List of Authors

Jean Arlat ..... LAAS-CNRS, Toulouse, F

Jean-Charles Fabre..... LAAS-CNRS, Toulouse, F

Valérie Issarny ..... INRIA, Rocquencourt, F

Cliff Jones ..... University of Newcastle upon Tyne, UK

Nicole Levy..... INRIA, Rocquencourt, F

Eric Marsden..... LAAS-CNRS, Toulouse, F

Panos Periorellis ..... University of Newcastle upon Tyne, UK

Manuel Rodriguez..... LAAS-CNRS, Toulouse, F

Alexander Romanovsky ..... University of Newcastle upon Tyne, UK

Ferda Tartanoglu..... INRIA, Rocquencourt, F

Ian Welch..... University of Newcastle upon Tyne, UK

Table of Contents

Further Results on Architectures and Dependability Mechanisms for Dependable SoSs

Jean Arlat[1], Jean-Charles Fabre[1], Valérie Issarny[2], Cliff Jones[3], Nicole Levy[2], Eric Marsden[1], Panos Periorellis[3], Manuel Rodriguez[1], Alexander Romanovsky[3], Ferda Tartanoglu[2], Ian Welch[3]

[1]LAAS-CNRS (Toulouse, F), [2]INRIA (Rocquencourt, F), [3]University of Newcastle upon Tyne (UK)

Introduction

This document reports recent results of work package on architecture and design related to architecture descriptions and design of dependability mechanisms for dependable systems of systems (SoSs), and to systems of systems integration. These results contribute further in the achieving the ultimate work package aim of delivering the definition of an environment for the construction of dependable SoSs out of autonomous heterogeneous systems and dependability mechanisms assisting in building such systems.

The deliverable extends further mechanisms and techniques presented in Project deliverable IC2 on *Initial Results on Architectures and Dependability Mechanisms for Dependable SoSs*. Development of the Conceptual Model (cf deliverable IC1: *Revised Version of the Conceptual Model*) has had a strong influence on the advances in work package on SoS architecture and design which are reported here.

The document is organised into three chapters and an Appendix. These chapters may be read independently, and corresponding bibliographical references are given separately at the end of the report. The first chapter proposes a structured approach to integrating complex Web applications that are built using existing Web services as the component systems. The next part of the deliverable puts forward a general framework for dealing with on-line upgrades of component systems at the level of linking interfaces (LIFs). The last chapter introduces an approach to detecting component system errors at the level of LIFs and to recovering after them, and demonstrates this approach using a real-time microkernel as a component system. A brief overview of the chapter contents is as follows:

- **Dependable Composition of Web Services:** This chapter introduces Project work towards supporting the development of dependable SoSs in the context of the Web Service Architecture. Our approach primarily lies in developing WSCAL (Web Service Composition Action Language) – an XML-based language for the abstract specification of the dependable composition of Web Services, which builds upon the CA Actions concept

for enforcing dependability. We further introduce base design of middleware support for the automatic generation of composite Web Services from their WSCAL specification. An Internet Travel Agency is used as a case study to demonstrate the approach proposed.

- **Structured Handling of On-Line Interface Upgrades in Dependable SoSs**: There are many practical situations in which the interfaces of the component systems are changed dynamically and without notification. In this chapter we propose an approach to handling such online upgrades in a structured and disciplined fashion. All interface changes are viewed as abnormal events and general fault tolerance mechanisms (exception handling, in particular) are applied to dealing with them. The chapter outlines general ways of detecting such interface upgrades and recovering after them. An Internet Travel Agency is used as a case study throughout the chapter.

- **From error detection to recovery wrappers**: this chapter discusses the notion of recovery action and how it fits within the framework developed in the previous work on error detection wrapping. In particular, it demonstrates how the wrapping framework proposed can be applied to real-time microkernel based systems. The case study discussed is based on a real-time application running on a COTS real-time microkernel. The chapter presents some of the results derived from the fault injection experiments and concludes with a discussion of the notion of wrapping within a CORBA system according to experimental results.

Appendix A reports recent Project work on B formalisation of CA actions and of their composition. This work gives a precise definition of CA actions and allows for rigorous reasoning about dependable behavior of SoSs integrated using this structuring paradigm.

Chapter 1 - Dependable Composition of Web Services

Ferda Tartanoglu, Valérie Issarny, Nicole Levy *(INRIA)*, Alexander Romanovsky *(University of Newcastle)*

1.1. Introduction

Systems that build upon the Web Service architecture[1] are expected to become a major class of wide-area SoSs in the near future due to the architecture support for integrating applications over the Web, which makes it particularly attractive for the development of multi-party business processes. More specifically, the Web Service architecture targets the development of applications based on the XML standard [W3C-XML], hence easing the development of distributed systems by enabling the dynamic integration of applications distributed over the Internet, independently of their underlying platforms. Currently, the main constituents of the Web Service architecture are as following:

(i) WSDL (Web Services Description Language) that is a language based on XML for describing the interfaces of Web Services [W3C-WSDL].

(ii) WSCL (Web Services Conversation Language) that is a language for specifying business-level conversations supported by Web Services [W3C-WSCL].

(iii) SOAP (Simple Object Access Protocol) that defines a lightweight protocol for information exchange [W3C-SOAP]; SOAP sets the rules of how to encode data in XML, and also includes conventions for partly prescribing the invocation semantics (either synchronous or asynchronous) as well as the SOAP mapping to HTTP.

The Web Service architecture is further conveniently complemented by UDDI (Universal Description, Discovery and Integration) that allows specification of a registry for dynamically locating and advertising Web Services [UDDI].

