted form
of adaptability can be achieved by changing the collaborations activities associated with a cooperation but not its participants. In this paper, a reflective implementation for cooperative object-oriented designs is described in which cooperations are represented by associating metaboobjects with each of the object involved in a cooperation. An advantage of this approach is that a clear separation between objects and their respective cooperations is obtained, which is one of the advantages of collaboration-based designs.

The rest of the paper is organised as follows. In the next section, we briefly introduce the cooperative object-oriented style, which is used to specify collaboration-based systems. Section 3 describes a reflective model of a cooperative object-oriented system, in which objects coordinate their cooperations in a decentralised way. The feasibility of the whole approach is discussed in section 4, in terms of a partial implementation of the Production Cell benchmark case study. Finally in section 5 we present some concluding remarks.

2. Cooperative Object-Oriented Style

Systems are defined by their components and the relationships among their components, hence when modelling systems using the conventional object-oriented approach, objects alone are insufficient to describe the system behaviour. One of the motivations for introducing cooperations as another architectural abstraction is that, in conventional object-oriented models, objects alone are considered to be the basic entities of processing, while the interactions between objects are usually associated with low-level primitives of programming languages, like the remote procedure call of object-oriented programming. In a cooperative object-oriented approach the conventional object-oriented model is extended by explicitly representing collaborative activities between objects in terms of cooperations, which are expressed in terms of cooperative actions (CO actions) /de Lemos 98/. In this approach, while objects are employed to model system structure and component behaviour, the other modelling abstraction, CO actions, are used to model cooperations between objects by preserving one of the main characteristics of object-oriented modelling which is encapsulation.

The main feature that CO actions add to an object-oriented approach, is the ability of CO actions of extracting from the specification of an object the dependencies that are related with its collaborative activities, thus avoiding that a specification of a cooperation be scattered among objects. As a result, the specification of an object will only include the services to be provided by the object, independently of its collaborations. A major drawback in the conventional object-oriented models is that for objects to interact between themselves they have to know the identity of the other objects. This causes a major problem whenever one of the objects changes its identity because the specification of the remaining interacting objects has to be modified, in order to accommodate the change. In a software architecture generated by the cooperative object-oriented style, objects do not need to know which are the objects they interact and the cooperations they participate, these
interdependencies are captured instead by the cooperations in the software architectures. Hence, an important feature of cooperative object—oriented software architectures is that the impact of changing an object identity is restricted to the cooperations in which the object is involved, which is very similar to pipe—and—filter systems where filters are not aware of the other filters in the system when interacting with them /Shaw 96/.

CO actions are a variant of coordinated atomic actions (CA actions) which are design mechanisms for structuring complex concurrent activities and supporting error recovery between multiple interacting objects in an object—oriented system /Xu 95, Randell 97/. In the following, we describe the main features of the architectural elements of the cooperative object—oriented style and describe the basic rules for configuring architectural elements.

2.1. Architectural Elements and Configuration Rules

For the description of the architectural elements of the cooperative object—oriented style, we define objects as the basic components of the architecture, while cooperations are the basic connectors. The difference between these elements is that an object performs its computation locally, while it is the role of the cooperations to coordinate the distributed computation performed by the objects. For the description of systems, we define the configuration rules of the cooperative object—oriented style which regulate how objects and cooperations can be combined. In the following, we will focus on the static properties of the cooperative object—oriented style.

2.1.1. Objects

As in conventional object—oriented models, objects in the proposed approach, support the representation of both structural and behavioural aspects of a system. An object is described by a template with the following fields: a name, declaration of attributes in terms of constants and variables which are local to the object, a description of its structure in terms of a collection of components in composed of and the intra—relations between the object and its components, and finally, a description of the behaviour of the object.

The behaviour field includes the initial state of the object, and behavioural assumptions or consistency invariants associated with the object. The behavioural field also includes the specification of the complete space of the behaviour of the object, in terms of its normal, exceptional and failure behaviours. Normal and exceptional behaviours are related with the liveness properties of a system (“something good” eventually happens), while failure behaviours are related with the safety properties of a system (“something bad” does not happen). The specification of the exceptional behaviour comprises: the definition of the exceptional event, and the definition of its respective handler in terms of the handler’s start and finish events; the handler starts its execution whenever an exception is raised and finishes when a set of sufficient and necessary conditions are satisfied. (Failure behaviour
was introduced for the purpose of conducting the safety analysis of the specifications; exceptional and failure behaviours will not be considered in the rest of this paper.)

