Describing Evolving Dependable Systems using Co-operative Software Architectures

Rogério de Lemos

Computing Laboratory
University of Kent at Canterbury, CT2 7NF, UK

r.delemos@ukc.ac.uk

Abstract

This paper describes an architectural approach that facilitates the modelling and analysis of dependable systems that are built from untrustworthy components whose designs, we assume, cannot be changed. The approach is based on the definition of an architectural style in which connectors are considered as first class entities, which embody the description of collaborative behaviour between components. This style is shown to be particularly suitable for describing system components that have to evolve in order for the system to provide dependable services. The feasibility of the proposed architectural style in dealing with evolving dependable systems is demonstrated in terms of the gas station case study.

1. Introduction

One of the problems when building large-scale software systems out of existing software components are the architectural mismatches that might occur between system components [12]. An architectural mismatch occurs when the assumptions that a component makes about another component, or the rest of the system, do not match. Mismatches occur when building dependable system out of untrustworthy components, which is essentially an evolution problem since the system, its components, and their interactions have to change according to the required dependable needs. However, the approach taken in this paper instead of changing the system components, it relies in changing the interactions between the components by adding or changing the roles that components play when providing different kinds of services. The focus being taken is at the architectural level, since it is the software architecture that tends to affect the flexibility of software in adapting to changes.

Architectural structures for systems tend to abstract away from the details of a system, but assist in understanding broader system-level concerns [21]. This is achieved by employing architectural styles that are appropriate for describing systems in terms of *components*, the interactions between these components - *connectors*, and the properties that regulate the composition of components - *configurations*. The major difference between a component (a unit of computation or data store), and a connector (a unit of interaction among components or rules that govern that interaction) is that connectors may not correspond to compilation units in an implemented system [15].

In this paper, we define an architectural style that is appropriate for representing the evolution of components, such as commercial off-the-shelf components (COTS), legacy systems, and component systems of systems-of-systems. The type of constraint usually associated with this sort of component is the inability of changing the component's design according to the different services required from it. Alternative means have to be found for permitting change to be made on the services delivered by a component without having to change the actual component itself. For example, if mechanisms for enforcing dependability of services have to be added to an untrustworthy component, then it is necessary to provide structural means for incorporating the required changes, without having to change the internals of the component. The proposed co-operative architectural style offers the appropriate means for structuring complex applications that are intrinsically collaborative in their nature, which is a feature of most fault tolerant mechanisms for enforcing the dependability of services. This paper discusses the provision of fault tolerance in the context of how exception handling mechanisms can be added to untrustworthy components.

The rest of the paper is organised as follows. In section 2, we briefly discuss the architectural

representation of evolving components. Section 3 presents a variant of the gas station case study, which is used for showing how different architectural styles are able to describe evolving dependable systems. In section 4, we describe the co-operative style being proposed, which is suitable for representing collaborative activity between components, an essential feature for the provision of dependable services. In section 5, we re-visited the gas station case study for showing the representation of exception behaviour in co-operative architectures. Finally in section 6, concludes with a discussion evaluating our contribution.

2. Architectural Representation of Evolving Components

Although there are several surveys that compare how architectural description languages handle the representation of an evolving system [15], in this paper, we focus instead on the architectural elements and constraints that form the basis of these languages, that is, the architectural styles [21. We believe that, the capability of an architectural representation to adapting to changes is more a feature of a style rather than of a language.

The architectural representation of components that have to change for enforcing the delivery of dependable services is discussed from the viewpoint of obtaining trustworthy interfaces from untrustworthy ones. But before that, we discuss our understanding of interface mismatch (which is the general case of architectural mismatch), the techniques to deal with it, and the architectural styles that are more appropriate to support these techniques.

2.1. Interface Mismatches

An *interface* between two components can be defined as the assumptions that components make about each other [19]. An *interface mismatch* occurs when the interface assumptions of both components do not match up. That is, the assumptions that describe the service provided by a component are different from the assumptions that describe the services required by a component for behaving as specified [18]. When building systems from existing components, it is inevitable that incompatibilities between the service delivered by a component, and the service that the rest of the system expects from that component give rise to interface mismatches. These mismatches are not exclusive to the behavioural aspects of components; mismatches may also include non-behavioural aspects, such as dependability, which can be related to component failure mode assumptions or its safety integrity levels.