There already exist various platforms that are compliant with the Web Service architecture, including .NET [MS-NET] and J2EE [SUN-J2EE]. In addition, integration within CORBA is being addressed [OMG-WS]. Even though the Web Service architecture is quite recent and not fully mature, it is anticipated that it will play a prominent role in the development of the next generation distributed systems mainly due to the strong support from industry and the huge effort in this area. However, there clearly is a number of research challenges in supporting the thorough development of distributed systems based on Web Services. One such challenge relates to the effective usage of Web Services in developing business processes, which requires support for composing Web Services in a way that guarantees dependability of the resulting composed services. This calls for developing new architectural principles of building such composed systems, in general, and for studying specialized connectors "glueing" Web Services, in particular, so that the resulting composition can deal with failures occurring at the

[1] http://www.w3.org/2002/ws.

level of the individual service components by allowing co-operative failure handling at the level of the composed systems.

Solutions that are being investigated towards the above goal subdivide into: (i) the definition of XML-based languages for the specification of Web Services composition, and (ii) revisiting classical transactional support so as to cope with the specifics of Web Services (e.g., crossing administrative domains, Web latency), i.e., defining connectors offering transactional properties over the Internet. Section 1.2 gives an overview of existing solutions to the two aforementioned issues, and assesses them with respect to Web Service composition and its dependability. In particular, it is emphasized that while the transaction concept offers a powerful abstraction to deal with the occurrence of failures in closed systems, it imposes too strong constraints over component systems in an open environment such as the Web. The main constraint imposed by transactions relates to supporting backward error recovery that, firstly, requires isolating component systems for the duration of the embedded (nested) transaction in which they get involved and hence contradicts the intrinsic autonomy of Web Services, and, secondly, relies on returning the service state back, which is not applicable in many real-life situations which involve documents, goods, money as well as humans (clients, operators, managers, etc.).

In the light of the above, this chapter puts forward a solution based on forward error recovery, which enables dealing with dependability of composed Web Services, and has no impact on the autonomy of the individual Web Services, while exploiting their possible support for dependability (e.g., transaction support at the level of each service). Our solution, presented in Section 1.3, lies in system structuring in terms of co-operative actions that have a well-defined behavior, both in the absence and in the presence of service failures. More specifically, we define the notion of Web Service Composition Action (WSCA) that builds upon the Coordinated Atomic Action concept developed at the University of Newcastle, which allows structuring composite Web Services in terms of dependable actions. Sections 1.4 and 1.5 then introduce a framework enabling the development of composite Web Services based on WSCAs, subdividing into an XML-based language for the specification of WSCAs and a platform supporting the execution of WSCAs. Finally, Section 1.6 discusses our current and future work aimed at enhancing the Web Service architecture with respect to dependability.

1.2. Background

Offering solutions to the dependable composition of Web Services has triggered a number of research projects over the last couple of years. Ongoing effort may be subdivided into two complementary lines of work: (i) offering languages for the abstract specification of Web Services and their composition so as to support the thorough design, analysis and construction of composite Web Services, (ii) offering transactional support for composite Web Services so as to enforce well-defined properties over composed Web Services in the presence of failures. The two following sections give an overview of proposed solutions in the two above areas.

1.2.1. Specifying the Composition of Web Services

Composing Web services relates to dealing with the assembly of autonomous components so as to deliver a new service out of the components' primitive services, given the corresponding published interfaces. In the current Web Service architecture, interfaces are described in WSDL

and published through UDDI. However, supporting composition requires further addressing: (i) the specification of the composition, and (ii) ensuring that the services are composed in a way that guarantees the consistency of both the individual services and the overall composition. This calls for the abstract specification of Web Services and of their composition that allows reasoning about the correctness of interactions with individual Web Services, which requires adhering to the interaction patterns assumed by the Web Service's implementation. In addition, the specification of Web Services should allow for automating as much as possible the analysis of specification as well as the implementation of service composition in most environments, including providing support for enhanced service delivery in terms of offered non-functional properties. Finally, it should be possible to reuse composed services in diverse environments, and in particular to dynamically select the component Web Services according to the specific environment in which a composed service is invoked. This puts forward the need for a high-level specification language of Web Services that is solely based on the components of the Web Service architecture and that is as much as possible declarative. Defining a language based on XML then appears as the base design choice for specifying the composition of Web Services.

A first requirement for the XML-based specification of Web Services is to enforce correct interaction patterns among services. This lies in the specification of the conversations that are assumed by the Web Service's implementation for the actual delivery of advertised services. In other words, the specification of any Web Service must define the observable behavior of a service and the rules for interacting with the service in the form of exchanged messages. Such a facility is supported by a number of XML-based languages: WSCL from W3C [W3C-WSCL], WSFL [WSFL], Microsoft's XLANG [XLANG], and Web Service Choreography Interface (WSCI) [WSCI]. While WSCL is the current W3C standard for specifying conversations associated with Web Services in the context of the Web Services architecture, the other languages mainly differ from WSCL by offering additional capabilities for specifying a sequence of actions over services in a way similar to a workflow schema in the case of WSFL, and for specifying the external behavior of services with respect to failure occurrences in the case of XLANG and WSCI, as further addressed in the next section.