2.1.2. Cooperative Actions (CO Actions)

CO actions are employed in the specification of cooperative behaviour between objects. CO actions can either coordinate the activities to be performed by the objects, or execute some activity which is not associated with any particular object which takes part in the cooperation. A CO action is described by a template with the following fields: the CO action’s **name**, the names and types of the **participants** of the CO action, declaration of **attributes** in terms of **constants** and **variables** local to the CO action, and the specification of the collaborative **behaviour** of the objects participating in the CO action.

The **initial** state of a CO action represents its state when is activated, and is dissociated from the pre—conditions of the CO action: it either refers to the state of objects participating in the cooperation or the state of the variables local to the CO action. Associated with the description of **normal** behaviour, **pre—conditions** and **post—conditions** establish the respective conditions for a set of objects to start and finish a particular collaborative activity, and the **invariant** establishes the conditions which should hold while the collaborative activity is being performed. For the successful execution of a collaborative activity it is necessary that the pre and post—conditions of the normal behaviour are satisfied, and that the invariant associated with the collaborative activity is not violated during its execution. For the specification of exceptional behaviour, the invariant is replaced by a **handler**, which identifies the exception event, together with the start and finish events associated with the handler of the exception. Although the pre—conditions for normal and exceptional behaviours are the same, the post—conditions for the exceptional behaviour might be different, depending on the degraded outcomes of a CO action, once an exception has occurred. In the definition of a CO action, an exception can be associated with the invariant whenever this is violated, or with the post—conditions whenever one of the conditions is not satisfied.

A CO action provides the basis for dealing with both cooperative and competitive concurrency by integrating two complementary concepts: **conversations** /Randell 75/ and **transactions** /Gray 93/. Conversational support is used to control cooperative concurrency and to implement coordinated and disciplined error recovery, whilst transactional support maintains the consistency of shared resources in the presence of failures and concurrency among different collaborative activities competing for these resources /Randell 97, Xu 95/.

2.1.3. Configuration Rules

The only architectural elements of the cooperative object—oriented style are objects and CO actions, which excludes any other class of connectors, such as associations, which are usually part of conventional object—oriented approaches. In a cooperative
object-oriented system, each object and CO action has a unique name. Objects can participate in more than one cooperation, and at least two objects have to be associated with a CO action, thus creating the context in which objects cooperate.

A cooperative object-oriented diagram represents, in a general way, the static structure of a system in terms of objects and the cooperations between these objects. The cooperative object-oriented diagram also provides a hierarchical representation of the system by employing the notions of aggregation and composition. The internal structure of an object can be decomposed into other objects and its behaviour refined by identifying new cooperations between the decomposed objects. In the same way that an object can be structurally decomposed, a CO action can be functionally decomposed into other CO actions. These CO actions can either replace a higher level cooperation when there is no state associated with it, or can become nested CO actions of an enclosed, higher level cooperation. The behavioural refinement of a CO action will lead to its re-definition in terms of the decomposed object participants.

2.2. Related Work on Collaboration-Based Design

The notions behind collaboration-based designs, which aim to explicitly specify the interrelations between objects in a conventional object-oriented model, are not new. Although differences might exist between the existing approaches for representing these abstractions, in general terms collaborations are known as a set of objects together with a set of activities that determine how objects interact. In a collaborative-based design the aim is to compose independently-definable collaborations when defining software systems. Some of the collaboration-based approaches have adopted the view that collaborations should be used to model message passing and state changes, by focusing on the representation of roles that an object has while participating in a collaboration /Smaragdakis 98a, VanHilst 96/ (although in /Smaragdakis 98b/ the definition of collaboration-components is based on the roles of a collaboration).

On the other hand, similar to the cooperative object-oriented style, there are those approaches which have adopted a broader view in which collaborations are modelling abstraction which are able to capture the properties of a collaborative activity to be performed by a group of objects /Helm 90, Kristensen 96, Kurki-Suonio 96/. Contracts were introduced in /Helm 90/ as abstractions to specify behavioural compositions and obligations on participating objects. A contract defines a set of communicating participants and their contractual obligations, pre-conditions which are required for participants to establish a contract, and an invariant which has to be maintained by these participants. Activities were introduced in /Kristensen 96/ as an abstraction mechanisms to model the interplay between objects, and it is claimed that activities are more powerful than the Mediator pattern /Gamma 94/ because they can be used as building mechanisms in creating modelling abstractions which can then represent concepts, such as roles and relations. The
theoretical foundation of the incremental derivation of collective behaviour of operational models of objects was defined in /Kurki–Suonio 96/ in the context of an action–oriented language.

