Safety Analysis of an Evolving Software Architecture

Rogério de Lemos

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

r.delemos@ukc.ac.uk

Abstract

The safety analysis of an evolving software system has to consider the impact that changes might have on the software components, and to provide confidence that the risk is acceptable. If the impact of a change is not thoroughly analysed, accidents can occur as a result of faulty interactions between components, for example. However, the process of safety analysis can be enhanced if appropriate abstractions are provided for modelling and analysing software components and their interactions. Instead of considering components as the locus of change, the proposed approach assumes that components remain unchanged while their interactions (i.e. connectors) adapt to the different requirements needs. The safety analysis is the performed using model checking to verify whether safe behaviour is maintained when interactions between components change. The feasibility of the approach is demonstrated in terms of a case study that deals with the safety procedures associated with the launching of a sounding rocket.

1. Introduction

The causes for failures of software systems are invariably related with the failure of multiple software components, rather than with the failure of a single component. The failure of multiple components is usually associated with incorrect interactions between components, also known as interface faults according with the study conducted by Lutz [15]. This study is consistent with previous studies, which have shown that systems which have highly interactive components had proportionately more errors that less interactive subsystems [18]. In the wider context of system's engineering, causes of system accidents can also be traced to errors occurring in the interactions between the components, rather than the failure of individual components [16].

A major reason for the existence of these *interaction faults* is the inherent difficulty of extracting behavioural dependencies from the specifications of components. Moreover, if the identification of these dependencies is left for the late stages of software development then the task of performing the analysis of the safety properties of a system becomes extremely complex, hence prone to errors. The proposed approach identifies the dependencies of a component at the early phases of the software development. This provides the appropriate level of abstraction for modelling and analysing the interactions between components, in addition to the behavioural analysis of individual components, which characterises the more conventional approaches.

Evolution of software systems can be made more robust to interaction faults if the appropriate modelling abstractions are used for describing the architecture of the software. Instead of relying on the provision of means and mechanisms which focus on supporting the adaptation of components [1,4], this paper presents an approach for the evolution of software systems that 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 COTS and legacy software, that 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 modelling 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 [2]. 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 [20]. There are several design description languages, which have rich vocabulary, and that are able to describe in the form of collaborative diagrams interactions between objects in terms of messages and events [3], and to represent the implementation of components as a composition of object roles [8]. However, these object-oriented languages lack the means for describing the properties associated with object and their interactions, which should be an essential feature of architectural description languages. In this paper, we employ the modelling abstraction *co-operative action* (CO action) as an architectural entity for representing collaborative activity between objects [6]. 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 [8]. In the proposed approach the architectural interpretation of an action (i.e. connector) is obtained by focusing on the specification of the participants, and the conditions for the participants for starting, maintaining and finishing a collaborative activity.

The rest of this paper is organised as follows. In section 2, we define the architectural style that provides the appropriate abstractions for modelling and analysing the safety properties of a system in terms of interactions between its components. Section 3 discusses the benefits of the proposed architectural style when considering the analysis of safety properties in evolving systems. In section 4 we discuss the feasibility of the proposed approach in terms of a case study which consists in specifying the destruction of a sounding rocket with the purpose of maintaining the safety requirements. Finally, section 5 concludes with a discussion evaluating our contribution.

2. Co-operative Object-Oriented Style

Architectural structures for systems tend to abstract away from the details of a system, but assist in understanding broader system-level concerns [19]. This is achieved by employing architectural styles that are appropriate for describing the software components, the interactions between these components, and the properties that regulate the composition of components.

In the following, we present the co-operative object-oriented style that adopts basic features of object-orientation, in which components are represented as classes and connectors as co-operative actions, and the instantiation of these abstractions are respectively, objects and co-operations. 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.

2.1. Architectural Elements

The architectural elements of the co-operative object-oriented style are *classes* as the basic components, and *co-operative actions* (CO actions) as the basic connectors. Co-operative actions (CO actions) were introduced as entities for modelling interactions between classes that characterise collaborative behaviour [6]. The use of CO actions is motivated by their ability of extracting from the specification of a class those issues related with its collaborative activities, thus avoiding a specification of a collaboration to be scattered among classes

In the co-operative object-oriented 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. The composition of these roles defines the interface of the class.