Given the specification of conversations associated with individual Web Services, the composition (also referred to as integration or aggregation) of Web Services may be specified as a graph (or process schema) over the set of Web Services, where the interactions with any one of them must conform to the conversations associated with them. The specification of such a graph may then be: (i) automatically inferred from the specification of individual services as addressed in [Narayanan & MvIllraith 2002], (ii) distributed over the specification of the component Web Services as in the XL language [Florescu *et al*. 2002], or (iii) be given separately as undertaken in [WFSL], [BPML], [Casati *et al.* 2001], [Fauvet *et al.* 2001], and [Yang & Papazoglou 2002]. The first approach is quite attractive but restricts the composition patterns that may be applied, and cannot thus be applied in general. The second approach is the most general, introducing an XML-based programming language. However, this makes more complex the reuse of composed Web Services in various environments since this requires retrieving the specification of all the component Web Services prior to deploy the composed service in a given environment. On the other hand, the last approach supports quite directly the dynamic deployment of a composed service from its specification by clearly distinguishing the specification of component Web Services (comprising primitive components that are considered as block-box components and/or inner composite components) from the specification of

composition. Then, by providing an XML-based language for specifying the composition of Web Services, the provided specification can serve automating the actual service composition at run-time, including support for the dynamic selection of component services through the exploitation of UDDI. Execution of the composed service may then be realized by a centralized service provider or through peer-to-peer interactions [Benatallah *et al.* 2002]. The latter approach that is in particular supported by the Self-Serv platform [Fauvet *et al.* 2001] is a priori more scalable due to its decentralized nature. However, this requires installing specific components on the sites hosting the participating component Web Services, which cannot be enforced in general. In addition, scalability issues in the case of centralized execution of the composite service provider raise only if the number of composed services is quite high, which is not expected to be the common case. Centralized execution of the composed service further raises availability problem, which is not much a concern since it can be quite easily solved through replication of the service provider.

In the light of the above, the development of composed Web Services can adequately be supported through the provision of:

- An XML-based declarative language for the abstract specification of component Web Services, including the definition of both the service's interfaces and the observable behavior of the service in the form of message exchanges. Such a language is already offered by the Web Service architecture through WSDL and WSCL.

- An XML-based language for specifying the composition process, which defines the graph of interactions among Web Services. A specified composition process needs not fix the instances of component Web Services that are actually composed upon invocation of the embedding service. They may be dynamically selected at runtime according to the specific invocation environment, exploiting in particular UDDI functionalities. The actual implementation of the service composition may further be decoupled from the specification of the composition process, leaving under the responsibility of the developer to check that his implementation conforms to the service's specification. Alternatively, the implementation may be generated from the specification. The latter approach is the most promising although raising the challenge of defining an XML composition language that is powerful enough to meet requirements of most composite services, while being abstract enough for supporting thorough analysis.

- Tools for automatically checking the consistency of Web Services composition with respect to the specification of component services. Static checking should here be promoted as much as possible. In this context, the correctness of a composed service with respect to the WSCL specification of its component services may be addressed through model checking. However, consistency checking at runtime is still necessary when selecting dynamically the component services instances. This further calls for advertising Web Services with their WSCL specification in addition to the WSDL specification, so as to implement the consistency check during the selection of service instances instead of deferring it during the invocation of services as addressed in [Kuno *et al.* 2001].

In addition to the above requirements, the composition of Web Services must promote the dependability of resulting Web Services. Existing work in the area primarily relies on transactional support as discussed in the next section.

1.2.2. Transactions for the Dependable Composition of Web Services

Transactions have been proven successful in enforcing dependability in closed distributed systems. The base transactional model that is the most used guarantees ACID (atomicity, consistency, isolation, durability) properties over computations. Enforcing ACID properties typically requires introducing protocols for: (i) locking resources (i.e., two-phase locking) that are accessed for the duration of the embedding transaction, and (ii) committing transactions (i.e., two or three phases validation protocols). However, such a model is not suited for making the composition of Web Services transactional for at least two reasons:

- The management of transactions that are distributed over Web Services requires cooperation among the transactional support of individual Web Services –if any-, which may not be compliant with each other and may not be willing to do so given their intrinsic autonomy and the fact that they span different administrative domains.

- Locking accessed resources (i.e., the Web Service itself in the most general case) until the termination of the embedding transaction is not applicable to Web Services, still due to their autonomy, and also the fact that they potentially have a large number of concurrent clients that will not stand extensive delays.

Enhanced transactional models may be considered to alleviate the latter shortcoming. In particular, the split model (also referred to as open-nested transactions) where transactions may split into a number of concurrent sub-transactions that can commit independently allows reducing the latency due to locking. Typically, sub-transactions are matched to the transactions already supported by Web Services (e.g., transactional booking offered by a service) and hence transactions over composed services do not alter the access latency as offered by the individual services. Enforcing the atomicity property over a transaction that has been split into a number of sub-transactions then requires using compensation over committed sub-transactions in the case of sub-transaction abortion. Using compensation comes along with the specification of compensating operations supported by Web Services for all the operations they offer. Such an issue is in particular addressed by XLANG [XLANG] and WSCI [WSCI]. However, it should be further accounted that using compensation for aborting distributed transactions must extend to all the participating Web Services (i.e., cascading compensation by analogy with cascading abort), which is not addressed by XLANG nor WSCI due to their focus on the behavioral specification of individual Web Services for assisting their composition. An approach that accounts for the specification of the transactional behaviour of Web Services from the standpoint of the client in addition to the one of the service is proposed in [Mikalsen *et al.* 2002]. This reference introduces a middleware whose API may be exploited by Web Services' clients for specifying and executing a (open-nested) transaction over a set of Web Services whose termination is dictated by the outcomes of the transactional operations invoked on the individual services.

