A Co-operative Object-Oriented Architecture for Adaptive Systems

Rogério de Lemos
Computing Laboratory
University of Kent at Canterbury, CT2 7NF, UK
r.delemos@ukc.ac.uk

Abstract

Adaptive systems should be able to adapt to changes that occur in their operating environment without any external human intervention. Software architectures for such systems should be flexible enough to allow components to change their pattern of collaboration depending on the environmental changes and goals of the system, without changing the actual components themselves. This paper describes a co-operative object-oriented style that is able to represent software architectures for adaptive systems. The connectors in this style, described as co-operations, embody the description of complex interacting behaviour between the architectural components. Depending on the environmental changes, the behavioural adaptability in a co-operative object-oriented architecture is achieved by replacing the connectors. The applicability of the architectural style is demonstrated in terms of a case study of a control system that has to adjust the height of a vehicle's suspension to different road conditions.

1. Introduction

In the engineering of computer based systems, there has been a trend in which the quality of services delivered by a system, in terms of its dependability, performance and cost, is directly related to the quality and extent of the computer facilities embedded in that system. Software has played a central role in this trend because of its inherent flexibility in emulating physical devices and replacing human operators. As the life span of new emerging software intensive applications increases, so does the need for software to have the capability of adapting to changes that occur in its operating environment. However, providing an adaptive capability leads to an increase in software size and complexity, which could put system integrity at risk unless the software architecture enables adaptability to be engineered in a disciplined and structured manner.

Architectural structures for systems tend to abstract away from the details of a system, but assist in understanding broader system-level concerns [15]. This can be achieved in software architectures by employing abstractions and notations that are appropriate for describing the software components, the interactions between these components, and the properties that regulate the composition of components. This paper introduces an architectural style based on components and connectors, where components embody computation, and connectors embody the description of interacting behaviour between components. However, instead of adopting the notion of a connector as an architectural element that just mediates interactions between components, in this architectural style connectors are also able to describe collaborative behaviour between components in terms of the roles played by the components [1]. That is, connectors in addition of being the place of communication between components, they are also the place of state and computation. This architectural style is associated with some of the design principles of *collaboration-based designs*. In these designs, software systems are represented as a composition of independently-definable collaborations [16]. *Collaborations* are a group of object roles together with a group of activities that determine how objects interact: the *role* of an object prescribes the activity of an object when involved in collaboration. There are several design description languages, which have richer vocabulary, and that are able to describe in the form of collaborative diagrams interactions between objects in terms of messages and events [2], and to represent the implementation of components as a composition of object roles [7]. However, these object-oriented languages lack the means for describing the properties associated with objects and their interactions, which should be an essential feature of architecture description languages. Moreover, there are several software applications in which the notion of collaboration is not sufficient to represent collaborative behaviour between components, for instance, in complex concurrent applications it is also necessary to capture the notion of co-ordination for supporting error handling between multiple interacting objects [14, 19].

In this paper, we introduce an architectural style for describing software systems in terms of co-operative object-oriented architectures. The architectural elements of this style are objects and co-operations: *objects* are modelling abstractions for representing the components of the system, while *co-operations* are modelling abstractions for representing the connectors of the system. Objects are able to participate in several co-operations through the different roles that they are able to play, while co-operations co-ordinate the interactions between the objects, through the roles that objects play. The behaviour of both objects and co-operations is described in terms of properties that have to be maintained for the system to provide the required services. This uniform way for describing the elements of a software architecture is advantageous for checking, early in the lifecycle, whether the composition of components and connectors are able to satisfy the requirements for the software system. Moreover, the description of software systems in terms of components and connectors provides the necessary architectural flexibility for describing adaptive software because of its convenience in manipulating software structures.

In the context of this paper, it is assumed that run-time adaptability of a software system can be achieved by changing the way components interact rather than changing the components themselves. For example, when adapting a command and control system to changes that might occur in its environment, instead of modifying or replacing the components of the system, the intent is to change how the components collaborate. The architectural description of adaptive software can then be made in terms of several architectural configurations that implement a wide range of behaviours. If objects and co-operations are used, then a co-operation is the architectural element that encapsulates change by representing the different collaborations between objects, thus capturing at the architecture level the dynamic composition associated with run-time adaptability. In this paper, the modelling abstraction *co-operative action* (CO action) is introduced as an architectural entity for representing co-operations. CO actions describe the collaborative activity between objects, which can either be co-operative or competitive [4]. The notion of a CO action has some similarities to that of an *action* in DisCo [12], and *joint actions* (or *use cases*) in Catalysis [7]. However in this paper we aim to give an architectural interpretation to the notion of an action (i.e. connector) while abstracting away from the actual activity. This is achieved, but focusing on the specification of the participants, and the conditions for the participants for starting, maintaining and finishing a collaborative activity.