2.3. Cooperations and Adaptability

It has been demonstrated how collaboration–based designs can be effective in improving adaptability during design–time /Kristensen 96, Smaragdakis 98a, VanHilst 96/. In this section we show how the same design principles can be employed to support adaptability during run–time. In the context of this paper, the provision of adaptability is achieved by changing the collaborative behaviour of a set of components that participating in a cooperation, without changing the actual participants. In terms of a cooperative object–oriented design this can be depicted by representing several instances of the same cooperation. The diagram of figure 1 represents a cooperative object–oriented design of a system in which objects instantiated from classes Cla1 and Cla2 are able to modify their combined behaviour between CoopA1 and CoopA2, according to changes that might occur in their environment.

Figure 1. A cooperative object–oriented design.

In the next section, we describe how cooperative object–oriented designs can be implemented in terms of a reflective architecture, which is able to support run–time adaptability.

3. A Decentralised Implementation of a Cooperative Object–Oriented Design

The implementation of a CO action can either be centralised or decentralised. In a centralised implementation, CO actions are represented by a single object which exist at run–time to coordinate the execution of the objects participating in the cooperation. Most of the existing implementations of cooperations follow the centralised approach, for instance, design pattern Mediator /Gamma 94/, activities for modelling the interplay between objects /Kristensen 96/, or collaboration–components which encapsulates in a single parameterized class all the roles which define a collaboration /Smaragdakis 98b/. In a decentralised implementation, there are no single objects to coordinate the cooperation, instead each object participant of a cooperation contains all the collaborating activities
which define a CO action, in other words, the implementation of a CO action is scattered among participating objects.

3.1. Reflective Object—Oriented Systems

A reflective software system contains structures representing aspects of itself which enable the system to support actions on itself. In object—oriented programming, reflection is realized with metaobjects, which represent and control the objects behaviour [Maes 87].

A reflective object—oriented system is split into a baselevel which implements the objects, and one or more metalevels which implement the metaobjects. The metaobject associated with an object is able to change the behaviour of the object without changing its code: the method calls to the object are intercepted by the metaobject forcing the execution to pass from the baselevel to the metalevel. Once the execution is finished at the metaobject, the flow returns to the baseobject. For example, a metalevel can implement object persistency, transparently for the baselevel, by properly storing and retrieving objects when they are referenced [Chiba 95].

The interactions between baseobjects and metaobjects are governed by a metaobject protocol (MOP), which provides the means for the metaobject to control the behaviour of their associated objects, and for metaobjects to communicate with each other. A MOP also controls the dynamic association, and disassociation, of metaobjects with objects — when a metaobject is associated with an object, the object behaviour is modified by that metaobject during the time frame of the association.

3.2. Reflective Model of a Cooperation

A cooperation between a set of objects is represented by associating a metaobject with each object involved in the cooperation. An object will have as many metaobjects associated with it as many cooperations in which is involved. A cooperative object—oriented system can be reconfigured at run—time by dynamically associating metaobjects with objects according with the cooperation to be executed. A reflective implementation of the cooperative object—oriented design of figure 1 is shown in figure 2. In this diagram, for each of the objects that take part in a cooperation, we associate a metaobject. For example, there are three classes involved in cooperation Coop$R$, hence we associate a metaobject with each of the classes, and the combined behaviour of the activities of these metaobjects should implement the referred cooperation. The different collaborative behaviours which can be obtained from the cooperation of classes Cla$1$ and Cla$2$ are represented by two pairs of metaobjects associated with these objects. During run—time, one of these pairs is dynamically associated with their respective objects. In the reflective model of cooperations, the notion of roles from collaboration—based designs is captured by the metaobjects which are associated with an object.
For a metaobject to implement a cooperation for an object, it needs the following three metamethods: *pre()* and *post()* which check, respectively, the pre— and post—conditions associated with a cooperation, and *coll()* which executes the collaborative activities. When a method call to a baseobject is trapped, the metamethod *Reflect()* is invoked to control the sequence of steps to be executed by the cooperation, which includes the dynamic association of the metaobject to the object.