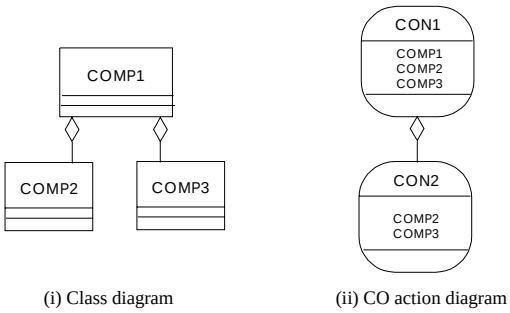
A CO action is described by a template with the following fields: the CO action's **name**, declaration of **attributes** in terms of the names and types of the **participants** of the CO action, **constants** and **variables** local to the CO action, and the specification of the collaborative **behaviour** of the classes participating in the CO action. The behaviour field includes **initial** state of the object, and the specification of the complete behaviour space of the CO action, in terms of its **normal**, **exceptional** and **failure** behaviours. 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, the **invariant** establishes the conditions that should hold while the collaborative activity is being performed, and the collaborative **operation** to be performed by the CO action. 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 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.

In addition of specifying the collaborative operation in terms of what the CO action should do, it is equally

important to specify what the CO action 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** when no services are delivered by the CO action, and failures of **commission** when the service delivered by the CO action is different from the required service. At the architectural level description of the system, is expected that both omission and commission failures are specified in terms of the hazards of the system. For the specification of exceptional behaviour, the operation is replaced by a **handler** that identifies the exception event, together with the start and finish events associated with the handler of the exception.

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 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. Only CO actions contain relational information. An advantage of this 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 restricting the impact of change.



(i) Class diagram (ii) CO action diagram

Figure. 1. Co-operative object-oriented architecture.

For describing the architecture of a software system, two different diagrams are employed: a *class diagram* describing the relationships between components, and a *CO action diagram* describing the relationships between connectors. These diagrams provide a compact representation of the software system, which can be completed with a more detailed textual description.

An example of a co-operative object-oriented architecture is shown figure 1: CO action CON1 has three participants (COMP1, COMP2 and COMP3), and the nested CON2 has two participants (COMP2 and COMP3).

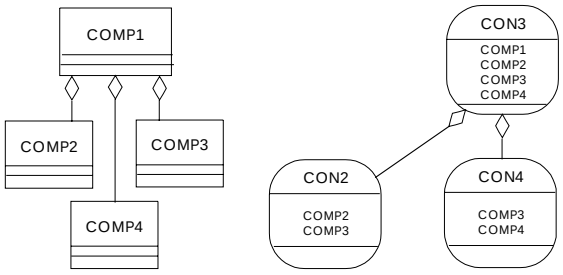
2.3. Evolving Co-operative Object-Oriented Architectures

The architecture of a software system is an important factor that affects the flexibility of software in adapting to changes. The dynamics of an evolving system is dominated by the slow changing components, which in the case of software are the large granularity abstractions that describe a software system. On the other hand, although the small granularity components are bound to change quicker, any change is certainly constrained by the slow changing components. In other words, changes at higher levels of abstraction are more likely to impact the architectural description of the system, compared with those made at the lower levels. Hence the need to focus on architectural description languages for handling change. The aim is to have languages that are able to restrict the impact of change, thus allowing localised changes to be incorporated more effectively by avoiding their propagation into the rest of the system. This is not the case, for example, in object-oriented design languages where a change in the interface or name of an object can have a great impact on the whole design of the software.

A co-operative object-oriented architecture of a software system can either evolve by changing the components (i.e. the interface of the components, or their implementation), or by changing the connectors (i.e. the roles of the components, or the collaborative operations). When a component changes, its impact is restricted to the connectors in which the component plays a role, because the system connectors, and not the components, maintain all the relational information in a co-operative object-oriented architecture. In this case, the other system components do not need to know that a component has changed. Similarly, when a connector changes, the components that play a role in the connector and the other connectors of the software architecture do not need to know about this change. The reason being that, first, there is no need to change neither the component interface nor its implementation when there is a change in one of the roles of a component, and second, there is information confinement across connectors.

The co-operative object-oriented architecture of figure 2 is an evolution of that of figure 1: the component COMP4 was added to the original architecture, and connector CON1 was replaced by CON3. The addition of a new component should not have an impact on the other components, but it is expected that connectors have to change, by modifying the roles played by the other components. From the CO action diagram, we infer that CON2 and CON4 are nested CO actions of CON3, and

that CON2 and CON4 should be mutually independent because they share a common resource (COMP3).



(i) Class diagram (ii) CO action diagram

Figure. 2. An evolving co-operative object-oriented architecture.

