COMPUTING SCIENCE

A Cooperative Object—Oriented Style for Control Systems

R. de Lemos

TECHNICAL REPORT SERIES

No. 675
July, 1999
A Cooperative Object-Oriented Style for Control Systems

Rogério de Lemos
Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU, UK

Abstract

In order to eliminate some of the drawbacks when using an object-oriented model to represent control systems, this paper describes an approach which extends the conventional object-oriented model by explicitly representing cooperations between objects. When using a cooperative object-oriented style for describing the software architecture of a control system, cooperations can be used to represent the control algorithms while objects are used to represent the components of the control system. The software architecture of the control system can then be implemented in terms of a design pattern which is able to encapsulate interactions between objects. The feasibility of the whole approach will be demonstrated in terms of the cruise control system benchmark case study.

Keywords: software architectures, objects, cooperations, architectural styles, design patterns

1. Introduction

Shaw has claimed that for certain applications when the operating conditions of a software system are not completely predictable, the normal software model, which corresponds to an open-loop control system, becomes inappropriate/Shaw95/. In order to compensate for this shortcoming, Shaw proposes an alternative architectural style, based on the closed-loop control system, to deal with possible disturbances that might affect the performance of the software. Instead of using process control paradigm in detriment of the object-oriented style, as it was proposed in Shaw's paper, in this paper we describe an architectural style for control systems, which extends the conventional object-oriented model by incorporating the notion of cooperations to represent the collaborative behaviour between objects. It is shown that the cooperative object-oriented style is able to model and analyse a control system problem from the perspective of classical closed-loop, thus providing the same level of expressiveness as the process control paradigm.

When describing a software system in terms of its components and their interactions, the choice of any modelling abstractions depends essentially on which aspects of the problem have to be emphasized or suppressed, hence for any particular problem the main issue is then to decide which modelling abstractions are most useful/Shaw95b/. In a control system there is a need to represent both the structural and functional aspects of the system. The structural aspects are concerned with the (physical and computational) components of the
system and their dependencies, and the functional aspects are concerned with the feedback relations between the monitored and controlled variables, which includes the stability criteria of the latter, the sequencing of events and their timing constraints. Several approaches exist in the literature that employ different kinds of representations for expressing particular aspects that have to be emphasized by different analyses. However, the usual drawback in these approaches is the inexistence of a formal correlation between the different representations which impairs the consistency checks between them.

The three styles discussed in this paper stress, differently, the structural and functional aspects of a system: the conventional object-oriented model places emphasis on the type of components that define a system (although there are other representations which support other kinds of analyses (Müller 97)), the process control paradigm places emphasis on the functional representation of systems in the form of a feedback control loop, and the cooperative object-oriented model represents both the type of components and the special relations that hold between these components. If a conventional object-oriented model and the process control paradigm were employed for modelling and analyse the software of a control system, then consistency between the two representations would be difficult to obtain due to the inexistence of a common underlying model. However, if a cooperative object-oriented model is employed for describing a control system, then the structural view is obtained from the object structure that defines the system, while the functional view is obtained from the cooperative representation which is attached to the object structure.

In a cooperative object-oriented approach, the collaborative behaviour between two or more objects is represented in terms of cooperations, which are sophisticated structural relationships able to represent complex interactions between objects. Instead of restricting the interaction between objects to that of procedure invocation, as is performed in conventional object-oriented models, cooperations define the properties of the collaborative activity that should hold during an interaction. The aim is to define an architectural style which can be used for describing software systems in terms of objects and cooperations. An advantage of using cooperations for representing the collaborative behaviour of objects is that several aggregate behaviours can be obtained without changing the behaviour of the actual objects involved in the cooperation, thus increasing the potential reuse of objects. This architectural style is particularly suitable for representing feedback control systems because under the coordination of the controller, several components of the control system are able to cooperate in order to attain the desired behaviour of the system.

The contents of the paper will be as follows. In the next section, the two architectural styles being discussed in this paper, namely, process control paradigm and cooperative object-oriented style, will be described in more detail. In section 3, it is shown how the cooperative object-oriented style can be used to represent a feedback control system. The feasibility of the cooperative object-oriented approach will be demonstrated in section 4.
in terms of the cruise control system benchmark case study, and compared with other approaches using some of the criteria established by Shaw/Shaw 93b. Finally, section 5 will present some concluding remarks.

2. Architectural Styles

An architectural style provides a specialized language for a specific class of systems that are related by shared structural and semantic properties, which include: a vocabulary of components and connectors types, configuration rules that constraint how components and connectors can be composed, semantic interpretations that provide well-defined meanings for the components, connectors, and compositions of these, and the type of analyser that can be performed on systems employing a particular style (Shaw 96). For example, a system might be described using one of the following more commonly used styles: pipes and filters, objects, repositories, layers, and interpreters. In the following, we provide a brief description of the process control paradigm as an example of an architectural style tailored for describing software for control systems (Shaw 93b), and present in more detail the cooperative object-oriented style, which takes as a basis the features of conventional object-oriented models.

2.1. Process Control Paradigm

The process control paradigm is a specialized form of dataflow architecture, and should be considered for the design of software when the task involves continuing action, behaviour, or state, when the software is embedded, and when the uncontrolled, or open loop, computation does not suffice (Shaw 93a). The process control paradigm consists of the following parts: two computational elements (process definition and control algorithm) which separate the process of interest from the control policy, three data elements (process variables, set point and sensors) which separate the information of the process from the control algorithm, and the control–loop paradigm which establishes how the control algorithm drives the process.

It is claimed that one of the advantages of the process control paradigm is its effectiveness in separating the issues related with the operation of the main process (which captures the desired functionality of the system) from the compensation for external disturbances (which captures how the system should respond to changes). In terms of the architectural elements, the former issues are represented by the process definition, while the latter are represented by the control algorithm.

2.2. 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). 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 interactions. 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.

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 composed of and the intra-relations between the object and its components, and finally, a description of the behaviour of the object. The template of an object is the following:

Object:
  
  attributes:
  constants:
  variables:
  structure:
    composed of:
    intra-relations:
  behaviour:
    initial:
    assumptions:
    normal:
    exceptional:
    failure:

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 (CD 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 template for describing a CO action is the following:

**CO Action:**
- **participants:**
- **attributes:**
  - **constants:**
  - **variables:**
- **behaviour:**
  - **initial:**
  - **normal:**
  - **exceptional:**
  - **failure:**

The initial state of a CO action represents its state when it 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/.

2.2. 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 approaches1. 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 those 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.

2.3. 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/VanHilst 96, Smaragdakis 98a/ (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 cooperations are modelling abstraction which are able to capture the properties of a collaborative activity to be

1. In a more detailed representation of a system, and outside the context of the cooperative object-oriented style, cooperations between classes can be described in terms of associations.
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. A
variant of contracts was employed in [de Lemos 95a] to specify collaborative behaviour
between objects during the phase of analysis. Activities were introduced in [Kristensen 96] as
an abstraction mechanisms to model the interplay between objects. 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.

3. A Cooperative Object-Oriented Representation of a Control Systems

The commonly accepted structure of control systems is to partition the system into three
distinct components: the plant, the operator and the controller. Basically there are two
possible configurations for control systems, either open-loop or closed-loop. In a
closed-loop control systems, which are the focus of this paper, the actual output of the
plant is measured and compared with the desired output response (or reference), and the
equal between these two will be used by the controller to calculate the new plant input which
will then cause a new actual output. In the context of this paper, we assume that the
controller is implemented by a computer, hence the need to have signal converters, such as
sensors and actuators. If the reference value can be changed depending, for example, on
the actual output of the plant, then the operator through the input and output devices can
assume this role. In the diagram of a closed-loop control system, shown in figure 1, all the
components except for the plant and the operator, which are represented by parallelograms, are elements of the controller.

A cooperative object-oriented representation of a feedback control system is shown in
figure 2. The aggregation hierarchy of the object structure of the control system is
represented in UML [Müller 97], which is supplemented with additional notation for
representing the cooperation between objects in terms of rounded rectangles. In the
cooperative object-oriented diagram, the Controlling System represents the cooperation
between the objects of the control system. The dotted lines connect diagrammatically the
components which are involved in the cooperation, and do not have any semantic meaning
attached to them.

The role of the Controlling System in this diagram is to read the information supplied by
the Sensors and Output Devices, and write into the Actuators the information produced
by the control algorithm which resides in the Controlling System. In such architecture, the
control algorithm can be easily modified without the need to change the specifications of the objects involved in the cooperation. Similarly, if for example a particular sensor of the Operator Interface is changed, except for the Controlling System, the other components of the Controller do not need to be notified in order to incorporate such change. In the following it is shown how such architecture can be useful in the modelling and analysis of an actual control system.
4. Case Study: Cruise Control System

The cruise control system, which was defined in [Booch 86], has become a benchmark case study for comparing different models and methods for the development of software [Shaw 95b]. The aim of the cruise control system is to maintain the speed of a car over varying terrain, when turned on by the driver, and when the car engine is on. The driver is also able to change the reference speed of the cruise control system, and whenever the break is applied the cruise control system ceases to be active until the driver resumes its operation.

Figure 3 shows the closed-loop architecture of the cruise control system. The active/inactive input is triggered by a variety of events which are associated with components not represented in the diagram. Depending on the reference speed set up by the Driver and the measured speed obtained from the WheelSensor, the CruiseControl calculates the new angle for the Throttle which compensates any error between the measured and reference speeds. Once the engine power changes, the speed of the Vehicle also changes which is captured by the WheelSensor. The role of the Driver on the other control loop is to establish the reference speed by adjusting the SpeedButton, depending on the value shown in the Speedometer (the Speedometer is not part of the initial problem). The parallelograms of the diagram of the feedback control system refer to those components of system which are not directly involved in the cooperation CruiseControl.

![Diagram of cruise control system](image)

**Figure 3. Architecture of the cruise control system.**

4.1. Cooperative Object-Oriented Diagram

The cooperative object-oriented diagram of the cruise control system is presented in Figure 4. This diagram only illustrates the elements of the Controller which are related with the cruise control problem, however, the other components of the vehicle which are part of the
environment of the Controller could also be represented together with the interactions that exist between the environment and the controller components.

![Cooperative object-oriented diagram of the cruise control system](image)

**Figure 4.** Cooperative object-oriented diagram of the cruise control system.

In the diagram of figure 4, the Controller is described as an aggregate of the components of the Driver Interface and Vehicle Interface. Also is part of this diagram the CO action CruiseControl which implements the control algorithm responsible for adjusting and maintaining the speed of the vehicle. In this design, the CruiseControl was partitioned into three other CO actions which correspond to the states of CruiseControl, once it is switched on: CCActive corresponds to the state in which the CruiseControl is active in maintaining the speed of the vehicle according with the established reference speed; CCInactiveAcc corresponds to the state in which the CruiseControl is inactive because either the AcceleratorPedal or the SpeedButton were pressed, and the CCInactiveBreak corresponds to the state in which the CruiseControl is inactive because the BreakPedal was pressed. The rationale behind the adopted design follows the designs defined by Ward and
Kaskar/Word 87/ and Shaw/Shaw 95b/ , except for few modifications. In the following, each of these CO actions will be formally specified in terms of Extended Real-Time Logic (ERTL) /de Lemos 96, Hall 96/ (an outline of ERTL is presented in the Appendix).

4.2. Formal Model of the CruiseControl

Using the templates previously presented, which define the architectural style, we formally specify in the following, the normal behaviour of the CO actions identified for the cruise control problem.

The CO action CruiseControl establishes when the control system is switched on, and independently of its state, if either ccButton on or engineButton on becomes false then CruiseControl is switched off.

\textbf{CruiseControl}:

\begin{itemize}
  \item \textbf{participants:}
    \begin{itemize}
      \item ccButton
      \item engineButton
    \end{itemize}
  \item \textbf{types:}
    \begin{itemize}
      \item \textbf{constants:}
      \item \textbf{variables:}
        \begin{itemize}
          \item ccOn
        \end{itemize}
    \end{itemize}
  \item \textbf{behaviour:}
    \begin{itemize}
      \item \textbf{normal:}
        \begin{itemize}
          \item \textbf{pre-condition:}
            \begin{itemize}
              \item $\forall E \in P: \Theta(\neg\text{CruiseControl}, i, t) \Rightarrow \Theta(\neg(\text{engineButton.on} \land \text{ccButton.on}), i, t)$
            \end{itemize}
          \item \textbf{invariant:}
            \begin{itemize}
              \item $\forall E \in P: \Theta(\text{ccOn}, i, t) \Rightarrow \Theta(\text{engineButton.on} \land \text{ccButton.on}, i, t)$
            \end{itemize}
          \item \textbf{post-condition:}
            \begin{itemize}
              \item $\forall E \in P: \Theta(\text{CruiseControl}, i, t) \Rightarrow$
                \begin{itemize}
                  \item $\Theta(\neg(\neg\text{engineButton.on} \lor \neg\text{ccButton.on}) \land \neg\text{ccOn}, i, t)$
                \end{itemize}
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

The following three CO actions are nested actions of CruiseControl, and only one of the actions will be activated whenever CruiseControl is on. The CO action CCActive implements the control algorithm which regulates the speed of the vehicle according to the established reference speed. There are three scenarios for the CruiseControl to become active, which is captured by the pre-condition of CCActive: when the CruiseControl is switched on, the acceleratorPedal and the speedButton are not on, and the resumeButton is on while the breakPedal is not.

The CruiseControl remains active while the acceleratorPedal, breakPedal, and speedButton remain untouched, otherwise the CO action CCActive will end its execution. These two situations are captured, respectively, by the invariant and post-condition associated with CCActive.
CCActive:

participants:
  acceleratorPedal  Potentiometer
  breakPedal        Potentiometer
  resumeButton     Switch
  speedButton      Potentiometer
  throttleActuator Servo
  wheelSensor      WheelSensor

types:
  constants:  
  variables:  
    referenceSpeed  R.

behaviour:

initial:
\[ \phi(-\text{acceleratorPedal} \land -\text{breakPedal} \land \text{resumeButton} \land -\text{speedButton}) \land (t, 0) \]

normal:

pre-condition:
\[ \forall t \in E: \phi(\neg \text{CCActive}, t, 1) \land \phi(\text{acceleratorPedal} \land -\text{breakPedal} \land \neg\text{speedButton}) \land \text{CruiseControl} \land (t, 1) \]

invariant:
\[ \forall t \in E: \phi(\text{throttleActuator}\_\text{setAngle}, t, 1) \land \phi(\text{acceleratorPedal} \land -\text{breakPedal} \land -\text{speedButton}) \land \text{CCActive} \land (t, 1) \land \text{referenceSpeed} \]

post-condition:
\[ \forall t \in E: \phi(\neg \text{CCActive}, t, 1) \land \phi(\text{acceleratorPedal} \land \text{breakPedal} \land \text{speedButton}) \land (t, 1) \]

There are two situations in which the CruiseControl becomes inactive: when either acceleratorPedal or the speedButton are pressed, or when the breakPedal is pressed. These two situations are captured by two CO actions, respectively, CCInactiveAcc and CCInactiveBreak.

During CCInactiveAcc the throttle angle is controlled by the driver whenever the acceleratorPedal is pressed or the speedButton is set to change the reference speed. The CruiseControl resumes its active state when both the acceleratorPedal and speedButton cease to be on. Another way to interrupt the execution of action CCInactiveAcc is to press the breakPedal.

CCInactiveAcc:

participants:
  acceleratorPedal  Potentiometer
  breakPedal        Potentiometer
  speedButton       Potentiometer

types:
  constants:  

variables:  

variables:

behaviour:

pre-condition:
∀i. ∀E. P: \( \neg CC \text{\text{activeBreak}}, i, t \) ⇒ \( \neg (\text{breakPedal} \land \text{speedButton} \land i, t) \)

post-condition:
∀i. ∀E. P: \( \neg CC \text{\text{activeBreak}}, i, t \) ⇒ \( \neg (\text{breakPedal} \land \text{speedButton} \land i, t) \)

The CruiseControl starts the CO action \( CC \text{\text{activeBreak}} \) whenever the breakPedal is pressed, and exits this action when the breakPedal is released and the resumeButton is pressed.

\( CC \text{\text{activeBreak}} \):

classes:

breakPedal Potentiometer
resumeButton Switch

behaviour:

pre-condition:
∀i. ∀E. P: \( CC \text{\text{activeBreak}}, i, t \) ⇒ \( \neg \text{breakPedal} \land i, t \)

post-condition:
∀i. ∀E. P: \( CC \text{\text{activeBreak}}, i, t \) ⇒ \( \neg \text{breakPedal} \land i, t \)

4.3. A Design Pattern Implementation of a Cooperation

The implementation of a CO action, as presented above, can either be centralised or decentralised. In a centralised implementation, CO actions are represented by a single object which exists at runtime to coordinate the execution of the objects participating in CO actions. Most of the existing implementations of cooperations follow the centralised approach, for instance, design pattern Mediator [Gamma 94], activities for modeling the interaction between objects [Kristensen 96], mixin classes for modeling the roles of a collaboration as parameterised classes [VanHilst 96], or mixin layers which are an extension of the latter [Smarragdakis 98]. 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 (Tramontana 98). In the following, we describe a centralised implementation of the cooperation CruiseControl in terms of the Mediator pattern (Gamma 94). The aim of this design pattern is to define an object that controls and coordinates how a group of objects interact by avoiding the objects in the group from referring explicitly to each other, thereby reducing the number of interconnections.

Figure 5 shows an implementation of the cruise control system, previously specified, in terms of the design pattern Mediator. In this implementation the cooperation ControllingSystem is associated with the Mediator class of the design pattern, and the components of the DriverInterface and VehicleInterface, represented as InputComponents and OutputComponents, are associated with the Colleague class. The three operations of the abstract class ControllingSystem are related with the definition of a CO action: CheckPre() checks the pre-conditions for starting the cooperation, ExecAct() executes the collaborative activities associated with the cooperation, and CheckPost() checks for the post-conditions for ending the cooperation. A particular instance of the operation ExecAct() is the execution of a ControlAlgorithm() which for a feedback control system (diagram of figure 1) has the following major steps: read the monitored variables, compare these measured values with the respective references, and calculate the new values for the controlled variables that will be written into the plant. In terms of the cruise control problem the ControlAlgorithm() is an operation of class CCActive and which is responsible for controlling the speed of the vehicle. This class together with CCInActiveAcc and CCInActiveBreak are composite classes of CruiseControl, and they are also specializations of ControllingSystem. Hence in this representation, the cooperations CCActive, CCInActiveAcc, and CCInActiveBreak are nested classes (representing cooperations) of CruiseControl, which is different from the implementation of a collaboration component where the nested classes correspond to the roles of a collaboration (Smaragdakis 98b).

Instead of using a pattern for the design of the cruise control system, an alternative design would be to adopt an hybrid version of the process control paradigm and the cooperative object-oriented style, in which system components are represented by objects, cooperations between components are represented by CO actions, and the behaviour of a CO action is represented by the control algorithm of the process control paradigm (the process definition corresponds to Vehicle, and most of the data elements correspond to objects). However, if more complex system are considered there are some limitations associated with the process control paradigm: it does not provide any method which supports the behavioural refinement between different representations of a control system, and it does not support communication with other controllers of the same system.
Figure 5. Class diagram of the cruise control system.

4.4. Evaluation of the Cooperative Object-Oriented Style

In the following, we evaluate the cooperative object-oriented style against the criteria established by Shaw for evaluating the different design approaches used for describing the cruise control system /Shaw 96/.
A cooperative object-oriented description although it is based on a object-oriented approach, it allows through the notion of cooperations a clear representation of the collaborative behaviour between objects, thus facilitating the process of analysing their interactions and data dependencies. In terms of safety analysis this provides the means to focus on those issues which are critical in maintaining the safe behaviour of a system (de Lemos 93a); the inherent complexity of a system is associated with the interaction of its components. Another feature of cooperative object-oriented description is that it provides the means to relate the object-oriented and state oriented views in a representation of a system: CO actions are able to define an operational model of the system by identifying the conditions that cause change in the system state and the sequencing of their occurrence.

Although a closed loop representation of a control system is not directly obtained from a cooperative object-oriented diagram, it nevertheless can be obtained by rearranging how the architectural elements are functionally interrelated: the role of the control algorithm is captured by the cooperation, while the system components (which can be software, interface or environment components) are represented by objects. In the specification of the architectural elements the details on how they should be implemented is avoided by focusing on the properties that define the elements: the components are defined in terms of the state description of their activities, while the connectors are defined by their collaborative activity. These modelling abstractions are not restricted for the description of software components and connectors. For example, the cooperative object-oriented description of the cruise control system is a representation of the real-world (environment of the controller) in terms of the interface components of the controller (InputComponents and OutputComponents). Alternatively, the design of the cruise control system could have been described in terms of the components of the vehicle, abstracting away from the computer system in which the controller will be implemented (de Lemos 93b). The relations between the real-world components and their counterparts in the controller could again be represented in terms of cooperations that would be implemented by the interface components of the controller.

5. Conclusions

A system is more than the sum of its parts; hence the different cooperations that exist between system components establish the different behaviours that the system can manifest. The motivation for using a cooperative object-oriented style for describing software systems, is to obtain simple object specifications in which all the collaborative activity with which an object is involved is specified in terms of CO actions. An advantage of this approach is that, several (simple) cooperations can be associated with a (simple) object, thus facilitating the process of reuse of both objects and CO actions.

The incorporation of cooperative actions in a conventional object-oriented model was motivated by the inadequacy of existing low level model abstractions (for example, message...
passing) to represent complex interactions between objects when describing systems at a higher level of abstraction. If the notion of software architecture is to be associated with the high level description of a software system, then there is the need to provide connectors that are at least as powerful, in terms of functional and structural decomposition, as the components that define a software architecture.

Acknowledgements

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

References


/Gamma 94/ E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Reading, MA. 1994.


Appendix — Extended Real Time Logic (ERTL)

The behavioural specification of both objects and CO actions will be made in terms of the event-action model which provides a set of primitive concepts for the modelling and analysis of phenomena associated with the computer system and its environment. In the event-action model an event serves as a temporal marker, an action is an operation which consumes a bounded quantity of resources, and a system predicate is an assertion about a system variable at a time point.

ERTL is a first order predicate logic for the modelling and analysis of hybrid systems, taking as a basis Jahanian & Mok’s Real Time Logic (RTL) /Jahanian 86/. Jahanian & Mok’s RTL uses uninterpreted predicates to relate events to the time of their occurrence, thereby providing the means for reasoning about absolute timing properties of real-time systems. The extensions provided by ERTL allow reasoning about system behaviour in both value and time domains through predicates defined in terms of system variables.

The occurrence relation ($\Theta$) captures the notion of real time by assigning a time value to each occurrence of an event. $\Theta(e_i, t)$ defines that the $i$th occurrence of event $e_i$ occurs at time $t$.

$$\forall e_i. \forall t. \Theta(e_i, t)$$

The $i$th occurrence of event MotorOn has occurred at time $t$.

A transition event is defined by the transition of a system predicate from false to true, or from true to false, at a particular time point. For a system predicate $P$, the respective transition events are $\neg P$ and $\neg \neg P$.
∀i ∈ P: \( \phi(\neg (\text{plateOnBeg} \land \text{bellOn}), i, \phi) \equiv \phi(\neg \text{plateOnEnd} \lor \neg \text{bellOn}), i, \phi) \)

The transition event which captures the instant when the conjunction of the predicates \( \text{plateOnBeg} \) and \( \text{bellOn} \) becomes false is equivalent to the transition event which captures the instant that the negation of either \( \text{plateOnBeg} \) or \( \text{bellOn} \) becomes true.

The \textit{holding relation} (\( \phi \)) captures whether a system predicate holds true at a time point. \( \phi(f, i, \phi) \) defines that a formula \( f \) holds for the \( i \)th time, at time \( t \).

∀i ∈ P: \( \phi(\text{moveDown}, i, \phi) \equiv \phi(\text{bottom} \land \text{plateOn}, i, \phi) \)

The predicate \( \text{moveDown} \) holds true if the conjunction of the predicates \( \text{bottom} \) and \( \text{plateOn} \) also holds true.