Depending on the properties associated with a cooperation the appropriate communications services have to be provided between the metaobjects that implement the cooperation. As previously mentioned, a CO action is able to represent both cooperative and competitive concurrency, hence the reflective implementation of a cooperation should include both conversational and transactional support, which should be provided by protocols controlling the interactions between metaobjects. For example, if for a particular application it is paramount to maintain consistency (even in the presence of failures) between the representations of the metaobjects that implement a cooperation, then it is necessary to implement a communication service that supports the ACID properties (atomicity, consistency, isolation and durability). Depending on the required support different metaclasses can be defined which can then be inherited according with the needs of the application being developed /Parrington 95/. However, in this paper, in order to simplify the presentation of the control flow, we assume a simpler way for coordinating access to shared objects in order to maintain consistency: the mutual exclusion mechanism is based on two additional metamethods (*request()* and *release()* which control the locks on the objects to maintain their representations consistent.

In the following we describe the flow of control that takes place when method *methodM1()* of class *Cla1* is invoked (assuming that class *Cla1* defines methods *methodM1()* and class *Cla2* defines *methodM2()*). The description will be based on diagram of figure 3, which shows the sequence of steps associated with the reflective implementation of a cooperation. Every call to *methodM1()* at the baselevel, is trapped and handled at the metalevel by the
method `Reflect()` of the associated metaobject `O1MetaCoopA1 [Ω]`. This metamethod is responsible for starting the execution of several steps associated with a cooperation.

![Diagram of cooperation](image)

**Figure 3.** A reflective implementation of a cooperation.

The first step is to ask for the permission to use object `:Cla1`, by calling the metamethod `request()`. When the permission is granted the flow returns to `Reflect()`, which requests permission to access object `:Cla2` by calling the method `request()` of the metaobject `O2MetaCoopA1 [Ω]`. Again, when the permission is granted the flow returns to `Reflect()` of `O1MetaCoopA1 [Ω]`.

The second step is to check whether the `pre`-conditions hold, which consists in verifying the states of the objects involved in the cooperation: (i) by calling the method `pre()` of `O1MetaCoopA1`, which accesses the baseobject `[Ω and Ω]` and, (ii) by calling the method `pre()` of `O2MetaCoopA1 [Ω]`, which also accesses its baseobject `[Ω and Ω]` and then returns a result to the calling metaobject `O1MetaCoopA1 [Ω]`.

If the `pre`-conditions are satisfied the third step is to invoke, concurrently, methods `coll()` of `O1MetaCoopA1` and `O2MetaCoopA1`, which might involve in invoking methods of their respective baseobjects `[Ω and Ω]`: `methodM1()` of `:Cla1` and `methodM2()` of `:Cla2`.

Once the execution of the activities is finalized, in the fourth step, the control returns back to `O1MetaCoopA1` which invokes the methods `post()` of `O1MetaCoopA1` and `O2MetaCoopA1`, for checking whether the post-conditions hold, in the same way as it was described for method `pre()`. The final step is to release the objects, calling `release()` of `O1MetaCoopA1` and `O2MetaCoopA1`. The execution then goes to the calling object `[Ω]`.

The reflective object-oriented language employed for implementing a reflective model of cooperations is Open C++ version 2.5. This version of OpenC++ makes available to a metaobject the following functions: `Reflect()` receives the control on a method trap, `Reify()` executes the trapped baseobject method, and `LookupMember()` accesses baseobject’s member /Chiba 98/. Unfortunately OpenC++ provides only some of the features that our approach requires from a MOP. Using OpenC++ it is not possible to associate more than one metaobject to an object, and to dynamically associate metaobjects to objects. In order to solve those implementation issues we are investigating other reflective object-oriented
languages, among these ABCL/R2 /Yonezawa 90/ which is a language based on hybrid group architecture and targeted for real time systems, and Dalang /Welch 98/ which is a reflective language based on Java.