3. Case Study: Destruction System for the VS-40X Sounding Rocket

The purpose of sounding rockets is to carry scientific instruments on the payloads into space. Their sub-orbital flight follows a parabolic trajectory that is appropriate for performing scientific experiments. The VS-40X is a two stages sounding rocket which has a dual purpose within the Brazilian Space Programme: apart from performing scientific experiments, it will be used as an experimental platform for the new equipment of the Brazilian Satellite Launcher (VLS). In this case study, we analyse the safety procedures for the destruction of a rocket when its trajectory violates a pre-defined flight envelope. Currently, the rocket is destroyed remotely by a safety operator, but the intention is to replace it with an automatic system for its self-destruction. However, before introducing a complete new system, confidence has to be obtained that this system is as safe as the existing manual one. In order to obtain such confidence, two additional intermediate configurations were considered in which the two systems are redundant in their operation, with the self-destruction system possessing different degrees of autonomy.

The purpose of this case study is to show how effective is a co-operative object-oriented architecture when checking the stability of the safety properties of an evolving system. Instead of having to re-model and re-analyse the whole system, this paper claims that the impact of changes can be restricted to the modelling and analysis of the interactions between the system components. Whether a system component is changed or removed, or a new component added, the impact of change is scoped by the co-operations between the components.

3.1. Safety and Mission Requirements

The *safety requirements* of the VS-40X system aim to maintain the integrity of the environment[1] of the vehicle (in terms of damage to property, injuries and loss of lives) when there is a failure in the behaviour of the vehicle.

Depending on the flight phase and the flight trajectory of the vehicle, we can identify two types of *accidents*:

- during the pre-launching or initial flight instants, an unintentional destruction of the vehicle can cause damage to the launching installations, injuries, or the loss of lives;
- during the rest of the flight, the fall of debris after a failure in the behaviour of the vehicle can cause damages to property, injuries, or the loss of lives.

There are two *hazards* associated with the above accidents, which can be stated and formalised as follows:

- during the pre-launching (v.preLauncPhase) and initial flight instants (v.initialPhase), there is a destruction of the vehicle (v.destroyed);
- during the intermediate phases of the flight (v.intermediatePhase), the projection of the point of impact of the vehicle's trajectory crosses the limit line of impact into the protected region (v.insidePR);

Based on the above hazards, the *safety specifications* for the destruction system of the VS-40X system are the following:

- during the pre-launching and initial phases of the flight, the destruction of the vehicle should be disabled;
- during the intermediate phases of the flight, once the vehicle trajectory violates the safety plan the vehicle should be destroyed[2];

The above safety requirements have to be considered in the context of the *mission requirement* for the VS-40X: under normal conditions during the flight, the vehicle should not be destroyed.

3.2. Evolving Architecture for the Destruction System

In this section, we present a partial model of the sounding rocket VS-40X from the viewpoint of the safety requirements. The co-operative object-oriented architecture of the system will focus on the components

[1] In this paper we are not concerned with the integrity of the actual vehicle.

[2] There are other two scenarios in which the vehicle has to be destroyed, however, these scenarios are related with the integrity of the vehicle rather than the integrity of the environment.

and interactions responsible for the destruction of the sounding rocket.

According with the class diagram of figure 3, the four basic components of the VS-40X System are the SafetyOperator, OperatorConsole, Vehicle, and LaunchingPad. In terms of the remote destruction system, the relevant components of the OperatorConsole are: CCTV which provides the visual information of the vehicle's flight trajectory, SISGRAF which provides tracking information from the radar, and Telemetry which provides measures of the key variables that define the state of the vehicle. The relevant components of the VS-40X Vehicle are: SafetyBox which provides the protection mechanism to avoid the unintentional destruction of the vehicle during the pre-launching and initial phases of the flight, and RemoteControl which receives and processes the control commands from the OperatorConsole (it contains a self-diagnostic mechanism which detects whether has failed or not). If self-destruction is considered, two additional components have to be included to the VS-40X Vehicle: Trajectory which calculates the flight trajectory of the vehicle based on information provided by the Inertial Reference System (IRS), and the ProtectionSystem which establishes whether the flight safety plan has been violated. These two components have similar roles of those associated with the OperatorConsole and SafetyOperator, respectively.

The four diagrams of figure 3 show the evolution of the destruction system in terms of the possible configurations of the components of the VS-40X System. In the following, these configurations are defined in terms of the CO actions that define the co-operations between the components. The CO actions enable to extract from the definition of the components of the VS-40X System those activities which are related to the destruction system.