Since the concern of this work is with architectural mismatches that are dependability related, in the following we describe the basis for obtaining dependable services from untrustworthy components by changing the interface of these components. There are two types of interfaces that can be associated with a component, depending on the dependability services required from it:

- **Untrustworthy interfaces** – are the interfaces of a component that are not able to deliver the required dependable services;
- **Trustworthy interfaces** – are the modified interfaces of a component that are able to deliver the required dependable services;

For obtaining trustworthy interface-components out of untrustworthy interfaces, we can rely on Neuman's concept of *depends on* [16] which is based on the concept of *depends upon* from Parnas [20]. If a component *depends upon* other component, then if the latter does not meet its requirements then the former may not also meet its requirements. On the other hand, the concept of *depends on* introduces the generalised sense of dependence in which greater trustworthiness can be achieved despite the presence of less trustworthy components. There are several mechanisms that enables to achieve a resulting trustworthiness greater than the constituent components [17].

2.2. Architectural Styles supporting Evolution

There are three classes of techniques for dealing with interface mismatch which are based on inserting code for mediating the interaction between the components [11]:

- **Wrappers** – are a form of encapsulation whereby some component is enclosed within an alternative abstraction, thus yielding to an alternative interface to the component;
- **Bridges** – translate some of the assumptions of the components interfaces. Different from a wrapper, a bridge is independent of any particular component, and needs to be explicitly invoked by an external component;
- **Mediators** – exhibit properties of both wrappers and bridges. Different from a bridge, a mediator incorporates a planning function that results in the run-time determination of the translation. Similar to wrappers, mediators are first class software architecture entities due to their semantic complexity and runtime autonomy.

From the three techniques above, we are specifically concerned with bridges. According with Shaw, there are two architectural styles that are considered appropriate as a structural solution for mediating interface mismatches [22]: the data-centred or repository style, and the communicating processes style. The first style is a repository that is accessed and updated by several clients, which have their own control threads. The main architectural representatives of this style are traditional

databases or file systems as passive repositories, and blackboard system as active repositories. The main features of using the repository style for representing an evolving software system is the relative independence of the clients from each other, and their independence towards the repository. Moreover, software architectures that instantiated from this style are scalable because clients can be easily added and modified without affecting the other clients.

If the clients are built as independently executing processes and the repository is considered passive, the interaction between a client and the repository can be represented by the communicating processes architectural style, being the client-server a well-know subtype [22]. In this style, a server provides services to clients, from which services requests are originated. The interchange between the server and the clients can either be synchronous or asynchronous. In the latter case, the client has its own thread of control. In architectures based on this style, interface mismatches of an evolving component are fixed by adding new *interface-components*. An evolving component is considered a black box server since its design is assumed to remain unchanged. The interface-components are the clients that implement the necessary bridges that provide additional services, by adapting the component behaviour. Also in this architecture, an interface-component can eventually evolve by associating to it other interface-components.

For the definition of a conceptual model for representing architectures, we follow the ontology established by the architectural interchange language Acme [13]. The basic elements for architectural description are *components*, *connectors*, and their *configurations*. Components and connectors have interfaces that are defined as a set of *ports* and a set of *roles*, respectively. A *port* is a point of interaction with a component; hence a component can have multiple interfaces by using different types of ports. A *role* is a point of interaction with a connector, and defines a participant of an interaction that is represented by the connector. In this conceptual model, a component plays a role, through its ports, when collaborating with other components, and that component may play different roles according with the collaborations in which is involved.

In the following sections, we exemplify, in terms of the gas station case study, how the evolution of software components can be achieved using the client-serve architectural style.

3. The Gas Station Case Study

The gas station system consists of customers who come to a gas station to obtain gas for their vehicles, cashiers who sell the gas, and pumps that discharge the gas. A representation of money is exchanged between customers and cashiers, and between cashiers and pumps, and a representation of gas is exchanged between customers and pumps. In this paper, we consider that the gas station can evolve in two ways: depending on the gas available on the tanks, gas can be rationed to the customers, and the gas station can evolve from manual to automatic payment.

With the aid of collaboration diagrams [6], in the following we proceed to describe an evolving gas station in terms of three possible scenarios. We start with the scenario in which the customer gets the gas first, and then pays for it (GetPayGas). The customer selects first the pump and gets the gas, and then communicates the ID of the pump to the cashier for he/she to read the amount, and then the customer pays the gas. When the transaction is finished, the cashier releases the pump. This scenario is represented in collaboration diagram of figure 1.

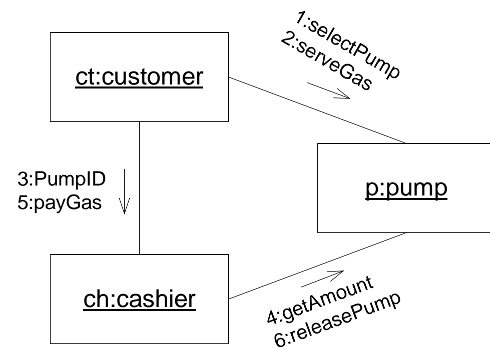


Figure 1. Customer gets the gas first and then pays (GetPayGas).