The aforementioned references concentrate on the specification of the transactional behaviour of Web Services. Complementary work is undertaken in the area of transaction protocols

supporting the deployment of transactions over the Web, while not imposing long-lived locks over Web resources. Existing solutions include THP (Transaction Hold Protocol) from W3C [W3C-THP] and BTP from OASIS [OASIS-BTP]. The former introduces the notion of tentative locks over Web resources, which may be shared among a set of clients. A tentative lock is then invalidated if the associated Web resource gets acquired. The BTP protocol introduces the notion of cohesion, which allows defining non-ACID transactions by not requiring successful termination of all the transaction's actions for committing.

Developing transactional support for dependable Web Services is an active area of research that is still in its infancy. From our point of view, solutions to the dependable composition of Web Services that use primarily transactions do not cope with all the specifics of Web Services. A major source of penalty lies in the use of backward error recovery in an open system such as the Internet, which is mainly oriented towards tolerating hardware faults but poorly suited to the deployment of cooperation-based mechanisms over autonomous component systems that often require cooperative application-level exception handling among component systems. An alternative then lies in relying on the existing support of Web Services for managing internal concurrency control, possibly including transactional support, so as to guarantee keeping the consistency of services, while relying on forward error recovery for ensuring the dependability of service composition. The next section introduces such a solution, which builds upon the concept of Coordinated Atomic (CA) Actions [Xu *et al.* 1995].

1.3. Web Service Composition Actions

The CA Actions [Xu *et al.* 1995] are a structuring mechanism for developing dependable concurrent systems through the generalization of the concepts of atomic actions and transactions. Basically, atomic actions are used for controlling cooperative concurrency among a set of participating processes and for realizing coordinated forward error recovery using exception handling, and transactions are used for maintaining the coherency of shared external resources that are competitively accessed by concurrent actions (either CA Actions or not). Then, a CA Action realizes an atomic state transition where:

1. The initial state is defined by the initial state S_{Pi} of the participant processes *Pi* and the states S_{Rj} of the external resources *Rj* at the time they were accessed by the CA Action.

2. The final state is defined by the state of the participant proceesses (S_{Pi}') at the action's termination (either standard or exceptional) and the state of the accessed external resources (S_{Rj}' in the case of either standard termination or exceptional termination without abortion, S_{Rj} in the case of exceptional termination with abortion).

CA Action naturally fits the specification of operations provided by composite Web Services:

- A participant (process) specifies interactions with each composed Web Service, stating the role of the specific Web Service in the composition. In particular, the participant specifies actions to be undertaken when the Web Service signals an exception, which may be either handled locally to the participant or be propagated to the level of the embedding CA Action. The latter then leads to co-operative exception handling according to the exceptional specification of the CA Action.

- Each Web service is viewed an external resource. However, unlike the base CA Action model, interactions do not have to be transactional. The interactions adhere to the semantics of the Web Service operations that are invoked. An interaction may then be transactional if the given operation that is called is. However, transactions do not span multiple interactions.

- The standard specification of the CA Action gives the expected behavior of the given operation of the composed Web Service in either the absence of failures or in the presence of failures that are locally handled (i.e., either system-level exceptions or programmed exceptions signaled by Web Services operations that do not need to be cooperatively handled at the CA Action level).

- The exceptional specification of the CA Action states the behavior of the given operation of the composed Web Service under the occurrence of failure at one or more of the participants, that need cooperative exception handling. The resulting forward recovery may then realize a relaxed form of atomicity (i.e., even when individual operations of the Web service are transactional, its intermediate states may be accessed by external actions between such operations executed within a given action) when Web Services offer both transactional and compensating operations (to be used in cooperative handling of exceptions).

The above application of CA Actions to the context of Web Services composition, leads us to introduce the concept of WSCA (Web Service Composition Action). WSCAs mainly differ from CA Actions in relaxing the transactional requirements over external resources (which are not suitable for wide-area open systems) and the introduction of dynamic nesting of CA Actions (i.e., nested calls of WSCAs). The interested reader is referred to Appendix A for the formal specification of CA Actions extended with dynamic nesting (referred as "action composition" in Appendix A), while it is part of our future work to provide a formal specification of WSCAs. We further do not exploit static nesting of CA Actions, which may be realized through dynamic nesting given the relaxed form of atomicity of WSCAs.

In order to illustrate the use of WSCAs for specifying the composition of Web Services, we take the Travel Agent case study. We consider joint booking of accommodation and flights using respective hotel and airline Web Services. Then, the composed Web Service's operation is specified using WSCAs as follows. The top-level WSCA *TravelAgent* comprises the *User* and the *Travel* participants; the former serves interacting with the user while the latter achieves joint booking according to the user's request through call to the WSCA that composes the *Airline* and the *Hotel* participants[2]. A diagrammatic specification of the WSCAs is shown in Figure 1.

[2] Note that these participants are not necessarily bound to a unique Web service, they may interact with various Web services satisfying the target interface, which may be located through UDDI.