The rest of the paper is organised as follows. In section 2 the architectural style is defined in detail by defining a meta-model for co-operative actions (CO actions). Section 3 discusses some basic issues related with run-time adaptability, and defines an architectural pattern that supports run-time adaptability. In section 4, we present a case study that will be used to illustrate the feasibility of representing adaptive software structures in terms of the co-operative object-oriented style. The architectural description of the case study is presented in section 5., and finally, section 6 concludes with a discussion evaluating our contribution and indicating directions for future work.

2. Co-operative Object-Oriented Style

An architectural style provides a specialised language for a specific class of systems that are related by shared structural and semantic properties [15]. The definition of an architectural style includes: a *vocabulary* of architectural elements (components and connectors), *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 *analyses* that can be performed on systems employing a particular style. For example, a software system might be described using one of the following more commonly used styles: pipes and filters, objects, repositories, layers, and interpreters.

Systems are defined in terms of their components and relationships among their components, which can be captured by connectors. Hence the need, when modelling systems using an object-oriented approach, to introduce *co-operative actions* (CO actions), as entities for modelling interactions between classes that characterise collaborative behaviour [4]. The use of CO actions in an object-oriented approach is motivated by the ability of CO actions to extract from the specification of a class those issues related with its collaborative activities (although preserving encapsulation property), thus avoiding a specification of a collaboration to be scattered among classes. CO actions are a variant of *co-ordinated 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 [14, 19]. In the following, we present in more detail the co-operative object-oriented style, which adopts as a basis the features of object-oriented models.

2.1. Architectural Elements

The architectural elements of the co-operative object-oriented style are *classes* as the basic components, and *CO actions* as the basic connectors. (Classes and CO actions are instantiated, respectively, into objects and co-operations.) In this style, CO actions in addition of being the place of communications, they are also the place for computation. The difference between components and connectors is that classes perform local computation, while CO actions can either co-ordinate the computation performed by the participant classes, or perform local computation that is not part of any participant class. In a CO action, the role of a class is prescribed by the activity of that class. A class may have as many roles as the number of CO actions it participates in, and the composition of these roles defines the interface of the class.

At the architectural level no relational information is spread across classes, only CO actions contain relational information (how a co-operative object-oriented architecture is implemented is discussed later, however a CO action can be instantiated into an UML association when interactions between classes are simple service requests). An advantage for only CO actions to contain relational information is that, once a co-operative object-oriented architecture is instantiated, co-operations can be added or removed without interfering with the implementation of objects, thus improving modularity and reusability of the software.

2.1.1. Classes

As in object-oriented models, classes in the proposed approach support the representation of both structural and behavioural aspects of a system. A class 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 class, a description of its structure in terms of a collection of components in composed of and the intra-relations between the classes and its components, and finally, a description of the behaviour of the class. The behaviour field includes the initial state of the object, and behavioural assumptions or (consistency invariants) associated with the class. The behavioural field also includes the specification of the complete space of the behaviour of the class, 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).

2.1.2. Co-operative Actions (CO Actions)

CO actions are employed in the specification of co-operative behaviour between classes. CO actions can either co-ordinate the activities to be performed by the classes, or execute some activity that is not associated with any of the class participants of the CO action. A CO action is described by a template with the following fields: the CO action's name, declaration of attributes in terms of the names and types of the participants of the CO action, constants and variables local to the CO action, and the specification of the collaborative behaviour of the classes 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 classes participating in the co-operation or the state of the variables local to the CO action. Associated with the description of normal behaviour, pre-condition and post-condition establish the respective conditions for a set of classes to start and finish a particular collaborative activity, and the invariant establishes the conditions that should hold while the collaborative activity is being performed. For the description of systems that are potentially concurrent, there is the need to consider the conditions that define the pre- and post-conditions to be trigger (necessary and sufficient) conditions. The successful execution of a collaborative activity occurs when 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 that 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 co-operative and competitive concurrency by integrating two complementary concepts: *conversations* [13] and *transactions* [9]. Conversational support is used to control co-operative concurrency and to implement co-ordinated 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 [14, 19].

2.2. Configuration Rules

For the description of systems, the configuration rules of the co-operative object-oriented style define how objects and co-operations can be combined. In the following, we will focus on the static properties of the co-operative object-oriented style, rather than describing how the architectural elements should be configured depending on their dynamic properties.