In the second scenario, we consider the case in which the rationing of gas is introduced depending on the amount of gas available on the tanks. This implies that the pump has to check first the availability of the gas before delivering it to the customer.

In the third scenario, we consider a self-service gas station, where the customer does not interact with the cashier (SelfServeGas). The sequence of events is: the customer first selects the pump, gets the gas and pays for it, and finally the cashier releases the pump. This scenario is represented in collaboration diagram of figure 2.

The changes in this scenario are to eliminate the interaction between the customer and cashier, and provide means for the customer to pay directly to the pump but without having to change the pump.

The exceptional behaviour to be considered is the situation where the pump runs out of gas while the customer is being served. This is the typical scenario where the pump alone cannot deal with this exception, because it has to raise an exception to the customer and cashier for them to handle the abnormal behaviour.

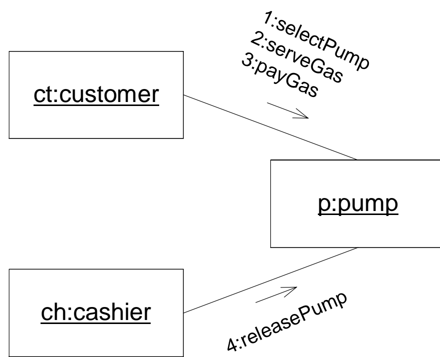


Figure 2. A self-service gas station (SelfServeGas).

3.1. Evolving the Gas Station using Client-Server Architectures

For illustrating how the client-server architectural style may be used to represent evolving components, we employ the gas station case study, and for the sake of brevity, we focus on the evolution of a single component, rather than the whole system. A simplified architectural representation of the gas station is shown in figure 3, in terms of its three basic components and relations between them (for simplifying the architectural diagrams, we have chosen to represent several connectors between two components as a single line). The ports associated with each of the components are the following:

- **cashier** – getAmount, releasePump, payGas, and givePumpID.
- **customer** – payGas, givePumpID, selectPump, and serveGas.
- **pump** – getAmount, releasePump, selectPump, serveGas, availableGas, and loadPump.

We assume that connectors are simple RPCs, each with two roles: the caller and the callee.

In addition to the three basic components of the gas station, we have also included in the diagram of figure 3 the interface-components responsible for ensuring that gas is rationed when there is a shortage (IC_serveGas), and that the customer is able to pay for her/his gas without having to interact with the cashier (IC_payGas). The ports associated with these interface-components are the following:

- **IC_serveGas** – gasAvailable, loadPump, and serveGas;
- **IC_payGas** – givePumpID, getAmount, and payGas.

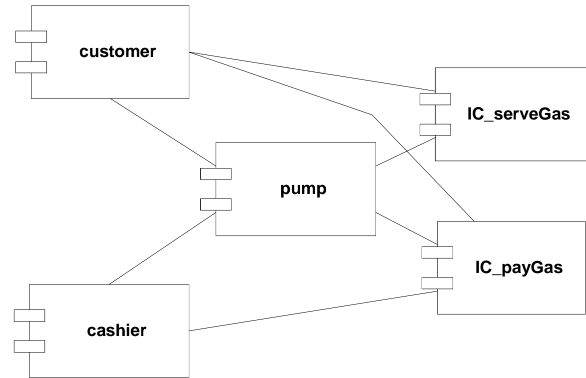


Figure 3. Architectural representation of a self-service gas station (SelfServeGas).

The behavioural description of components and interface-components will be made in terms of the UPPAAL model [14]. The basis of the UPPAAL is the notion of timed automata extended with data variables, such as integer and Boolean variables. The automata consist of a collection of control nodes connected by edges. The control nodes of the automata are decorated by *invariants* that are conditions expressing constraints on the clock values. The edges of the automata are decorated with *guards* that express a condition to be satisfied for the edge to be taken, *synchronisation actions* that are performed when the edge is taken, and *clock resets* and *assignments* to integer variables.

In the following, we specify using extended time automata only the component pump, and its two associated interface-components. The automaton for the pump is presented in figure 4. All the ports associated with the pump are specified as synchronisation labels (p_getAmount?, p_releasePump?, p_selectPump?, p_serveGas?, p_availableGas?, and p_loadPump?), and the activity of each port is specified in terms of pre- and post-conditions, following the configuration GetPayGas of the gas station.