3.3. Cooperative Metaobjects as Generic Metaobject Classes

An OpenC++ generic metaobject class was defined for supporting the implementation of cooperations as metaobjects. The methods \texttt{pre()}, \texttt{coll()} and \texttt{post()} of the class are defined as virtual; \texttt{pre()} and \texttt{post()} are logical functions which take as parameters the states of the baseobjects. An array of pointers to metaobjects is used to specify the other metaobjects which are part of a cooperation. References to the other metaobjects and the implementation of the methods have to be specified by the programmer when the generic metaobject class is inherited. The following shows metaobject class \texttt{Coop} implemented using OpenC++.

```cpp
class Coop : public Metaobj {
public:
    virtual int pre() {}
    virtual int coll() {}
    virtual int post() {}
    virtual void initialize() {}
    void Reflect( ... ) { ... 
      // acquire objects
      request( my_pid );
      for ( int i=0; mobj[i] != NULL; i++ )
        mobj[i]->request( my_pid );
      if ( pre() == OK ) {     // invoke pre of each object
        for ( int i=0; mobj[i] != NULL; i++ )
          if ( mobj[i]->pre() != OK ) {
            start = false;
            break;
          }
      }
      if ( start ) {     // invoke coll of each object
        for ( int i=0; mobj[i] != NULL; i++ ) {
          if ( fork() ) mobj[i]->coll();
        }
      }
      coll();
      // invoke post of each object (analogously as pre)
      ... 
    }
    // release objects (analogously as request)
    ... 
    void request( int pid ) { ... }
    void release() { ... }
};
```
In addition to the above metaclass which is able to maintain consistency between the
metaobjects that implement a cooperation, other metaclasses could equally be defined for
supporting other kinds of service, for instance, error recovery based on exception handling.

3.4. Related Work

The implementation of roles in the approach proposed by VanHilst and Notkin /VanHilst
96/ uses standard C++ class templates parameterized by each of the participants in a
collaboration. Roles are defined in terms of classes by instantiating the templates and
binding its parameters to specific classes representing the participants. Instantiation of class
templates allows to define new collaborations without modifying the definition of roles.
However, approach by VanHilst and Notkin focuses on the implementation of roles rather
than collaborations.

Smaragdakis and Batory /Smaragdakis 98b/ extend the concepts of roles and collaborations
of /VanHilst 96/ by defining collaborations as primitive abstractions for building software
systems. Collaborations are implemented through inheritance of nested classes using
standard C++ class templates parameterized by the roles it uses. Collaborations can be
composed by instantiating one collaboration—component with another as its parameter.
Collaborations are statically defined by the templates, so they cannot change during
run—time.

Kristensen and May /Kristensen 96/ use the concept of activity to describe collaborations
between objects. A collaboration is described by specifying a set of roles and their
corespondent activity. An activity is described by a relation—class, which is a nonstandard
C++ class able to relate the participants of an activity. Roles are described using
nonstandard C++ classes called role—classes. An object can have role—objects allocated
and deallocated from role—classes during run—time, therefore roles can be used to support
dynamic changes in the functionality of participants.

/VanHilst 96/ and /Smaragdakis 98b/ do not provide any support for changing collaboration
between objects during run—time, while /Kristensen 96/ provides some support which is
limited to changing the roles and not the activities which define a collaboration. Our
approach, instead, defines several collaborations for a set of objects and selects one
collaboration, during run time, by dynamically associating metaobjects to objects. To
facilitate the implementation of cooperations, a generic metaclass was defined which
incorporates the basic services to be inherited by a metaobject which has a role to play in
a cooperation.

4. Case Study – Production Cell

The production cell is a benchmark case study /Lewerentz 95/ useful for describing the
issues discussed in this paper. The production cell is composed of 6 devices used to process
metal plates. In the production process, the metal plates are conveyed to an elevating rotary table by a feed belt. We will focus on these two devices (FeedBelt and Table) and on their respective cooperation (LoadTable) which moves a plate from the feed belt into the table. The cooperation LoadTable is implemented in terms of the metaobjects FBLoadTable and TLoadTable which are associated with FeedBelt and Table, respectively.