1. The destruction system is solely based on the remote destruction – the two CO actions are the EnableDestruction which describes the collaborative activity between OperatorConsole, SafetyBox and Vehicle for enabling and disabling the destruction of the vehicle, and RemoteDestruction which describes the collaborative behaviour of components of the VS-40X System for destroying the vehicle by the SafetyOperator;

2. The destruction system is based on the remote or self-destruction – in addition to the two CO actions of the previous configuration, the other two CO actions are the SelfADestruction which describes the collaborative behaviour of components of Vehicle for its self-destruction, and StateSISGRAF which maintains consistent the states of SISGRAF and RemoteControl thus providing a means for enforcing mutual exclusion between remote (RemoteDestruction) and self-destruction (SelfADestruction);

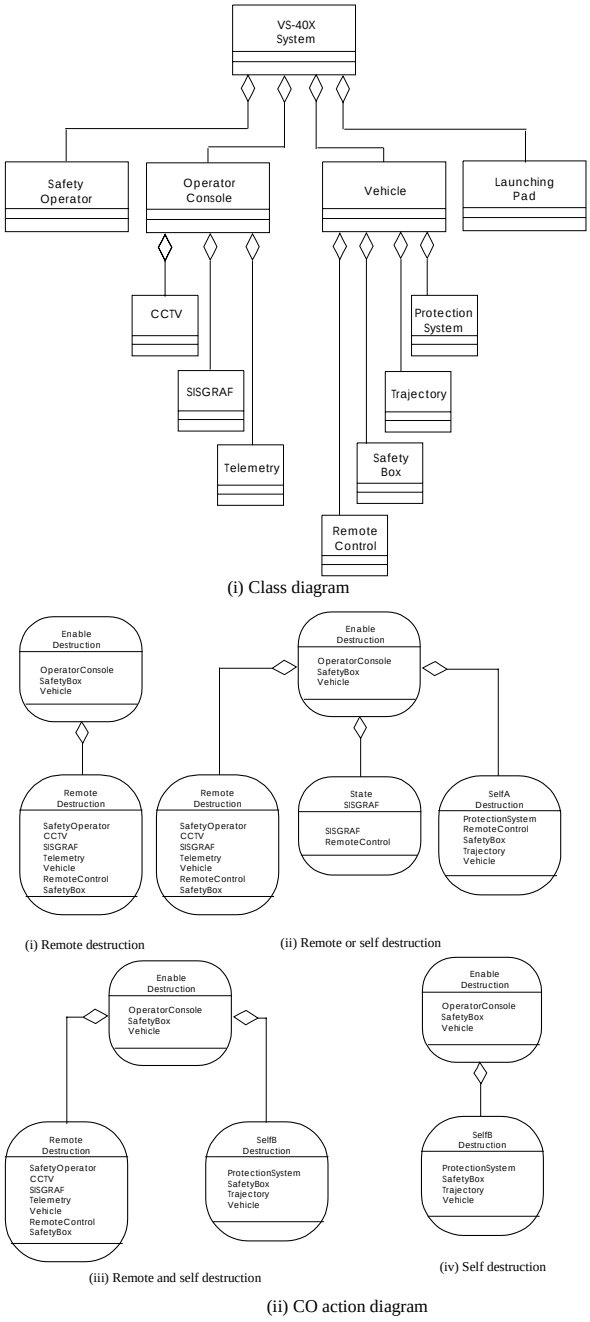


Figure. 3. Evolution of the destruction system

3. The destruction system is based on the remote and self-destruction – in addition to the two CO actions of the first configuration, CO action SelfBDestruction

is a simplify version of SelfADestruction in which there is no need for maintaining mutual exclusion between remote (RemoteDestruction) and self-destruction (SelfBDestruction);

4. The destruction system is based on the self-destruction – in addition to EnableDestruction, the other CO action is SelfBDestruction which is responsible for the self-destruction of the Vehicle.

3.3. Modelling and Analysis of Components Interactions

In this section, we formally specify the behaviour of CO actions, and analyse the safety properties associated with every CO action. The intent is to verify whether the safety behaviour of the system is maintained by the interactions between the system components. For sake of brevity, we will focus on the fourth configuration of the destruction system, although the safety analysis of the other three configurations was equally performed.

3.3.1. Modelling Interactions

The behaviour of the CO actions is formally specified in terms of Extended Real-Time Logic (ERTL) [5,9] (an outline of ERTL is presented in the Appendix - the occurrence and holding relations will be suppressed without loss of detail.