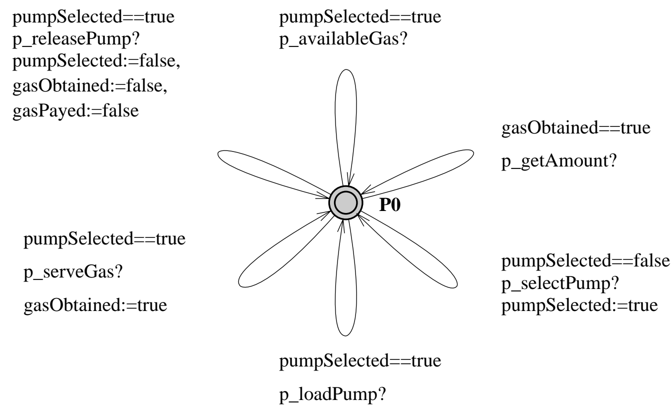
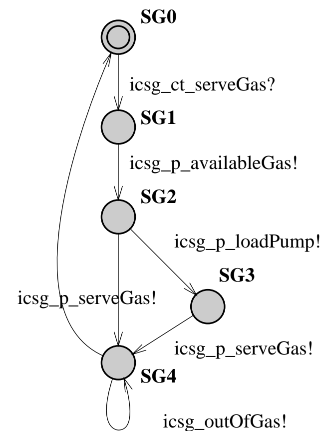


Figure 4. Automaton Pump.

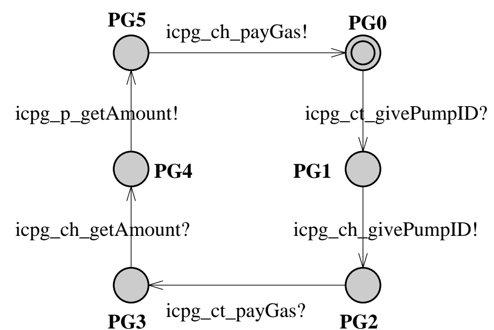
The automata for the two interface-components are presented in figure 5. The interface-component IC_serveGas is responsible for intercepting any request from the customer to serve gas (ct_serveGas?), and check before delivering the gas (serveGas!) the amount of gas available in the tanks of the station (gas Available!). Rationing is imposed (loadPump!) if the gas available is below a certain threshold. The self-loop in control mode IC_serveGas.SG4 captures an exception being raised when the pump runs out of gas (the exceptional behaviour of the gas station is discussed in section 5).

The interface-component IC_payGas allows the customer to pay his/her gas directly on the pump without interacting with the cashier. This additional functionality is provided to the pump without changing the actual pump. All the interactions needed for accomplishing this service can be made transparent to the customer. For example, the need for the customer to provide the identification of the pump (pumpID!) is a pre-condition, in the original system, for the customer to pay the gas, and this interaction can be made transparent in the process of the customer selecting the pump and taking the gas. For the cashier to accept the payment for the gas (payGas!), via the IC_payGas, it is necessary first for the cashier to receive the amount of gas taken (getAmount!).

In this section, we have shown how the client-server architectural style can be employed for representing evolving components. However, the architectural elements of this style become inadequate when changes involve the representation of collaborative activity between system components, which is the case when describing means for enforcing the delivery of dependable services, such as, exceptional behaviour. In the next section, we present an architectural abstraction that allows capturing the behavioural dependencies between interface-components, thus facilitating the structuring of systems at the architectural level, which enhances the design and implementation of dependable systems.



(a) Automaton IC_serveGas.



(b) Automaton IC_payGas.

Figure 5. The automata for the interface-components.

4. Co-operative Architectural Style

Considering a collaboration to be a set of components and a set of rules (or allowed behaviour) that determines how components interact [23], the client-server style on its own is not effective in representing component-based software systems that are collaborative in their nature, that is, there are more than two components interacting for providing a specific service. Additional abstractions have to be devised for describing interactions between the system components (or interface-components), and the properties associated with these interactions. In this section an architectural style is identified which enhances the client-server style by providing an abstraction, in the form of a "sophisticated" connector, that captures the collaborative behaviour between architectural components.

Instead of relying on the provision of means and mechanisms that focus on supporting the adaptation of components [1, 7], the proposed approach for evolution is based on adapting the interactions between components. The motivation for this approach comes from the current trend of component-based software engineering that relies on the re-use of ready available software components, such as COTS and legacy software, which are not expected to undergo any type of change. Hence, it is assumed that components remain unchanged, while the behavioural dependencies between the components may change according to the evolving needs of the software.

In order to represent interactions between the components for the purpose of facilitating the incorporation of change, components and connectors are employed as architectural abstractions: while *components* embody computation, *connectors* embody the description of interacting behaviour between components. However, in the proposed approach, connectors in addition of mediating interactions between components, they are also able of describing collaborative behaviour between components in terms of the roles played by the components [3]. That is, connectors in addition of being the place of communication between components, they are also the place of state and computation. This approach has some similarities with the features of *collaboration-based designs*. In these designs, software systems are represented as a composition of independently-definable collaborations [23].

