Objecting “Beyond Objects”

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

Abstract

In adaptive control systems, although the behaviour of the environment of the system might be unpredictable, the behaviour of the system itself has to be predictable at all times. An approach to obtain such a predictable behaviour from a system, is to wrap the physical components of the system (the components that lie outside the computing system) with a layer of software. This would allow the computational representation of the physical components to interact in such a way that the behaviour of the whole system can adapt to changes that occur in its environment. The feasibility of the whole approach will be demonstrated in terms of the cruise control system benchmark case study.

Keywords: software architectures, objects, coordinated atomic actions, reflection, adaptive systems

1. Introduction

Shaw has claimed that, for certain applications, when the operating conditions of a software system are not completely predictable the traditional software paradigm, which is correspondent to a open–loop control system, becomes inadequate /Shaw 95/. In order to compensate this shortcoming, she has proposed an alternative software architecture, based on the closed–loop feedback control system, to deal with possible disturbances that might affect the performance of the software.

Instead of using control–loop paradigm in detriment of the object oriented paradigm, as it was proposed in Shaw’s paper, this paper describes an approach for developing software for adaptive control systems, which integrates (with some modifications) both paradigms. The adaptability feature, in our control systems, introduces a new degree of complexity, which requires for the software of the controller to reconfigure itself according with the changes that occur in its environment. In other words, a controller has to change its control strategies depending of the state of the physical process (which includes disturbances that either affect the physical process or the components of the interface between the physical process and the controller), and the state of other controllers that make part of the controller’s environment. The software architectural approach proposed in this paper, is essentially based on the object oriented paradigm, in which the cooperations between objects are represented by Coordinated Atomic Actions (CA actions) /de Lemos 97/, and the behaviour of the objects is represented by an augmented subset of the architectural
elements of control–loop paradigm (in a configuration more appropriate for the modelling and analysis of hybrid systems /Grossman 93/).

The contents of the paper will be as follows. In the next section, the control–loop paradigm is described in more detail, together with an object oriented approach that will provide the basis for the development of software for adaptive control systems. The feasibility of the integrated approach will be demonstrated in section 4, in terms of the cruise control system benchmark case study. Section 3 presents the approach being proposed for the development of software for adaptive control systems, in terms of the elements of the software architecture and a method for incremental development of software. An adaptive version of the cruise control system is discussed in section 5, in terms of the proposed software architecture. Finally, section 6 will present some concluding remarks.

2. Software Architectures for Control Systems

Shaw has introduced in her paper a set of process control paradigms (resembling traditional configurations for automatic control system), which provide the basis for defining architectures for software that controls continuous physical processes. In the definition of the software paradigm for process control, the computational elements of the architecture are the physical process and the controller, the data elements are the system variables and the setpoints, and the relationships between the computational and data elements are established by the control–loop paradigm. Although it is claimed that the proposed architecture is effective in separating issues of the physical process from ones of the controller (where the software will eventually reside), it is not clear the reason for making this distinction at the software architectural level, if such essential separation can be captured by the software development method /de Lemos 95/.

The method for developing software for adaptive control systems, is similar to the approach adopted for the incremental development of software for safety–critical control systems /de Lemos 95/. That approach is predominantly top–down, in the sense that, it starts with the formal modelling and analysis of the components of the environment of the computing system (i.e. physical objects), from which successive structural decompositions and behavioural refinements are applied until an architectural specification of the software (i.e. computational objects) is obtained. In this approach, while objects are employed to model system structure and component behaviour, the other modelling abstraction, Coordinated Atomic Actions (CA actions), are used to model cooperations between objects /de Lemos 97/. The concept of CA action was initially introduced as a design mechanism for structuring complex concurrent activities and supporting error recovery between multiple interacting objects /Xu 95/.

Objects and CA actions are used as modelling abstractions throughout the software lifecycle, and depending on on the stage of the development, appropriate techniques are
employed to formally specify the complete space behaviour of the components (and their respective cooperations) which are involved in a safety related activity. The method for software development is based on the following major steps, which can then be applied recursively until the desired level of component granularity is achieved: identification of the system structure in terms of its major components (i.e. objects), identification of all cooperations between components (i.e. CA actions), specification of the behaviours of the object classes, and specification of the behaviours of CA actions.

For describing object behaviour, and their cooperations, the software architecture for the controller, proposed by Shaw, can be effective for certain classes of hybrid system (i.e. systems which combine both continuous variable and discrete event dynamics /Grossman 93/) because is able to clearly separate, within the same structure, the control laws, from the events that triggered the control laws. Such an architecture could easily be extracted from an object class aggregation hierarchy, in which cooperations between objects are also represented. In general terms, this can be achieved by associating the objects with the process definition of the control−loop paradigm, and the CA actions with the control algorithm. However, there are some limitations associated with the control loop paradigm if more complex system are considered, namely, it does not provide any method which supports the behavioural refinement between different representations of a system, and it does not support communication with other controllers of the same system. These limitations can be overcome by adopting alternative architectures, such as, the architecture for multiple−agent hybrid controllers /Hayes−Roth 90, Nerode 93/.

In summary, considering that a system is more than the sum of its parts, and it is the different cooperations that exist between the system components that establish the different behaviours that the system can manifest. The aim of the approach, described above, is to obtain simple object specifications in which all the collaborative activity with which an object is involved, is specify in terms of CA 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 CA actions.

3. 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. The aim of the cruise control system is to maintain the speed of a car over varying terrain. A detail description of the case study can be obtained in /Booch 86, Shaw 95/.

The partial representation, from the viewpoint of the cruise control system, of the object class aggregation hierarchy of the car, is presented in figure 1. Together with the object classes, the cooperations between the objects are also identified. The behaviour of the
identified objects and their cooperations can then be formally specified, using either axiomatic /Cepin 97, de Lemos 97/ or operational formalisms /de Lemos 93/.

The two basic services that could be associated with the *Throttle* are *RotateClose* and *RotateOpen*, while the service which controls the position of the *Throttle*, for the purpose of maintaining the speed of the car constant, could be specified as CA action *CruiseControl* because it involves the cooperation between several components. While the cooperation links associated with *CruiseControl* resemble, with minor modifications, Booch’s object

*Figure 1. Object class aggregation hierarchy and CA actions.*
oriented design for the cruise control system, the internal architecture of CruiseControl could be represented by the same controller architecture defined by Shaw.

4. Software Architectures for Adaptive Control Systems

The aim of this work is to define a set of software architectures, thereby providing a powerful support for the development of software for adaptive control system (i.e. systems that are capable dynamically adapting to changes that occur in their environment, without any human intervention).

From the wide range of adaptive control systems, the class of systems being targeted by the proposed approach are those systems which predominantly support process descriptions instead of data descriptions, according with the classification adopted in /Simon 81, MacFarlane 93/. The reason for constraining the solution domain, thus excluding solutions based on neural networks and genetic algorithms, stems from the need to have software which is able to behave predictably under any unforeseen circumstance. As well as the requirement for adaptability, there is also a need to consider safety and timing requirements that are vital in many control systems applications. Hence, an essential requirement for these software systems is the provision of predictable behaviour while the environment in which the software system is embedded might not be completely predictable.

An approach to obtained such a predictable behaviour from a computer based system, is to wrap the physical components of the system (the components that lie outside the computing system) with a layer of software. This would allow the computational representation of the physical components to interact in a such away that the behaviour of the whole system can adapt to changes that occur in its environment.

In general terms, the envisaged approach for the development of software for adaptive control systems is almost the same as the approach described in section 2. In that approach, the aim of the software lifecycle activities associated with the stage of domain and requirements analysis was to identify, in the environment of the computing system, the physical objects and their cooperations, and during the stages of design and implementation of the software, the aim was to define the computational objects which, in most cases, were computational representations of their physical counterparts.

An essential distinction between a control system and an adaptive control system, is the capability of the latter to be more tolerant to disturbances that might occur in its environment while maintaining the same quality of service. This capability is achieved by providing a range of control strategies which are able to handle different control scenarios. In order to support this additional feature there is a need to improve the process of modelling and analysis for the purpose of identifying potential collaborative behaviour and the respective operational limits, between the system components, and to provide support
for the internal organization of the object and its interface to the environment for the purpose of accommodating the different control strategies and the process of decision making when choosing between alternative control strategies. The choice between alternative, and perhaps conflicting, control strategies should be made according to the internal state of the component, and the state of its environment (which includes the state of the plant, and the state of other controllers).

The provision of adaptability is related with the flexibility of computational objects changing their pattern of cooperation, and this can only be achieved if the definition of the objects are kept as simple as their physical counterparts. Conversely, if objects are made robust, in order to support a wide range of services, then this might impair their flexibility in establishing other potential cooperations. In other words, except for the basic services of the object, all the other services provided by the object should be established according with the cooperations with other objects. The operational limits for adaptability will be established from the constraints, in terms of physical laws and rules of operation, to be associated with the definition of objects and their cooperations.

A reflective object oriented design and implementation for such approach, using, for example, OpenC++ or ReflectiveJava, will associate with the base–level of the object, its data structure, basic operations, and operational (safety) limits, and with the meta–level of the object, its cooperations with other objects (represented by CA actions) and the process of decision making /Maes 87, Kiczales 92/. The process of decision making, that will provide the basis for solving conflicting situations arising from the adaptation process, will be based on a game theoretical approach /Rosenschein 94/.

5. Case Study: Adaptive Cruise Control System

The need for adaptability in the case study discussed in section 3 might rise from the necessity to adapt automatically the driving conditions of the vehicle (for example, change the suspension set-ups), and in particular the cruise control system, to various terrain conditions, whether the road is slippery or not, and type of surface, whether is tarmac or gravel. Depending on the road conditions different control strategies could be selected in order to optimize the performance of the vehicle. These control strategies would be
represented in terms of CA actions in the meta-level of the object definition, and their selection will depend on the states of the controller and its environment.

![Diagram](image)

**Figure 2.** Reflective implementation of *Throttle* and *CruiseControl*.

The need for adapting to stimulus from the environment might not be restricted to the variables of the physical process being controlled, it might be the case that the controller needs to interact with other controllers, in order to select optimal control strategies and establish control set points. This is the case in automated motorway systems when a car joins or leaves a platoon of cars /Hedrick 94/.

6. Conclusions

Although the paper by Steels /Steels 94/ could have provided the insight for future research on (object oriented) software agents, it was actually Shaw’s paper /Shaw 95/ that has provided the motivation for the proposed approach for the development of software for adaptive control systems. This approach, which comprises a software architecture and a development method, combines the object oriented and the control–loop paradigms, and provided the basis for defining a general architecture for software agents, targeted for control systems applications: while an augmented object–oriented paradigm (which incorporates CA actions) can be used for describing the structure of systems in terms of its agents and their cooperations, the control–loop paradigm can then be used for describing the individual behaviour of the software agents.

Acknowledgements
The author would like to acknowledge the financial support of EPSRC/UK ADAPT and SafeGames projects.

References