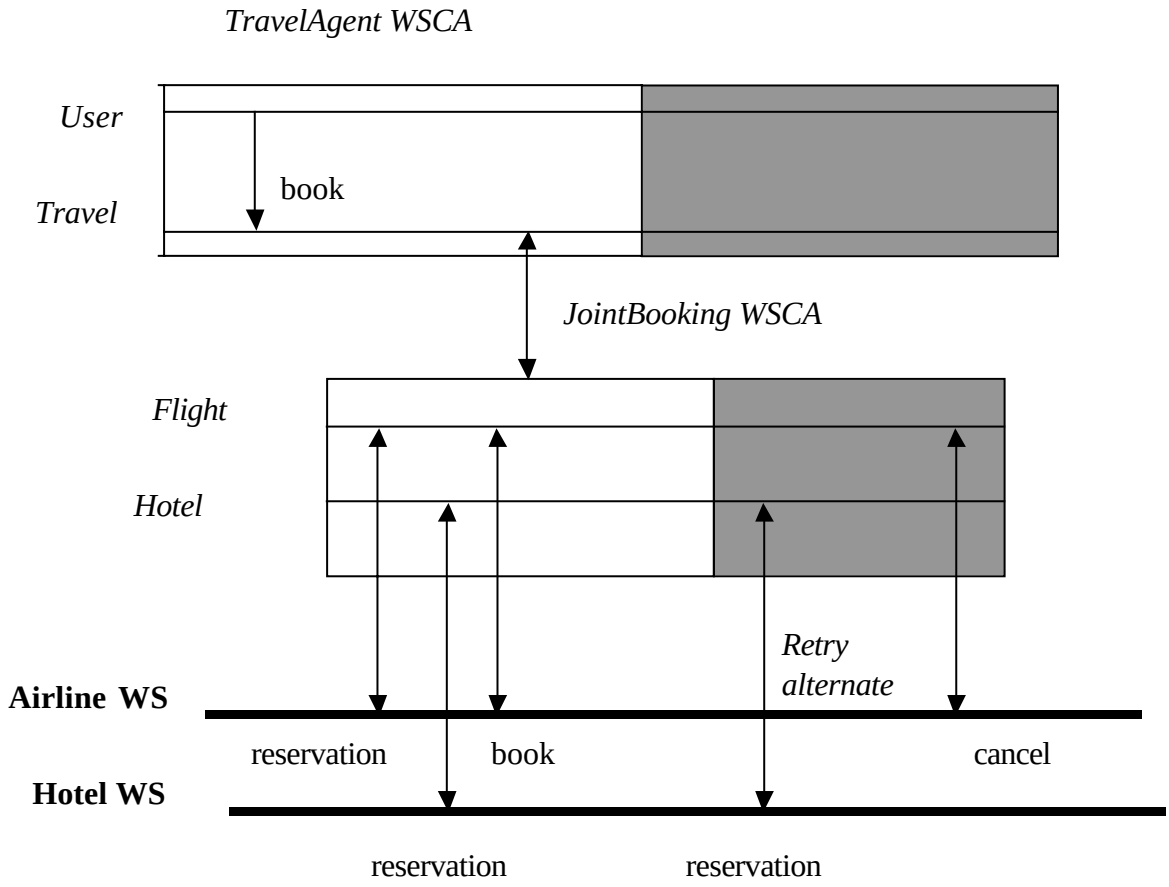


Figure 1 WSCA for composing Web Services

In *TravelAgent*, the *User* participant requests the *Travel* participant to book a return ticket and a hotel room for the duration of the given stay. This leads the *Travel* participant to invoke the *JointBooking* WSCA that composes the *Hotel* Web Service and the *Airline* Web Service. The participants of the *JointBooking* WSCA respectively requests for a hotel room and a return ticket, given the departure and return dates provided by the user. Each request is subdivided into reservation for the given period and subsequent booking if the reservation succeeds[3]. In the case where either the reservation or the booking fails, the participant raises the *unavailable* exception that is cooperatively handled at the level of the *JointBooking* WSCA denoted by the greyed box in the figure. If both participants signal the *unavailable* exception, then *Travel*

[3] Such a workflow process is certainly not the most common since the user is in general requested for confirmation prior to booking. However, this scenario that applies most certainly to in-hurry-not-bother users enables concise illustration of the various recovery schemes that are supported.

signals the *abort* exception so that the exception gets handled by *TravelAgent* in a cooperation with the User (e.g., by choosing an alternative date). If only one participant raises the *unavailable* exception, cooperative exception handling includes an attempt by the other participant to find an alternative booking. If this retry fails, the booking that has succeeded is cancelled and the abort exception is signaled to the calling *TravelAgent* WSCA for recovery with user intervention.

Compared to the solutions that introduce transactional supports for composed Web Services, ours mainly differs in that it exploits forward error recovery at the composition level, as well as transactional supports offered by individual Web Services – if available. Hence, the underlying protocol for interaction among Web Services remains the one of the Web Service Architecture (i.e., SOAP) and does not need to be complemented with a distributed transaction protocol. Similarly to our solution, the one of [Mikalsen *et al.* 2002] does not require any new protocol to support distributed open-nested transactions. An open-nested transaction is declared on the client side by grouping transactions of the individual Web Services, through call to a dedicated function of the middleware running on the client. The transaction then gets aborted by the middleware using compensation operations offered by the Web Services, according to conditions set by the client over the outcomes of the grouped transactions. Our solution is then more general since we allow for the specification of forward error recovery at the composition level, enabling in particular to integrate non-transactional Web Services while still enforcing dependability at the composition level. The next section further introduces the WSCAL XML-based language for the specification of composite Web Services based on WSCAs, which allow generating corresponding implementation of dependable composite Web Services, as discussed in Section 1.5.

1.4. WSCAL for the Abstract Specification of Dependable Web Services Composition

From the WSCA definition given in the previous section, the specification of a WSCA-based composite Web Service subdivides into the specification of:

- The abstract interface of the composite Web Service, which is given in terms of WSDL and WSCL specification, as for any Web Service.
- Participants binding to the composed Web Services. The Web Service instance associated to a given participant may be either statically set or dynamically retrieved according to the service's abstract specification. In addition, for the sake of availability, we allow a participant to be bound to a set of Web Service instances implementing the service's specification. Any composed Web Service is characterized by associated WSDL and WSCL documents, where the WSDL document includes concrete binding information only in the case of static binding. As raised in the previous section, transactional support of individual services is exploited when available. The abstract interface of any Web Service is thus further characterized by the service's transactional behavior, in a way similar to existing solutions in the area (e.g., [Mikalsen *et al.* 2002], [WSCI]).