In the following, we present the main features of the co-operative architectural style in terms of its architectural elements and configuration, and how exception behaviour can be represented.

4.1. Architectural Elements

Components support the representation of both structural and behavioural aspects of a system. A component is described by a template with the following fields: a **name**, declaration of **attributes**, a description of its **structure** that enumerates the components which is **composed of** and the **intra-relations** between the components and its subcomponents, and finally, a description of the **behaviour** of the component, which identifies the ports of the components. The behaviour field includes the **initial** state of the component, and behavioural **assumptions** or (consistency invariants) associated with the component. The behavioural field also includes the specification of the complete space of the behaviour of the component, in terms of its **normal**, **exceptional** and **failure** behaviours.

C*o-operative connectors* in addition of being the place of communications, they are also the place for computation. In this style, the difference between components and co-operative connectors is that components perform local computation, while connectors encapsulate the collaborative activity between the several components: either co-ordinates the activity of the components, or performs some local computation that is not part of any component. Hence, our notion of connectors can be seen as a collection of roles played by the components taking part in a collaborative activity. As mentioned before, a component can play several roles, thus allowing the component to participate simultaneously in several collaborative activities.

A connector is described by a template with the following fields: a **name**, declaration of **attributes** in terms of the names and types of the **roles** involved in the collaborative action, which identifies the roles of the connectors, and the specification of the collaborative **behaviour**. The behaviour field includes the **initial** state, and the specification of the complete behaviour space of the connector, in terms of its **normal**, **exceptional** and **failure** behaviours. Associated with the description of *normal behaviour*, **pre-condition** and **post-condition** establish the respective conditions for a set of components to start and finish a particular collaborative activity, the **invariant** establishes the conditions that should hold while the collaborative activity is being performed, and the collaborative **operation** to be performed under the control of the connector. The successful execution of a collaborative operation 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*, a **handler** replaces the operation, and defined in terms of its start and finish events. 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 collaborative operation, once an exception has occurred. In addition of specifying the collaborative operation in

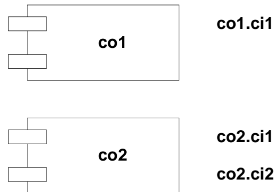
terms of what should do, it is equally important to specify what should not do, mainly those behaviours that can affect the safety of the system. Two types of *failure behaviours* have to be considered: failures of **omission**, and failures of **commission**.

4.2. Architectural Configuration

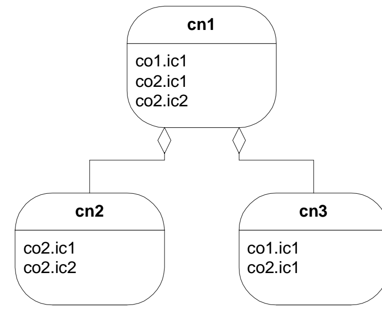
For the description of systems, the configuration rules of the co-operative style define how components and connectors can be combined.

In a co-operative architecture each component and connector has a unique name. The ports of a component can be linked to several connectors, and at least two components have to be associated with a connector, thus avoiding the "dangling" of connectors. A connector defines and is defined by the roles of the components, thus creating the context in which components collaborate. Only connectors contain relational information. An advantage of this is that, connectors can be added or removed without interfering with the implementation of components, thus restricting the impact of change.

Two different diagrams describe the co-operative architecture of a software system: a *component diagram* describing the relationships between components, and a *connector diagram* describing the relationships between co-operative connectors. The behavioural complexity of connectors can be as high as that of components, hence the need to represent the relationship between connectors in the same way to that of components. Moreover, it was chosen not to represent the links between two diagrams because this will give rise to complex diagrams with a multitude of links, since a connector can be linked with several components and a component can be linked to several connectors. An example of a co-operative architecture is shown figure 6. The depicted system is composed by two components (co1 and co2) that are interconnected by three connectors (cn1, cn2, and cn3), which means that these two components are able to interact in three different ways. For example, the connector cn1 captures the collaborative activity between the three roles associated with components co1 and co2. Also it is shown that the collaborative activities associated with connectors cn2 and cn3 are nested to connector cn1.



(a) Components diagram.



(b) Connectors diagram.

Figure 6. Co-operative software architecture.

In the following, we present how the co-operative connector can be used to represent exceptional behaviour in a self-service gas station. For that, we combine the client-server and the co-operative architectural styles. In this hybrid architectural style there are two ways of expressing connectors. If connectors capture simple flow of information between components, of the RPC type, then they are represented by lines, as depicted in figure 3. However, if connectors capture complex interactions between components, then they are represented by rounded boxes, as depicted in figure 6.