In a co-operative object-oriented architecture each class and CO action has a unique name. Classes can participate in more than one CO action, and at least two classes have to be associated with a CO action, thus avoiding the "dangling" of CO actions. A CO action defines and is defined by the roles of the classes, thus creating the context in which classes collaborate. For describing the architecture of a software system, two different diagrams are employed: a *class diagram* that describes the relationships between components, and a *CO action* diagram that describes the relationships between connectors. These diagrams provide a compact representation of the software system, which can be completed with a more detailed textual description. For a rigorous description of software system, a first order predicate logic can be used for describing the properties of the architectural elements.

Instead of using two diagrams for describing the architecture of a system, we could have employed a single diagram to represent both the components and their interacting activities, in a similar way of *collaboration diagrams* of the Catalysis approach [7]. From our experience, such diagrams are adequate for systems that have few numbers of components, and which have interactions involving few components. However, the type of system we are concerned with are complex systems containing several components that are able to engage in interactions which might involve most of the system components. Hence the preference for having one diagram of components and other of connectors, which facilitates the structural representation of complex systems.

At the architectural level of representation, the only type of relationships acceptable between the nodes of both class and CO action diagrams are generalisations, and aggregations/compositions (white and black diamonds represent aggregation and composition, respectively). The diagrams of figure 4 provide an example how to represent software systems using the co-operative object-oriented style.

2.3. Meta-Model of a Co-operative Action (CO Action)

In the following, we define in more detail the concept of a CO action according with the semantic description of UML [18]. A CO action is considered as a specialisation of *Classifier* in the Core package of UML Foundation, which also includes the following specific forms: *Class*, *DataType* and *Interface*. The diagram of figure 1 shows the concrete constructs that define the relationships of a CO action.

The purpose of a *CO action* is to declare the attributes and the collaborative activities that fully describe the structure and behaviour of co-operations. All the co-operations instantiated from a CO action will have attribute values matching the attributes of the CO action descriptor, and will support the collaborative activities defined by the CO action descriptor. An *Attribute* is a name property of a CO action that describes the range of values that instances of the property may hold. These attributes can either refer to remote attributes defined by the classes which take part in the co-operation, or local attributes to the CO action which includes the list of participants that take part in the CO action. A *Collaborative Activity* is the implementation of a service that can effect the behaviour of two or more objects. Associated with the collaborative activity of a CO action there are a *Pre-condition* that defines the start of the activity, and one or more *Post-conditions* which define end of the activity. A collaborative activity of a CO action is specified in terms of a name, together with an *Invariant* which defines the collaborative activity, a set of *Operations* which establish the normal behaviour of the co-operation, and a set of *Exceptions* which establish the exceptional behaviour of the co-operation. The exceptions are defined in terms of exceptional *Events* and *Handlers*. The *Interface* of a CO action is the collection of collaborative activities that define the service to be delivered by the CO action.

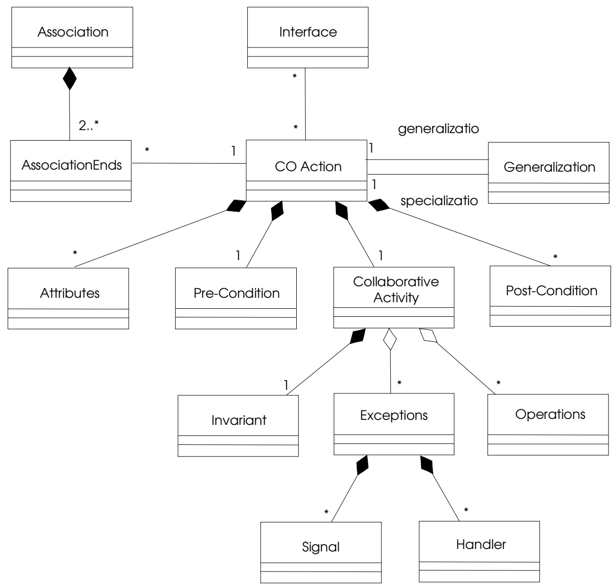


Figure 1. Meta-model of a CO action.

A *Generalisation* is a taxonomic relationship between a more general element and a more specific element. A CO action can have generalisations to other CO actions, but not with classes. The full CO action descriptor of a CO action is derived by inheritance from its own segment declaration and those of its ancestors.

Following the Catalysis approach, we could instead have considered a single diagram to represent both classes and CO actions. If this is the case then the notion of association has to be included. An *Association* is a structural relationship that specifies a connection between classifiers, e.g. classes and CO actions. Associations are described in terms of a name, at least two *AssociationEnds* (which define the roles and the properties that should be observed of the classifier participating in the association), and a multiplicity property. An association may represent an aggregation between CO actions, but not between CO actions and classes. An aggregation specifies a whole-part relationship between the aggregate ("the whole") and a component ("the part"). Composition is a strong form of aggregation, and requires that a part instance be included in at most one composite at time, although the owner may be changed over time.