- WSCAs defining the operations provided by the composite Web Service. The definition of a WSCA specifies the standard and exceptional behavior of the WSCA's participants, including cooperative exception handling.

- The exception resolution tree that serves resolving the exceptions that are concurrently raised within WSCAs into a single exception, as supported by CA Actions [Xu *et al.* 1995].

The following section defines WSCAL (WSCA Language) that is the XML-based language that is introduced for the specification of WSCA-based composite Web Services. It is then followed by an example of composite service specification, still in the context of the Travel Agent case study.

1.4.1. WSCAL

The following XML schema defines WSCAL[4], introducing the embedded XML elements (i.e., **`WS`** for the abstract definition of service interfaces and **`WSC`** for the abstract specification of composite services) where the reader is assumed to be familiar with the XML schema language[5]:

```
<xsd:schema xsd:id=``wscal''
          xmlns:xsd=``http://www.w3.org/2001/XMLSchema"
          xsd:targetNamespace=``http://www-rocq.inria.fr/arles/2002/WSCAL"
          xsd:elementFormDefault=``qualified''>
      <xsd:element name=``Definition''>
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref=``WS'' minOccurs=``0''/>
                        <xsd:element ref=``WSC'' minOccurs=``0''/>
                  </xsd:sequence>
                  <xsd:attribute name=``name'' type=``xsd:string''/>
                  <xsd:attribute name=``targetNamespace'' type=``xsd:anyURI''/>
            </xsd:complexType>
       </xsd:element>
       ...
</xsd:schema>
```

Specifying service interfaces

The interface of a Web Service is characterized by the messages exchanged with the Web Service (as given by the associated WSDL document) and the protocol of interactions assumed by the service (as given by the associated WSCL document). The service interface is further enriched with the characterization of the service's transactional behavior. More precisely, interfaces of services are abstractly defined using the following **`WS`** element.

---

[4] Note that we omit the possible definition of comments.

[5] http://www.w3.org/XML/Schema.

```
<xsd:element name=``WS''>
     <xsd:complexType>
          <xsd:sequence>
               <xsd:element name=``Interface'' type=``XMLDocumentType''/>
               <xsd:element name=``Conversation''
                    type=``XMLDocumentType''
                    minOccurs=``0''/>
               <xsd:element ref=``Transactional''

                    minOccurs=``0'' maxOccurs=``unbounded''/>
          </xsd:sequence>
          <xsd:attribute name=``name'' type=``xsd:string'' use=``required''/>
     </xsd:complexType>
</xsd:element>

<xsd:complexType name=``XMLDocumentType''>
     <xsd:attribute name=``hrefSchema'' type=``xsd:anyURI''/>
</xsd:complexType>
```

Where the embedded **`Interface`** element gives the URI of the related WSDL document (limited to the abstract part) and the **`Conversation`** element gives the URI of the related WSCL document. The latter is optional although it is advisable to provide it for more rigorous specification of services, allowing in particular enforcing a consistent interaction protocol with the service. Note that the abstract definition of the interfaces of the Web Service operations given in a WSDL document includes the exceptions that may be signalled by the operations through the **`wsdl:fault`** element [W3C-WSDL]. The optional **`Transactional`** element further serves specifying the transactional behaviour of the Web Service. In a first step, we consider only support for open-nested transactions through compensation, which we view as the most common in the context of Web Services. The **`Transactional`** element thus defines a transactional operation (**`operation`** attribute) whose execution can be compensated, together with the corresponding compensentation operation (**`compensate`** attribute), both operations being defined in the WSDL document associated with the service:

```
<xsd:element name=``Transactional''>
     <xsd:complexType>
              <xsd:attribute name=``name'' type=``xsd:string''/>
              <xsd:attribute name=``operation'' type=``xsd:string''/>
              <xsd:attribute name=``compensate'' type=``xsd:string''/>
     </xsd:complexType>
</xsd:element>
```

Note that we assume that executing the compensation operation for an operation Op leads to cancelling the effect of executing Op (considering though that the effect of executing Op may have been externally observed before the compensate took place). We realise that this cannot be assumed in general (e.g., cancelling booking may lead the consumer to pay penalty fees) and it is part of our future work to extend WSCAL for the precise specification of the transactional behaviour of Web Services. However, we do not see this as a major issue in the light of existing work in the area (see Section 1.2.2 for references).

Specifying composite services

The specification of a composite Web Service gives:

- The service's interface through the **`Abstract`** element that refers to the corresponding `WS` element defined in the given WSCAL document.

- The exception resolution tree defined by the **ExceptionTree** element that refers to the corresponding XML document.
- The Web Services that are composed as defined by the **Participants** element.
- The behavior of the supported operations, which are WSCAs, as defined by the **WSCA** element.