4.3. Representation of Exceptional Behaviour

Exception handling is structuring technique that facilitates the design of dependable computing systems by encapsulating the process of error recovery [8]. It is a typical activity that requires co-ordinated co-operation between components in order to recuperate the system into a known consistent state, once an abnormal behaviour is detected. Dealing with concurrent manifestations of several faults at different phases of system development has been recognised as a serious problem which has not received enough attention [2]. Only recently exceptional behaviour has been considered in the general context of the software lifecycle [10], and in the particular context of the requirements phase [24]. In this paper, we consider exceptional behaviour at the architectural level, similarly to what was presented in [9], but introducing the notion of a sophisticated connector to deal with collaborative activities.

Figure 7 shows the representation of exceptional behaviour of a co-operative connector in terms of timed automata extended with data variables [14]. The automaton on the left starts a co-operation when the pre-condition is true (pre==true), it executes the collaborative operation (operation:=true) while the invariant holds (inv==true), and exists the co-operation when the post-condition of normal behaviour is true (post_normal==true). The representation of exception behaviour is related to the raising of an exception

(signal!) that starts a handler on the automaton on the right. Once the handler is executed (handler:=true), the co-operation is finished assuming that the post-condition for exception behaviour is true (pos_exceptional==true).

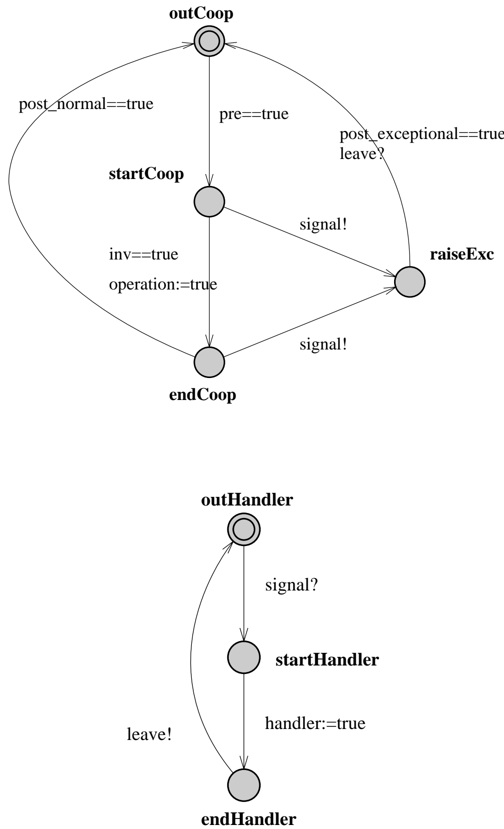


Figure 7. Automata representation of exceptional behaviour.

In the following we present how the above model of a co-operative connector can be used for representing exceptional behaviour in the self-service gas.

5. Exceptional Behaviour in Co-operative Architectures

In this section, we present how to obtain a trustworthy interface by using the co-operative architectural style. While the interface-components of the client-serve style provide the adequate support for changing the interface of components, the connectors of the co-operative style are able to represent complex behavioural dependencies between components, which are typical of exceptional behaviour. In the case of the gas station, the exceptional behaviour is related to the situation where the pump runs out of gas while the customer is being served.

A new architectural representation of the self-service gas station is shown in figure 8, in terms of its three basic components, the two interface-components previously specified, and a co-operative connector. The purpose of the co-operative connector is to structure the collaborative activity between the components, in order to avoid the propagation of errors, and to encapsulate the process of error recovery. The role that a connector has in a co-operative architecture is similar to that of co-operative actions (CO actions) in the context of object-oriented design, which is to co-ordinate the collaborative activity between classes, involving also the handling of exceptions [10].