3. Architectural Pattern for Supporting Adaptability

3.1. Run-Time Adaptability

Run-time adaptability is the ability of a software system to adapt itself to changes that occur either internally or in its operating environment. A system can either change its behaviour or its structure, although most adaptive systems contain a mixture of these two types of adaptability:

- In *behavioural adaptability*, the system structure remains the same while the architectural elements of the system can be modified or replaced. For example, a component could be modified for the provision of new services, or a connector could be replaced for changing the communication protocols between the components.
- In *structural adaptability*, the system behaviour remains the same while the configuration of the architectural elements changes. For example, depending on changes that might occur in the environment of the system, it might be necessary to reconfigure the system for coping with different workloads. At the design level, an example of structural adaptability is adaptive fault tolerance [11].

In a co-operative object-oriented architecture the degree of run-time adaptability of a software system depends on the flexibility of components changing their pattern of collaboration. Instead of having a software system based on components that are individually able to provide a wide range of services, the proposed approach relies on the ability of components to reconfigure their collaborations while they remain unchanged. The aim of this paper is to define an architectural pattern that allows behavioural adaptability to be incorporated in co-operative object-oriented architectures.

3.2. Co-operative Object-Oriented Architectural Pattern

In a co-operative object-oriented system, behavioural adaptability is obtained by changing how objects co-operate, and the selection of a co-operation depends on the state of the collaboration between the objects. An architectural representation of such system should describe the collaborative activities between classes in terms of CO actions. The conditions for selecting a co-operation should be part of the definition of a CO action, and these conditions are related to either the internal state of the co-operation or the states of the objects (i.e. roles) participating in the co-operation. Hence the architectural representation should be able to describe, across different states, the behavioural adaptability of the collaborative activities between classes.

In this section, we define an architectural pattern that supports run-time adaptability targeted for co-operative object-oriented architectures. An *architectural pattern* provides guidance for combining architectural elements in established and proven ways. The aim is not to define mechanisms for adding, removing and replacing objects, which provide the means for systems to dynamically reconfigure. Instead, we intend to define an architectural pattern that facilitates the representation of system configurations that provide the basis for the system to adapt to changes that occur in its environment.

The intent of the *State* design pattern is to allow an object to alter its behaviour when its internal state changes [8]. In this paper we claim that this design pattern can be also used to provide the required support for a co-operation to alter its behaviour when its state changes. The structure of the design pattern *State*, in terms of CO actions, is shown in figure 2. The abstract *Select* CO action defines the interface common to all the CO actions that represent the different states of the co-operation (an instance of COAction). COAction delegates all state-specific collaborations to the *Select* CO action, and depending on its state, COAction uses an instance of a specialised CO actions (COAction1, COAction2,...) of the abstract *Select* CO action to implement a collaboration. Using the architectural pattern *Select*, we are able to obtain an effective and structured representation of behavioural adaptability using the co-operative object-oriented style.

The applicability of this architectural pattern can be demonstrated in terms of two examples. The first example deals with intelligent motorways, where autonomous vehicles are able to travel in platoons at high speeds while maintaining minimal distances between themselves. Depending on the weather conditions, the properties of the vehicles do not need to change, instead changes can be associated with the collaborations that are responsible for maintaining the required distances between the vehicles. Another example comes from the world of the Internet. An Internet bookshop might require, for payment assurances, additional information from a not so trustworthy customer (the vice-versa being equally possible). Instead of changing the components representing the bookshop and the customer, the collaboration between these two entities can be changed, depending on the roles taken by one of its participants.

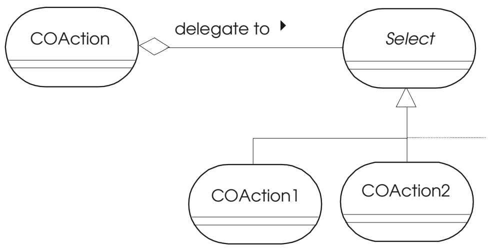


Figure 2. *State* design pattern in terms of CO actions.

4. Description of the Case Study: Electronic Height Control System (EHCS)

The electronic height control system (EHCS) controls the height of a vehicle by regulating the individual heights of the wheels through a pneumatic suspension. The aim of this system is to adjust the chassis level depending on the road conditions, in order to improve driving comfort and keep the headlight load-independent [17].

For this case study three distinct types of road are considered, namely, off-road, gravel and motorway. For each type of road we define a set point and two sets of tolerance intervals, as shown in the diagram of figure 1. In each of the four wheels there is an pneumatic suspension which is able to control the height of an individual wheel. Whenever the height of a wheel is outside the outer tolerance interval, the controller has to bring the height into the inner tolerance interval around the set point.