The object FeedBelt provides the method BeltOn() which is responsible for moving the feed belt. When the method BeltOn() is invoked, the call is trapped to metaobject FBLoadTable, which forces the execution of method Reflect(), responsible for calling the methods request() of FBLoadTable and TLoadTable, asking permission for accessing objects :FeedBelt and :Table, respectively. When permission is granted, the pre-conditions are checked by metamethods pre() of FBLoadTable and TLoadTable. If pre-conditions are satisfied the collaborative activities are able to start which involves executing metamethod coll() of TLoadTable and executing metamethod coll() of FBLoadTable for moving a plate from FeedBelt to Table by using method BeltOn(). The execution then goes to the metamethods post() of FBLoadTable and TLoadTable for checking whether the post-conditions hold. The objects are then released by using release(), and finally the execution goes to the calling object. Figure 4 represents the previous description.

![Figure 4. Reflective model of cooperation LoadTable.](image)

Inheriting the generic metaclass Coop, the code for the metaobject class FBLoadTable which implements the cooperation LoadTable for FeedBelt is the following.

```cpp
class FBLoadTable : public Coop {
public:

    // pre, coll and post are defined here
    int pre() {
        Member m1, m2, m3;
        LookupMember("flagbeltOn", m1);
        LookupMember("plateOnBeg", m2);
        LookupMember("plateOnEnd", m3);
        if ( !m1 && m2 && !m3 ) return 1;
        else return 0;
    }

    int coll() {
        Member m;
        LookupMember("beltOn", m);
```

13
m();
}
int post() {
    Member m1, m2, m3;
    LookupMember("flagbeltOn", m1);
    LookupMember("plateOnBeg", m2);
    LookupMember("plateOnEnd", m3);
    if ( !m1 && !m2 && m3 ) return 1;
    else return 0;
}
void initialize() { ... }
);

Inheriting the generic metaclass Coop, the code for the metaobject class TLoadTable which implements the cooperation LoadTable for Table is the following.

class TLoadTable : public Coop {
public:
    int pre() {
        Member m1, m2;
        LookupMember("plateOn", m1);
        LookupMember("bottom", m2);
        LookupMember("angle", m3);
        if ( !m1 && m2 && (m3==0) ) return 1;
        else return 0;
    }
    int coll() { }
    int post() {
        Member m1, m2, m3;
        LookupMember("plateOn", m1);
        LookupMember("bottom", m2);
        LookupMember("angle", m3);
        if ( m1 && m2 && (m3==0) ) return 1;
        else return 0;
    }
    void initialize() { ... }
};

If in the above example different behaviours for loading the table were required depending on the type of plate, then alternative cooperations could have been defined involving the same participants but with different collaborative behaviours. During run-time, a new cooperation would be established depending of the plate to be moved.

5. Conclusions

This paper has described an approach for the implementation of collaborative–based designs in terms of a reflective architecture. While the base–level objects are associated with software components, their respective metaobjects capture the roles of an object which establish the cooperations in which an object can be involved. In such implementation run–time adaptability is achieved by dynamically binding objects to metaobjects. In terms of software evolution, when using the cooperative object–oriented style the libraries of software components contain a reduced number of components of low complexity because
all the particular features of service to be provided by the system will be obtained from the
cooperations between the components.

A key element in any adaptive software system is the mechanism for selecting, during
run-time, a particular software configuration according with changes that occur in the
environment of the software. During design-time, decisions are taken by the software
developer when defining a software architecture from a set of alternative components and
collaborations. During run-time, the process decision making has to be more dependable,
due to the lack of human redundancies, although it is usually more complex due to the
different levels of decision making that are necessary: depending on the state of the
environment, and the goals and obstacles associated with an application, the appropriate
collaborations have to be defined, then the roles needed to implement a collaboration have
to be established, and finally the components which are responsible for playing the
established roles have to be identified. Research is being performed in this area.

Acknowledgements

The authors would like to thank Alexander Romanovsky for the discussion on the topic, and
would like to acknowledge the financial support of EPSRC/UK ADAPT and SafeGames
projects.

References

/Biggerstaff 94/ T. J. Biggerstaff. “The Library Scaling Problem and the Limits of Concrete
Component Reuse”. Proceedings of the 3rd International Conference on Software Reuse

/Bosch 98/ J. Bosch. “Superimposition: A Component Adaptation Technique”.


/Chiba 98/ S. Chiba. Open C++ 2.5 Reference Manual. Institute of Information Science and

/de Lemos 98/ R. de Lemos, and A. Romanovsky. “Coordinated Atomic Actions in
Modelling Object Cooperation”. Proceedings of the 1st IEEE International Symposium on
152–161.

/Gamma 94/ E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements