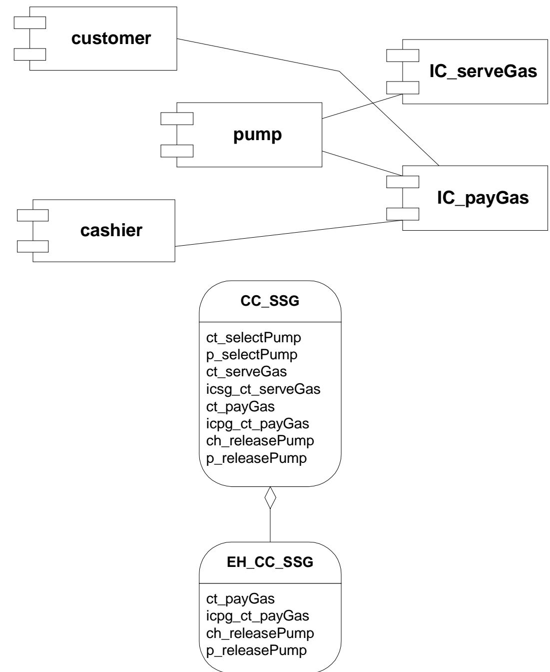


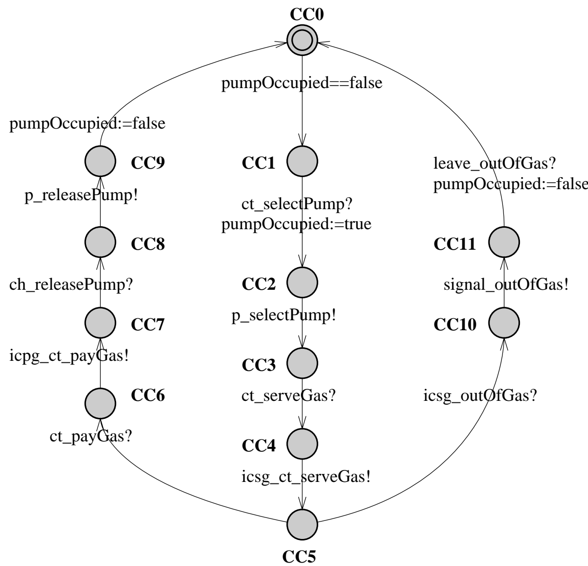
Figure 8. A co-operative architecture of a self-service gas station (SelfServeGas).

Comparing the diagrams of figure 8 with the architectural representation of the gas station of figure 3, we notice that some of the connectors of the latter have been removed. These connectors have been replaced by a co-operative connector, which encapsulates key collaborative activity between the system components.

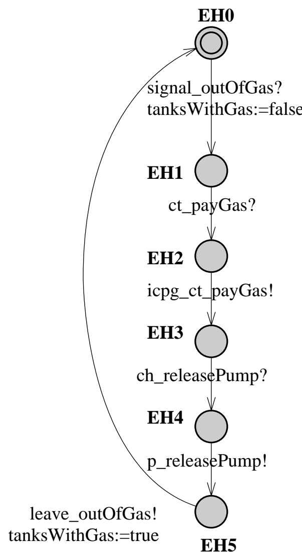
The behavioural description of the co-operative connector CC_SSG is presented in figure 9, in terms of extended timed automata. The automaton on the left (CC_SSG) captures the collaborative activity between the components, while the automaton on the left (EH_CC_SSG) describes the handler when an exception is raised.

There are two flows on the automaton of the left: the normal behaviour flow that establishes the sequence of collaborative activities associated with the purchase of gas

in a self-service station (ct_payGas?), and the exceptional behaviour flow that captures an exception when gas runs out while the customer is being served (signal_outOfGas!). This exception is initially raised by the interface-component IC_serveGas (icsg_outOfGas!), which is then propagated to the co-operative connector CC_SSG (icsg_outOfGas?).



(a) Automaton CC_SSG.



(b) Automaton EH_CC_SSG.

Figure 9. The automata for the co-operative connector and its exception handler.

According with the co-operative connector CC_SSG, the collaborative activity between the components of the gas station starts when the pre-condition holds true (tanksWithGas==true and pumpOccupied==false). When the flow of control reaches CC_SSG.CC5, it can either follow the normal flow by accepting the gas payment from the customer (ct_payGas?), or follow the exceptional behaviour by capturing the exception being propagated by IC_serveGas. Following the normal flow, the collaborative activity is finished when the post-condition holds true. On the other hand, when the exceptional flow is followed the exception handler EH_CC_SSG is invoked. This handler will then performed a series of activities to leave the gas station in a consistent state (another alternative measure could be to send a notification message to both the customer and cashier), before stopping the gas station. Once the handler finishes its operations, the flow leaves the handler (leave_outOfGas!), and finishes the co-operation when the post-condition for the exceptional behaviour (tanksWithGas==false) holds true.

The behavioural description of the architectural components, in particular the co-operative connector, using extended time automata has facilitated the task of model checking whether the architectural model of the gas station is able to satisfy the required system properties. Once an acceptable architectural representation of the gas station is obtained, the system can be designed and implemented using either the approach described in [10] or the one described in [5].

6. Conclusions

The claim of this paper is that existing architectural styles, such as the client-server, may not effectively represent the fault-tolerant mechanisms that allow obtaining trustworthy components from untrustworthy ones. Instead, new forms for representing software systems are necessary if there is the need to deal with dependability related architectural mismatches, which might be associated with the necessity for obtaining dependable services from untrustworthy components. The co-operative architectural style discussed in this paper considers connectors as first class entities that embody the description of collaborative behaviour, which provides the basis for implementing error recovery in the presence of faults. The combining usage of the co-operative and the client-server styles form the basis of a notation that facilitates the structural representation of dependability means that permit obtaining trustworthy component interfaces from untrustworthy ones.