We get the following definition for the **WSC** element:

```
<xsd:element name=``WSC''>
     <xsd:complexType>
          <xsd:sequence>
             <xsd:element name=``Abstract'' type=``XMLDocumentType''/>
             <xsd:element name=``ExceptionTree'' type=``XMLDocumentType''/>
             <xsd:element ref=``Participants'' />
             <xsd:element ref=``WSCA'' maxOccurs=``unbounded''/>
          </xsd:sequence>
          <xsd:attribute name=``name'' type=``xsd:string'' use=``required''/>
     </xsd:complexType>
</xsd:element>
```

The definition of the **Participants** element amounts to specifying the Web Services that are composed. Each such Web Service is defined using the **Participant** element and is partly characterized by the **Service** element that refers to the WSCAL document defining the corresponding WS element. In addition, each participant may be statically bound to a specific service instance (as defined by the **StaticParticipant** element) and/or dynamically bound to an instance matching the abstract definition of the service's interface that is given by the corresponding service attribute (as defined by the **DynamicParticipant** element). In the former case, concrete binding information is provided through the WSDL document associated with the service's instance (i.e., **Instance** element), which must match the definition of the service's abstract interface (i.e., the abstract parts of the respective WSDL documents match, which is currently defined as syntactic matching). In the latter case, a matching service instance is located at runtime using a location service such as a UDDI service instance[6]. Dynamic binding of participants with associated Web Services may take place either upon invocation of the service's WSCAs or upon instantiation of the composite Web Service, according to the value of the **onCall** Boolean attribute of the given participant. Notice that in the case of dynamic binding at call-time and nested calls of WSCAs, the Web Service is dynamically located once, upon the invocation of the top-most WSCA that first involves the corresponding participant. Finally, we allow each participant to be bound with a set of instances matching the specification of associated service rather than a single instance for the sake of availability; this is specified using the **multiple** boolean attribute in the DynamicParticipant element and by stating as many instances as required in the

[6] Ideally, the location service must allow retrieving an instance that matches the WSCAL specification of the service's interface (i.e., matching the WSDL, WSCL and transactional definitions) and not just the WSDL abstract part, requiring location services handling WSCAL and related documents. Alternatively, base UDDI services may be used for locating Web Services matching the provided WSDL specification. This issue is further discussed in Section 2.5.

StaticParticipant element. We get the following definition for the **Participants** element:

```
<xsd:element name=``Participants''>
     <xsd:complexType>
          <xsd:sequence>
               <xsd:element ref=``Participant'' maxOccurs=``unbounded''/>
          </xsd:sequence>
     </xsd:complexType>
</xsd:element>

<xsd:element name=``Participant''>
     <xsd:complexType>
          <xsd:sequence>
               <xsd:element name=``Service'' type=``XMLDocumentType''/>
               <xsd:element ref=``StaticParticipant''
                    minOccurs=``0'' maxOccurs=``unbounded''/>
               <xsd:element ref=``DynamicParticipant''
                    minOccurs=``0'' />
          </xsd:sequence>
          <xsd:attribute name=``name'' type=``xsd:string'' use=``required''/>
     </xsd:complexType>
</xsd:element>

<xsd:element name=``StaticParticipant''>
     <xsd:complexType>
          <xsd:attribute name=``Instance''
               type=``XMLDocumentType'' maxOccurs=``unbounded''/>
     </xsd:complexType>
</xsd:element>

<xsd:element name=``DynamicParticipant''>
     <xsd:complexType>
          <xsd:attribute name=``onCall'' type=``xsd:boolean'' default=``true''/>
          <xsd:attribute name=``multiple''
               type=``xsd:boolean'' default=``false''/>
     </xsd:complexType>
</xsd:element>
```

The definition of a WSCA specifies the behavior of each of its participants, which are bound with participants of the composite service. The WSCA participant behaviour is defined as a process, through classical statements, in a way similar to existing XML-based language for specifying the composition of Web Services (see Section 1.2.1 for references). The specifics of WSCAL comes from structuring the operations provided by composite Web Services as WSCAs[7] that coordinate the execution of Web Services operations with respect to failure occurrences, in particular introducing the specification of coordinated exception handling. More precisely, the definition of the **WSCA** element embeds a sequence of **ParticipantWSCA** elements, each specifying the behaviour of a participant, and the **operation** attribute that gives the name of the operation of the embedding composite Web Service that is being specified among the one given in the associated WS definition:

```
<xsd:element name=``WSCA''>
     <xsd:complexType>
          <xsd:sequence>
               <xsd:element ref=``ParticipantWSCA''
                    maxOccurs=``unbounded''/>
          </xsd:sequence>
          <xsd:attribute name=``name'' type=``xsd:string''/>
```

[7] Note that a WSCA with one participant defines a classical process involving a single Web Service.

```
            <xsd:attribute name=``operation'' type=``xsd:string''/>
      </xsd:complexType>
</xsd:element>
```

The specification of any WSCA participant amounts to defining:

- The participant of the embedding composite Web Service to which the WSCA participant is bound through the **`bind`** attribute that gives the name of the participant among those defined in the `Participants` element of the embedding `WSC` element. Note that the attribute is optional since a participant may actually be introduced for processing results of nested WSCAs as illustrated by the *Travel* participant of the *TravelAgent* WSCA introduced in Section 1.3.

- Parts of the messages associated with the WSCA that are relevant to the specific participant, which is defined using the **`Input`**, **`Output`** and **`Fault`** elements. The two first elements are subset of the corresponding elements within the definition of the operation implemented by the WSCA that is given in the associated WSDL document. In addition, the union of the **`Output`** elements defined in the WSCA's participants must be equal to the `Output` element defining the result of the related operation. Finally, the **`Fault`** elements define the exceptions that may be raised by the participants, which require cooperative exception handling and get composed with the exceptions concurrently raised by peer participants using the `exceptionTree` attribute.

- The local state of the specific participant through the **`State`** element that defines the local variables.