The major components of the EHCS are a valve and a height sensor at each wheel, and an escape valve and a compressor to be shared by all the wheels, as shown in the diagram of figure 3. The suspension height is increased by opening the wheel valve, closing the escape valve, and pumping air into the suspension. The suspension height is decreased by releasing air from the suspension by opening the escape valve, and the valve of the wheel from which the height has to be reduced. The compressor and the escape valve cannot be used simultaneously, priority is given to the compressor when both have to be used. It is assumed that the height values provided by the sensors are mean values of the actual readings from which the disturbances, like road holes, are eliminated.

The aim of this case study is to define a software architecture that enables the EHCS to adapt at run-time to changes that occur in the system environment. In terms of the height control system, the adaptability element is related with selection of the appropriate control algorithm depending on the type of road. In terms of the EHCS software architecture, the software components remain the same, while the pattern of collaboration changes between the components.

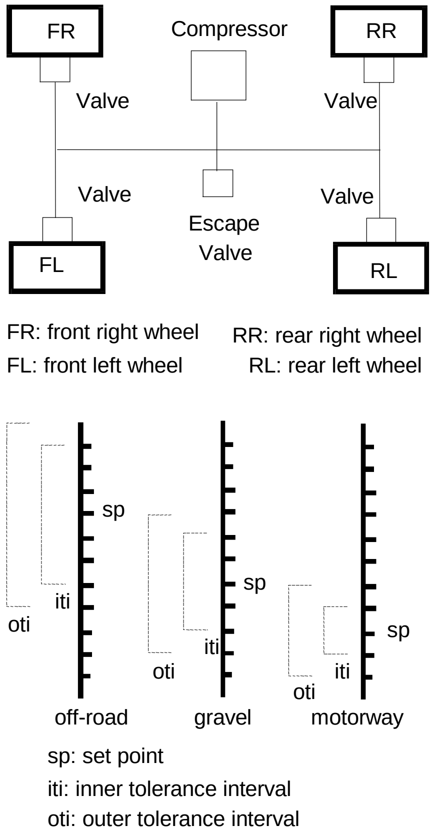
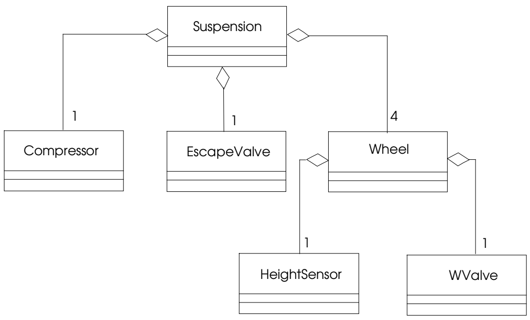


Figure 3. Diagrammatic representation of the EHCS.

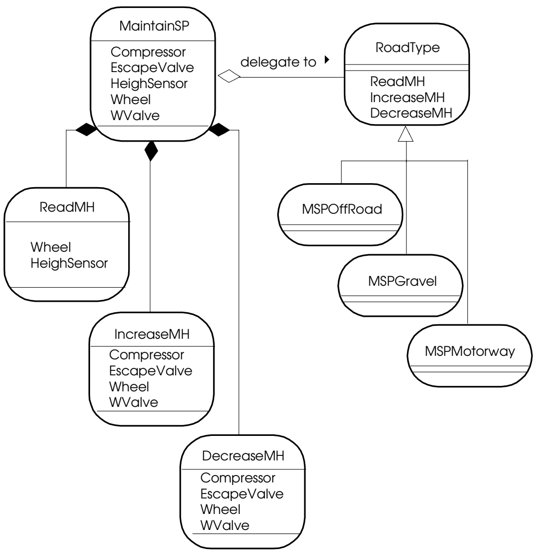
5. A Software Architecture for the EHCS

In this section the software architecture for the electronic height control system (EHCS) of a vehicle suspension is established in terms of the co-operative object-oriented style, previously defined. The diagram of figure 4 represents the class and CO action diagrams for the EHCS. The notation follows UML [2], except for the CO actions that are represented as boxes with rounded corners.

The CO action MaintainSP is responsible for maintaining the height of the suspension around the set point. It is composed by three other CO actions: ReadMH which is responsible for updating Wheel with the value of mean height of the suspension, and IncreaseMH and DecreaseMH which are responsible, respectively, for increasing and decreasing the suspension height of a wheel. Depending on road condition, a different control algorithm is necessary for maintaining the height of the suspension, hence MaintainSP can be specialised into MSPOffRoad, MSPGravel, and MSPMotorway. For example, it might be necessary, depending on the road conditions, to establish different tolerance levels for IncreaseMH and DecreaseMH, or to establish time intervals for updating (Δupdate) the value of the mean height in class Wheel.



(i) class diagram



(ii) CO action diagram

Figure 4. Co-operative object-oriented diagrams for the EHCS.

In this paper, the dynamic behaviour of CO actions is specified using a property-oriented formalism instead of operational formalism (like Statecharts, which are part of UML). The reason for this is that, at the architectural level, we rather to focus on the properties of an adaptive system, than on how a design should be implemented. In the following, the CO actions will be formally specified in terms of Extended Real-Time Logic (ERTL) [3, 10] following the template previously presented. In this paper, the behavioural specification of CO actions will be restricted to the normal behaviour. It is not in the scope of this paper to deal with exceptional and failure behaviours, which were presented elsewhere [5,6]. For the sake of brevity, and to avoid repetition, only the composite CO action MaintainSP will be specified. The specifications of MSPOffRoad, MSPGravel, and MSPMotorway follow directly from MaintainSP. We assume that MaintainSP, which is considered appropriate for all types of road, is replaced by other three CO actions that encapsulate control algorithms that are specific for particular types of road.

The CO action MaintainSP co-ordinates the activities between the components of the Suspension for maintaining the mean height of a Wheel around the established set point. The co-ordination of the collaborative activities is partitioned into three CO actions, detailed below. The definition of MaintainSP states that the pre-condition for MaintainSP to be activated is when EHCS is switched on (ehcs.on), and it will remain activated until the EHCS is switched off, which is also captured by the invariant. The provision of adaptive software will be based on the specialisation of MaintainSP for the different types of road, which will be presented at the end of this section.

MaintainSP:

attributes:

participants:

c	*Compressor*
ev	*Valve*
w	*Wheel*
w.hs	*HeightSensor*
w.wv	*Valve*
ehcs	*EHCS*

behaviour:

initial:

$\Phi(\neg ehcs.on, 1, 0)$

normal:

pre-condition:

$\forall t \bullet \forall i \in \Im^{+}: \Theta(\nearrow maintainSP, i, t) \Leftrightarrow \Theta(\nearrow ehcs.on, i, t)$

invariant:

$\forall t \bullet \forall i \in \Im^{+}: \Phi(maintainSP, i, t) \Leftrightarrow \Phi(ehcs.on, i, t)$

post-condition:

$\forall t \bullet \forall i \in \Im^{+}: \Theta(\searrow maintainSP, i, t) \Leftrightarrow \Theta(\searrow ehcs.on, i, t)$

The CO action ReadMH captures the collaboration between Wheel and HeightSensor, and is responsible for updating periodically the Wheel's variable for the mean height of the suspension, which is obtained from the HeightSensor. The pre-condition for normal behaviour establishes that ReadMH starts periodically every Δupdate. The invariant states that for ReadMH to be active the current update has still to be made and the interval for the next reading has not expired. The post-condition is captured by two transition event predicates that specify the necessary and sufficient conditions for the co-operation to end: the variable w.height has been updated, or the time interval available for updating that variable has expired.

ReadMH:

attributes:

participants:

w	*Wheel*
w.hs	*HeightSensor*

variables:

Δupdate	*Real*

behaviour:

initial:

normal:

pre-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\nearrow \text{readMH}, i, t) \Leftrightarrow$
$\exists t_1 \bullet \Theta(\nearrow \text{readMH}, i\text{-}1, t_1) \wedge t = t_1 + \Delta\text{update}$

invariant:

$\forall t \bullet \forall i \in \Im^+$: $\Phi(\text{readMH}, i, t) \Leftrightarrow$
$\exists t_1 \bullet \Theta(\nearrow \text{readMH}, i, t_1) \wedge t < t_1 + \Delta\text{update}$

post-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\searrow \text{readMH}, i, t) \Leftrightarrow$
$\Theta(\nearrow(\text{w.height=w.hs.value}), i, t) \vee$
$(\exists t_1 \bullet \Theta(\nearrow \text{readMH}, i, t_1) \wedge t \geq t_1 + \Delta\text{update})$

The CO action IncreaseMH is responsible for increasing the mean height of the suspension once a minimum threshold is reached. The pre-condition for normal behaviour establishes that the CO action starts when the Compressor is off (captured by c.on), the EscapeValve and WheelValve are closed (captured by variables ev.open and w.wv.open, respectively), and the minimum height threshold is reached (captured by varaible w.height). While the mean height of the suspension is being increased the Compressor should be on, the EscapeValve closed, the WheelValve open. Once the mean height is within the inner tolerance interval (iti), IncreaseMH ceases to be active.

IncreaseMH:

attributes:

participants:

c	*Compressor*
ev	*Valve*
w	*Wheel*
w.hs	*HeightSensor*
w.wv	*Valve*

variables:

iti, oti	*Real*

behaviour:

initial:

$\Phi(\neg\text{c.on} \wedge \neg\text{ev.open} \wedge \neg\text{w.wv.open}, 1, 0)$

normal:

pre-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\nearrow \text{increaseMH}, i, t) \Leftrightarrow$
$\Theta(\nearrow(\neg\text{c.on} \wedge \neg\text{ev.open} \wedge \neg \text{ w.wv.open} \wedge$
$(\text{w.height}<(\text{w.setPoint-oti}/2))), i, t)$

invariant:

$\forall t \bullet \forall i \in \Im^+$: $\Phi(\text{increaseMH}, i, t) \Leftrightarrow$
$\Phi(\neg\text{ev.open}, i, t) \wedge \Phi(\text{ w.height}<(\text{w.setPoint-oti}/2), i, t)$

operations:

$\forall t \bullet \forall i \in \Im^+$: $\Phi(\text{w.height}<(\text{w.setPoint-oti}/2), i, t) \Rightarrow$
$\Phi(\text{c.on} \wedge \text{w.wv.open}, i, t)$

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\nearrow((\text{w.setPoint+iti}/2)>\text{w.height}>$
$(\text{w.setPoint-iti}/2\text{increaseMH})), i, t) \Rightarrow$
$\Theta(\nearrow(\neg\text{c.on} \wedge \neg \text{ w.wv.open}), i, t)$

post-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\searrow \text{increaseMH}, i, t) \Leftrightarrow$
$\Theta(\nearrow(\neg\text{c.on} \wedge \neg\text{ev.open} \wedge \neg \text{ w.wv.open} \wedge$
$((\text{w.setPoint+iti}/2)>\text{w.height}> (\text{w.setPoint-iti}/2))), i, t)$

The description of CO action DecreaseMH follows the same pattern of IncreaseMH. However, for reducing the mean height of the suspension the Compressor should be off, and the EscapeValve and WheelValve should be open. (At this stage of development, we are not concerned with the priority associated with the Compressor, instead the implementation of this requirement should be part of a CO action responsible for co-ordinating the controllers of the individual wheels.)

DecreaseMH:

attributes:

participants:

c	*Compressor*
ev	*Valve*
w	*Wheel*
w.wv	*Valve*

variables:

iti, oti	*Real*

behaviour:

initial:

$\Phi(\neg\text{c.on} \wedge \neg\text{ev.open} \wedge \neg \text{ w.wv.open}, 1, 0)$

normal:

pre-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\nearrow \text{decreaseMH}, i, t) \Leftrightarrow$
$\Theta(\nearrow(\neg\text{c.on} \wedge \neg\text{ev.open} \wedge \neg \text{ w.wv.open} \wedge$
$(\text{w.height}>(\text{w.setPoint+oti}/2))), i, t)$

invariant:

$\forall t \bullet \forall i \in \Im^+$: $\Phi(\text{decreaseMH}, i, t) \Leftrightarrow$
$\Phi(\neg\text{c.on}, i, t) \wedge \Phi(\text{w.height}>(\text{w.setPoint+oti}/2), i, t)$

operations:

$\forall t \bullet \forall i \in \Im^+$: $\Phi(\text{w.height}>(\text{w.setPoint+oti}/2), i, t) \Rightarrow$
$\Phi(\text{ev.open} \wedge \text{w.wv.open}, i, t)$

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\nearrow((\text{w.setPoint+iti}/2)>\text{w.height}>$
$(\text{w.setPoint-iti}/2\text{increaseMH})), i, t) \Rightarrow$
$\Theta(\nearrow(\neg\text{ev.open} \wedge \neg\text{w.wv.open}), i, t)$

post-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\searrow \text{decreaseMH}, i, t) \Leftrightarrow$
$\Theta(\nearrow(\neg\text{c.on} \wedge \neg\text{ev.open} \wedge \neg \text{ w.wv.open} \wedge$
$((\text{w.setPoint+iti}/2)>\text{w.height}> (\text{w.setPoint-iti}/2))), i, t)$

According with the proposed approach, outlined in section 3.2, an adaptable software system for the EHCS can be obtained by applying the design pattern *State* [8] to the CO actions that capture the collaborative activity between the components of the Suspension. Referring to the CO action diagram of figure 4, the behaviour of MaintainSP depends on the state of *RoadType*, and according to this state, the behaviour of MaintainSP must change at run-time. Instead of defining a CO action for all types of road, the design pattern *State* allows to partition MaintainSP into other CO actions, namely, MSPOffRoad, MSPGravel and MSPMotorway. These three CO actions are specified to be mutually independent, and their selection during run-time depends on the type of road.

6. Conclusion

In this paper we have presented how collaborations between objects can be exploited when defining software architectures for adaptive systems. As a basis for the proposed approach, we have assumed that objects are rigid entities, and the basis for adaptability depends on how they collaborate. Hence all the additional features which enables an object, or a group of objects, to adapt to changes that occur in its environment are not captured in the object itself, instead they are defined in the co-operations in which the object is a participant. For describing the architectures for adaptive software systems we have defined a co-operative object-oriented style where components are the classes, and connectors are the co-operative actions (CO actions). As architectural elements, CO actions capture the behavioural dependencies between the classes that are related with the adaptability features of an object, or group of objects.

Although the definition of a CO action was presented in the context of components (objects) which have very simple structures, the aim of the work is to obtain a more general definition of a CO action which can be used as a sophisticated connector for structurally more complex software components. For that, it might be necessary to define a CO action as an architectural pattern (or framework) which can be instantiated into several domain related applications. Also in this paper, we have only considered the type of run-time adaptability where the components remains unchanged while the behaviour of the system changes, however, depending on the type of application and the purpose of the system, another types of adaptability may also be considered.

References

[1] R. Balzer. Instrumenting, Monitoring, and Debugging Software Architectures. http://www.isi.edu/divisions/index.html.

[2] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley. Reading, MA. 1998.

[3] R. de Lemos, J. G. Hall. "Extended RTL in the Specification and Verification of an Industrial Press". *Hybrid Systems III*. Lecture Notes in Computer Science 1066. Eds. R. Alur, T. A. Henzinger, E. Sontag. Springer-Verlag. Berlin, Germany. 1996. pp. 114-125.

[4] R. de Lemos, A. Romanovsky. "Coordinated Atomic Actions in Modelling Object Cooperation" *Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Kyoto, Japan. April 1998. pp. 152-161.

[5] R. de Lemos, A. Romanovsky. "Exception Handling in a Cooperative Object-Oriented Approach". *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Saint Malo, France. May 1999. pp. 3-13.

[6] R. de Lemos. "Analysis of the Safety Properties of a System from the Viewpoint of its Components Interactions". *Proceedings of the 9th Brazilian Conference on Fault Tolerant Systems*. Campinas, SP. Brazil. July 1999. pp. 35-48.

[7] D. F. D'Souza, A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley. Reading, MA. 1998.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1994.

[9] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Kaufman Publishers. San Mateo, CA. 1993.

[10] J. G. Hall, R. de Lemos. "ERTL: an Extension to RTL for the Specification, Analysis and Verification of Hybrid Systems". *Proceedings of the 8th EUROMICRO Workshop on Real-Time Systems 96*. L'Aquila, Italy. June 1996. pp. 3-8.

[11] K. H. Kim, T. Lawrence. "Adaptive Fault Tolerance in Complex Real-Time Distributed Applications". *Computer Communication, Vol. 15(4)*. May 1992. pp. 243-251.

[12] R. Kurki-Suonio. "Fundamentals of Object-Oriented Specification and Modelling of Collective Behaviours". *Object-Oriented Behavioural Specifications*. Eds. H. Kilov, and W. Harvey. Kluwer Academic Publishers. Boston, MA. 1996. pp. 101-119.

[13] B. Randell. "System Structure for Software Fault-Tolerance". *IEEE Transactions on Software Engineering Vol. SE -1(2)*. 1975. pp. 220-232.

[14] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, A. F. Zorzo. *Co-ordinated Atomic Actions: from Concept to Implementation*. Technical Report 595. Department of Computing Science. University of Newcastle. UK. 1997.

[15] M. Shaw, D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc. Upper Saddle River, NJ. 1996.

[16] Y. Smaragdakis, D. Batory. "Implementing Reusable Object-Oriented". *Proceedings of the 5th International Conference on Software Reuse (ICSR'98)*. Victoria, Canada. June 1998.

[17] T. Stauner, O. Müller, M. Fuchs. "Using HYTECH to Verify an Automative Control System". *Hybrid and Real-Time Systems*. Ed. O. Maler. Lecture Notes in Computer Science 1201. Springer-Verlag. Berlin, Germany. 1997. pp. 139-153.

[18] UML Semantics (version 1.1). Rational Software. September 1997.

[19] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu. "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery". *Proceedings of the 25th International. Symposium on Fault-Tolerant Computing*. Pasadena, CA. 1995. pp. 499-509.