The feasibility of the proposed approach was demonstrated in the context of an evolving gas station: during the evolution of the system, there was no need to change its components because the focus of change was

instead the interactions between components. This approach has shown to be particularly appropriate when dealing with exceptional situations, such as when the gas runs out while the client was being served. In such situations, it is fundamental to structure the interactions between components for restricting error propagation and facilitating error recovery. However, when considering systems that do not contain interactions involving several components that have to agree on the conditions for entering, holding and leaving co-operations, the proposed architectural solution might prove too expensive during run-time because of the number of checks that are required. For these cases, architectural solutions like the client-server should be sufficient for the problem at hand.

In this paper, the behaviour of architectural elements is represented in terms of extend timed automata, but the intent is to define an architectural description language that is able to capture the structural and behavioural features of the co-operative style. The objective is to obtain a language with a formal underpinning that would facilitate the precise high-level description and analysis of dependable systems that are built from untrustworthy components. Currently, we are also investigating how the co-operative style can be useful in enforcing other dependability attributes at the architectural level.

Acknowledgements. The author would like to thank Alexander Romanovsky for the fruitful discussions on the initial drafts of this paper.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa. "Abstracting Object-Interactions Using Composition-Filters". *Proceedings of the Workshop on Object-Based Distributed Programming at the European Conference on Object-Oriented Programming (ECOOP'93).* Kaiserslautern, Germany. 1993. Lecture Notes in Computer Science 791. R. Guerraoui, O. Nierstrasz, and M. Riveill (eds). Springer-Verlag. 1994. pp. 152-184.

[2] A. Avizienis. "Toward Systematic Design of Fault-Tolerant Systems". *Computer 30 (4).* April 1997. pp. 51-58.

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

[4] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Addison-Wesley. 1998.

[5] D. Beder, B. Randell, A. Romanovsky, C. Rubira. "On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems". *4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01).* Magdeburg, Germany. May 2001.

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

[7] J. Bosch. "Superimposition: A Component Adaptation Technique". *Information and Software Technology*. Elsevier. 1999.

[8] F. Cristian. "Exception Handling and Tolerance of Software Faults". *Software Fault Tolerance* . Ed. M. Lyu. Wiley. 1995. pp. 81-107.

[9] V. Issarny, J.-P. Banâtre. *Architecture-Based Exception Handling*. 2000.

[10] R. de Lemos, A. Romanovsky. "Exception Handling in a Cooperative Object-Oriented Approach". *Proc. of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99).* Saint Malo, France. May, 1999. pp. 3-13.

[11] R. DeLine. "A Catalog of Techniques for Resolving Packaging Mismatch". *Proceedings of the 5th Symposium on Software Reusability (SSR'99).* Los Angeles, CA. May 1999. pp. 44-53.

[12] D. Garlan, R. Allen, J. Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard". *IEEE Software 12(6).* November 1995. pp. 17-26.

[13] D. Garlan, R. Monroe, D. Wile. "Acme: An Architecture Description Interchange Language". *Proceedings of CASCON'97*. November 1997.

[14] K. G. Larsen, P. Pettersson, W. Yi. "UPPAAL in a Nutshell". *International Journal on Software Tools for Technology Transfer 1(1–2).* October 1997. pp. 134–152.

[15] N. Medvidovic, R. N. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages". *IEEE Transactions on Software Engineering TSE 26(1).* January 2000. pp. 70-93

[16] P. G. Neumann. *Architectures and Formal Representations for Secure Systems*. Technical Report CSL 96-05. SRI International. Menlo Park, CA. October 1995.

[17] P. G. Neumann. *Practical Architectures for Survivable Systems and Networks*. SRI International. Menlo Park, CA. January 1999. http://www.csl.sri.com/neumann/arl-one.html

[18] P. Oberndorf, K. Wallnau, A. M. Zaremski. "Product Lines: Reusing Architectural Assets within an Organisation". In [4]. pp. 331-344.

[19] D. L. Parnas. "Information Distribution Aspects of Design Methodology". *Proceedings of the 1971 IFIP Congress. Amsterdam*. 1971.

[20] D. L. Parnas. "The Influence of Software Structure on Reliability". *Current Trends in Programming Methodology I*. Ed. R. Yeh. Prentice-Hall. 1977. pp. 111-119.

[21] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall. 1996.

[22] M. Shaw. "Moving from Qualities to Architecture: Architecture Styles". In [4]. pp. 93-122.

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

[24] A. van Lamsweerde, E. Letier. "Integrating Obstacles in Goal-Driven Requirements Engineering". *Proceedings of the 20th International Conference on Software Engineering*. Kyoto, Japan. April, 1998. pp. 53-62.