The CO action EnableDestruction specifies the collaborative activities of the components of the VS-40X System involved in enabling and disabling the destruction of the Vehicle. The normal behaviour pre-condition establishes the start of EnableDestruction at the instant that OperatorConsole sends a signal notifying that it is safe to destroy (oc.safeDest). The collaborative operations associated with EnableDestruction include: to check whether the operator console (oc.safeDest) activates the safety relay (v.sb.safeDest), and to initialise a temporisation mechanism, which activates the inhibiting relay (v.sb.enDest), after the disconnection of the umbilical (v.disUmbil). The post-condition of EnableDestruction is captured by a transition event predicate that specifies the necessary and sufficient conditions for the co-operation to end: either when the destruction of the Vehicle is disabled (v.sb.enDest), or when the Vehicle is destroyed (v.destroyed). The specification of the failure behaviour for the EnableDestruction follows directly from the hazards. There is a commission fault (commission1_enableDestruction), related to hazard_A, when the EnableDestruction is operational, Vehicle is either in pre-launching or initial phases of the flight, and the destruction of the Vehicle is enabled (v.sb.enDest). There is an omission fault (omission_enableDestruction), related to hazard_B, when during the intermediate phases of the flight, the EnableDestruction is operational and the destruction of the Vehicle is not enabled.

EnableDestruction:

attributes:

participants:

oc	*OperatorConsole*
v	*Vehicle*
v.sb	*SafetyBox*

behaviour:

initial:

$\Phi(\neg oc.safeDest \wedge \neg v.disUmbil \wedge \neg v.destroyed \wedge \neg v.sb.enDest \wedge \neg v.sb.safeDest \wedge v.preLaunchPhase \wedge \neg v.initialPhase \wedge \neg v.intermediatePhase, 0)$

normal:

pre-condition:

$\forall t\bullet \Theta(\nearrow enableDestruction, t) \Leftrightarrow \Theta(\nearrow oc.safeDest, t)$

operation:

$\exists T0: \forall t<T0\bullet \Theta(\nearrow oc.safeDest, t) \Rightarrow \Theta(\nearrow v.sb.safeDest, t)$

$\exists T0\bullet \Theta(\nearrow v.sb.safeDest, t) \wedge \Theta(\nearrow v.disUmbil, T0) \Rightarrow \Theta(\nearrow v.sb.enDest, T0+5)$

post-condition:

$\forall t\bullet \Theta(\searrow enableDestruction, t) \Leftrightarrow \Theta(\nearrow v.destroyed, t)$

failure:

$\forall t\bullet \Phi(commission\_enableDestruction, t) \Leftrightarrow \Phi(enableDestruction, t) \wedge \Phi(v.preLaunchPhase \vee v.initialPhase, t) \wedge \Phi(v.sb.enDest, t)$

$\forall t\bullet \Phi(omission\_enableDestruction, t) \Leftrightarrow \Phi(enableDestruction, t) \wedge \Phi(v.intermediatePhase, t) \wedge \Phi(\neg v.sb.enDest, t)$

The CO action SelfBDestruction specifies the collaborative activities of the components of the Vehicle for its self-destruction. The normal behaviour pre-condition establishes the start of SelfBDestruction at the instant that SafetyBox enables the destruction of the Vehicle (v.sb.enDest). The invariant and the collaborative operations associated with SelfBDestruction include: to activate the self-destruction (v.ps.actDest) when the destruction system detects that the trajectory taken by the vehicle has violated the safety plan (v.tr.outsideSP), and to destroy the vehicle once the self-destruction is activated. The post-condition states that the components of the Vehicle should leave the SelfBDestruction co-operation when the destruction of the Vehicle is disabled, or when the Vehicle is destroyed. There is a commission fault (commission1_selfBDestruction), related to hazard_A, when the SelfBDestruction is operational, Vehicle is either in pre-launching or initial phases of the flight, and the ProtectionSystem activates the destruction of the Vehicle. There is another commission fault (commission2_selfBDestruction), related to the violation of the missionRequirement, when during the intermediate phases of the flight, the SelfBDestruction is operational, the flight trajectory of the Vehicle is not outside the safety plan, but the ProtectionSystem activates the destruction. There is an omission fault

(omission_selfBDestruction), related to hazard_B, when during the intermediate phases of the flight, the SelfBDestruction is operational, the flight trajectory of the Vehicle is outside the safety plan, but the ProtectionSystem does not activate the destruction.

SelfBDestruction:

attributes:

participants:

v	*Vehicle*
v.ps	*ProtectionSystem*
v.sb	*SafetyBox*
v.tr	*Trajectory*

behaviour:

initial:

$\Phi(\neg$v.ps.actDest $\wedge$ $\neg$v.tr.outsideSP $\wedge$ $\neg$v.sb.enDest $\wedge$ $\neg$v.destroyed $\wedge$ v.preLaunchPhase $\wedge$ $\neg$v.initialPhase $\wedge$ $\neg$v.intermediatePhase, 0)

normal:

pre-condition:

$\forall t \bullet \Theta(\nearrow$selfBDestruction, t) $\Leftrightarrow \Theta(\nearrow$v.sb.enDest, t)

invariant:

$\forall t \bullet \Phi$(selfBDestruction, t) $\Leftrightarrow \Phi$(v.sb.enDest, t)

operation:

$\forall t \bullet \Phi$(selfBDestruction, t) $\wedge \Theta(\nearrow$v.tr.outsideSP, t) $\Rightarrow$ $\Theta(\nearrow$v.ps.actDest, t)

$\forall t \bullet \Phi$(selfBDestruction, t) $\wedge \Theta(\nearrow$v.ps.actDest, t) $\Rightarrow$ $\Theta(\nearrow$v.destroyed, t)

post-condition:

$\forall t \bullet \Theta(\searrow$selfBDestruction, t) $\Leftrightarrow \Theta(\nearrow$v.destroyed, t)

failure:

$\forall t \bullet \Phi$(commission1_selfBDestruction, t) $\Leftrightarrow$ Φ(selfBDestruction, t) $\wedge \Phi$(v.preLaunchPhase $\vee$ v.initialPhase, t) $\wedge \Phi$(v.ps.actDest, t)

$\forall t \bullet \Phi$(commission2_selfBDestruction, t) $\Leftrightarrow$ Φ(selfBDestruction, t) $\wedge \Phi$(v.intermediatePhase, t) $\wedge$ $\Phi(\neg$v.tr.outsideSP, t) $\wedge \Phi$(v.ps.actDest, t)

$\forall t \bullet \Phi$(omission_selfBDestruction, t) $\Leftrightarrow$ Φ(selfBDestruction, t) $\wedge \Phi$(v.intermediatePhase, t) $\wedge$ Φ(v.tr.outsideSP, t) $\wedge \Phi(\neg$v.ps.actDest, t)

3.3.2. Safety Analysis of the Interactions

The analysis of the co-operative behaviour confirms that the combined normal behaviour described in the CO actions is able to maintain the safe properties of the system, which are specified by the CO actions failure behaviour. Evidence was obtained by using model checking which is a formal verification technique based on state exploration. Given a state transition system and a property, model checking algorithms exhaustedly explore the state space to determine whether the system satisfies the property. The result is either a claim that the property is true or a counter-example in terms of a sequence of states that falsifies a property.

The model checker employed for performing the verification of the CO action behavioural specification is UPPAAL, an automated tool for the analysis of real-time systems [14]. The operational representation of the system behaviour is modelled using timed automata, which are obtained from the specification of a CO action normal behaviour, defined by ERTL formulas. This is a straightforward transformation in which the ERTL transition events that define the pre-, post-conditions and collaborative operations are associated with the transitions between the states of an automaton. The safety properties to be confirmed are obtained from the specification of failure behaviour of a CO action. The system context for conducting the safety analysis of the destruction system consists of the following automata: EnableDestruction (4 states), SelfBDestruction (4 states), FlightPhases (7 states), FlightTrajectory (2 states), and OCSafeDestruction (2 states). This particular configuration is relatively simple to be analysed using model checking.

The behavioural specification of CO actions EnableDestrcution and SelfBDestruction in terms of timed automata is shown in figure 4. The vertices of the automata represent location, and the edges represent transitions between locations, which are labelled with a guard, an assignment, or a synchronisation label. The CO action pre-conditions are represented by the outgoing arcs from location *ED0 (SBD0)*, and the post-conditions by incoming arcs to *ED0 (SBD0)*. The guards represent the conditions that allow the state of the automaton to evolve. For example, in the automaton SelfBDestruction the state of v.sb.enDest is updated by automaton EnableDestruction, and the state of v.tr.outsideSP is updated by the automaton that simulates whether the trajectory of the Vehicle is outside the safety plan (FlightTrajectory).

For the analysis of the safety properties of the destruction system, the first step was to check using UPPAAL queries whether the normal behaviour of SelfBDestruction would not violate its safe behaviour. The next step was to check whether the combined behaviour of CO actions EnableDestruction and SelfBDestruction is able to maintain the safety of the VS-40X System. The VS-40X System enters into a hazard state (associated with a commission fault) whenever the Vehicle is destroyed (`v_destroyed==1`) during the pre-launching (`FP.FP0` or `FP.FP1`) or initial phases of the flight (`FP.FP2`). This scenario can be represented in terms of UPPAAL query language the states that are to be avoided:

```
A[] not (( FP.FP0 or FP.FP1 or FP.FP2 ) and
      v_destroyed==1).
```

Using UPPAAL model checking capabilities we have confirmed that the above property is not violated for the specifications of CO actions EnableDestruction and SelfBDestruction. We have also confirmed confirm that the VS-40X System is safe for those hazards associated with omission faults. When the vehicle is in the intermediate phases of the flight (`FP.FP3` or `FP.FP4`), it

is outside the safety plan (`v_tr_outsideSP==1`) but the vehicle is never destroyed (`v_destroyed==0`):

```
E<> not (( FP.FP3 or FP.FP4 ) and
      v_tr_outsideSP==1 and v_destroyed==0).
```

As a result we can conclude that the VS-40X System is safe for those hazards associated with commission and omission faults, and that the mission requirement is not violated despite the safety mechanism:

```
E<> not (( FP.FP3 or FP.FP4 ) and
       v_tr_outsideSP==0 and v_ps_actDest==1).
```

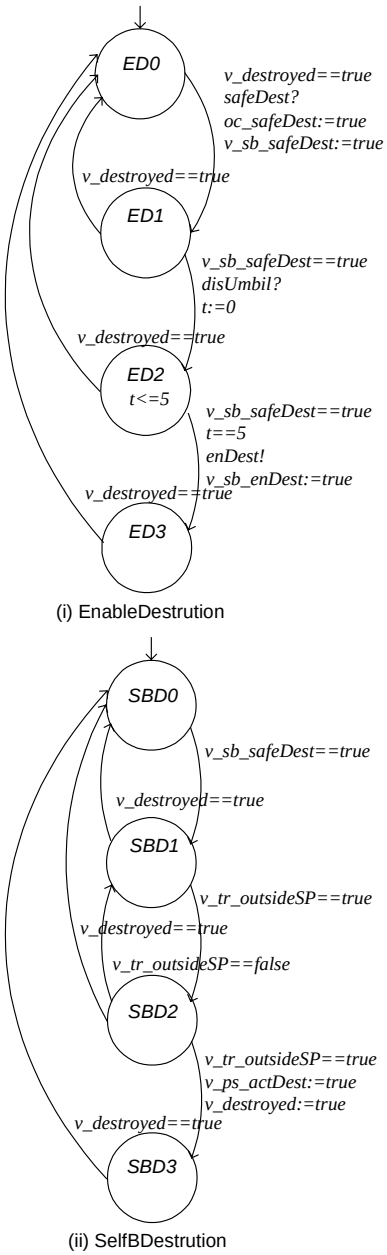


Figure. 4. Hybrid automata of EnableDestruction and SelfBDestruction.

A similar safety analysis exercise using laborious deductive and inductive analysis techniques was performed of the same destruction system [13]. The purpose of this exercise was not so much to compare different techniques, but instead was to obtain diverse arguments in the provision of evidence that the safety properties of the system are not violated. The combination of a co-operative object-oriented architecture and model checking has shown effective when dealing with systems that undergo constant change. However, caution must be taken over the (false) confidence that can be obtained when employing model checking [7].

Using a co-operative object-oriented architecture for representing the destruction system of the sounding rocket has proven to be effective way for performing the safety analysis because of the few changes that needed to be made between the different configurations. The basic building blocks of the models used by the model checking and the deductive and inductive techniques were connectors of the architecture that incorporated the changes between the different configurations. For that, we had to assume that components remained unchanged between the configurations, and that all the changes could be implemented by modifying the interactions between the components. This approach seems appropriate for systems that contain components that are amenable to change.

4. Conclusions

This paper has presented an approach for checking the stability of the safety properties of evolving software in which the safety analysis is enhanced by extracting from the definition of the system components the behavioural dependencies associated with their interactions. The proposed approach is different from existing approaches that rely solely on the behavioural specification of system components for obtaining confidence that the system safety will be maintained whenever there is a change.

The basic claim of this paper is that the co-operative object-oriented style can enhance the safety analysis of evolving software systems. Compared with other approaches the major difference of the proposed approach is that, connectors in addition of being the place of communication, they are also the place of state and computation: they encapsulate roles of the components and collaborative operations between the components. Instead of having to spread change among a group of interacting components, a co-operative object-oriented architecture allows change to be localised in its connectors. Assuming the roles played by the components are mutually independent, the impact of change can be restricted because all the relational information is associated with the connectors, rather then the components. The feasibility of the proposed approach was demonstrated through the specification and verification of the destruction system of a sounding rocket. While keeping the components unchanged, the destruction

system has evolved by changing the interactions between the components. Instead of having to consider the whole system, the modelling and analysis of the safety properties have focused on the interactions between the components.

Acknowledgements

I would like to thank the staff from the Aeronautics and Space Institute/Aerospace Technical Centre (IAE/CTA) in Brazil for their support in defining the case study, and to acknowledge the financial support from CAPES/Brazil, BC/UK and EPSRC/UK ADAPT Project.

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

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

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

[5] 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.

[6] 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.

[7] R. de Lemos, A. Saeed. "Validating Formal Verification using Safety Analysis Techniques". *Proceedings of the 18th International Conference on Computer Safety, Reliability and Security (SAFECOMP'99)*. Toulouse, France. September, 1999. pp. 58-66.

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

[9] 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.

[10] F. Jahanian, A. Mok. "Safety Analysis of Timing Properties in Real-Time Systems". *IEEE Transactions on Software Engineering Vol. SE-12(9)*. September 1986. pp. 890-904.

[11] F. Jahanian, A. Mok, D. A. Stuart. *Formal Specifications of Real-Time Systems*. Technical Report TR-88-25, Department of Computer Science, University of Texas at Austin, TX. June 1988.

[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] C. H. N. Lahoz, et al. *Safety Analysis of the Sounding Rocket VS-40X Destruction System*. IAE/CTA Technical Report. May 2000. (In Portuguese)

[14] K. G. Larsen, P. Peterson, and Wang Yi. "UPPAAL in a Nutshel". *International Journal on Software Tools for Technology Transfer 1(1-2)*. October 1997. pp. 134-152

[15] R. R. Lutz. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems". *Proceedings of the IEEE International Symposium on Requirements Engineering*. San Diego, CA. January 1993. IEEE Computer Society Press. pp. 126-133.

[16] C. Perrow. *Normal Accidents: Living with High-Risk Technology*. Basic Books Inc. New York, NY. 1984.

[17] R. W. Selby, V. R. Basili. "Analysing Error-Prone System Structure". *IEEE Transaction on Software Engineering Vol. SE-17(2)*. February 1991. pp. 141-152.

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

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

Appendix - Extended Real Time Logic (ERTL)

The basis for Extended Real Time Logic (ERTL) is the event-action model that provides a set of primitive concepts for the modelling and analysis of phenomena associated with the computer system and its environment. In the event-action model, an *event* serves as a temporal marker, an *action* is an operation which consumes a bounded quantity of resources, and a *system predicate* is an assertion about a system variable at a time point.

Extended Real Time Logic (ERTL) [5,9] is a first order predicate logic for the modelling and analysis of hybrid systems, taking as a basis Jahanian & Mok's Real Time Logic (RTL) [10,11]. RTL uses uninterpreted predicates to relate events of a system to the time of their occurrence, thereby providing the means for reasoning about absolute timing properties of real-time systems. The extensions provided by ERTL allow reasoning about system behaviour in both value and time domains through predicates defined in terms of system variables.

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

$$\forall t \bullet \forall i \in P: \Theta(\text{Motor\_On}, i, t)$$

The ith occurrence of event MotorOn has occurred at time t.

A *transition event* is defined by the transition of a system predicate from false to true, or from true to false, at a particular time point. For a system predicate P, the respective transition events are $\nearrow P$ and $\searrow P$.

$$\forall t \bullet \forall i \in P: \Theta(\searrow \text{plateOnBeg}, i, t) \Leftrightarrow \Theta(\nearrow(\text{plateOnEnd} \wedge \neg \text{beltOn}), i, t)$$

The transition event which captures the instant which of the predicate plateOnBeg becomes false is equivalent to the transition event which captures the instant that the conjunction of plateOnBeg and the negation of beltOn becomes true.

The *holding relation* (Φ) captures whether a system predicate holds true at a time point. $\Phi(f, i, t)$ defines that a formula f holds for the ith time, at time t.

$$\forall t \bullet \forall i \in P: \Phi(\text{moveDown}, i, t) \Leftrightarrow \Phi(\neg \text{bottom} \wedge \neg \text{plateOn}, i, t)$$

The predicate moveDown holds true *iff* the conjunction of the negating predicates bottom and plateOn also holds true.

For simplicity, the unindexed versions of the occurrence and holding relations have also been defined.