- The behaviour of the specific participant using the **`Behavior`** element. The participant behavior subdivides into the participant's standard (as defined by the **`Standard`** element) and exceptional (as defined by the **`Exceptional`** elements) behavior. Each such behavior is defined as a process using classical statements, including in particular interaction with the Web Service instance(s) associated with the participant, message exchanges with peer participants and exception handling, as further detailed hereafter. The exceptional behavior of the participant actually defines the handlers associated with the exceptions that need coordinated exception handling, as identified using the `exceptionTree` element and the `fault` elements associated with the WSCA's participants. The specific exception that is being handled by a given handler is identified by the **`handles`** attribute, which must name an exception of the subtree of the `ExceptionTree` element that encompasses all the exceptions raised by the WSCA's participants.

We get the following definition for the **`ParticipantWSCA`** element:

```
<xsd:element name=``ParticipantWSCA''>
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name=``Input'' type=``paramType''
                        minOccurs=``0'' />
                  <xsd:element name=``Output'' type=``paramType''
                        minOccurs=``0'' />
                  <xsd:element name=``Fault'' type=``faultType''
                        minOccurs=``0'' maxOccurs=``unbounded''/>
                  <xsd:element ref=``State'' minOccurs=``0'' />
                  <xsd:element ref=``Behavior''/>
```

```
                </xsd:sequence>
                <xsd:attribute name=``name'' type=``xsd:string'' use=``required''/>
                <xsd:attribute name=``bind'' type=``xsd:string'' minOccurs=``0''/>
        </xsd:complexType>
</xsd:element>


<xsd:complexType name=``paramType''>
        <xsd:attribute name=``name'' type=``xsd:NMTOKEN'' use=``optional''/>
        <xsd:attribute name=``message'' type=``xsd:QName'' use=``required''/>
</xsd:complexType>
<xsd:complexType name=``faultType''>
        <xsd:attribute name=``name'' type=``xsd:NMTOKEN'' use=``required''/>
        <xsd:attribute name=``message'' type=``xsd:QName'' use=``required''/>
</xsd:complexType>

<xsd:element name=``State''>
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name=``internal'' type=``typeType''
                                minOccurs=0 maxOccurs=``unbounded''/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:complexType name=``typeType''>
        <xsd:attribute name=``name'' type=``xsd:NMTOKEN'' use=``optional''/>
        <xsd:attribute name=``type'' type=``xsd:QName'' use=``optional''/>
        <xsd:attribute name=``element'' type=``xsd:QName'' use=``optional''/>
</xsd:complexType>

<xsd:element name=``Behavior''>
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref=``Standard''/>
                        <xsd:element ref=``Exceptional''maxOccurs=``unbounded''/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name=``Standard''>
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref=``State'' minOccurs=``0'' />
                        <xsd:element ref=``Body''/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name=``Exceptional''>
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref=``State'' minOccurs=``0'' />
                        <xsd:element ref=``Body''/>
                </xsd:sequence>
                <xsd:attribute name=``handles'' type=``xsd:string'' use=``required''/>
        </xsd:complexType>
</xsd:element>
<xsd:element name=``Body'' type=``statement''/>

<xsd:complexType name=``statement''>
        <xsd:sequence>
                <xsd:choice maxOccurs=unbounded/>
                        <xsd:element name=``Assign'' type=``assign''/>
                        <xsd:element name=``Sequence'' type=``sequence''/>
                        <xsd:element name=``Par'' type=``par''/>
                        <xsd:element name=``Choice'' type=``choice''/>
                        <xsd:element name=``Iteration'' type=``while''/>
                        <xsd:element name=``Call'' type=``call''/>
                        <xsd:element name=``Returns'' type=``return''/>
                        <xsd:element name=``Send'' type=``send''/>
                        <xsd:element name=``OnInput'' type=``onInput''/>
```

```
                    <xsd:element name=``Wait'' type=``wait''/>
                    <xsd:element name=``Raise'' type=``raise''/>
                    <xsd:element name=``Try'' type=``try''/>

            </xsd:choice>
      </xsd:sequence>
</xsd:complexType>
```

WSCAL statements are quite similar to the ones introduced by XML-based languages for the specification of composite services (see Section 1.2.1 for references). We more specifically base the definition of WSCAL on the CSP language for the base statements, providing a sound basis towards reasoning about WSCAL specifications. We thus detail here only the statements that are specific to Web Services composition, i.e., handling of interactions with participants and Web Services, and of exceptions (element given in bold face in the above definition of `Statement`).

The **`Call`** statement allows specifying (synchronous) operation calls where the invoked operation may be either local to the embedding composite Web Service (i.e., local WSCA) or provided by the Web Service to which the participant is bound[8] (which may be a WSCA if the service is itself composite). The **`Returns`** statement is the dual statement allowing specifying the message to be returned as partial result of the embedded operations, which is to be merged with the results returned by peer participants. We get the following definition:

```
<xsd:complexType name=``call''>
      <xsd:attribute name=``service'' type=``boolean'' default=``false'' />
      <xsd:attribute name=``operation'' type=``xsd:QName'' use=``required'' />
      <xsd:attribute name=``input'' type=``xsd:QName'' minOccurs=0 />
      <xsd:attribute name=``output'' type=``xsd:QName'' minOccurs=0 />
</xsd:complexType>
<xsd:complexType name=``return''>
      <xsd:attribute name=``element'' type=``xsd:QName'' minOccurs=0 />
</xsd:complexType>
```

The Send statement allows specifying the sending of a message to a peer participant or Web Service[9] whose dual reception may be expressed using either the blocking **`Wait`** or the non-blocking **`onInput`** statement. We get:

```
<xsd:complexType name=``send''>
      <xsd:attribute name=``recipient'' type=``xsd:QName'' minOccurs=0 />
      <xsd:attribute name=``message'' type=``xsd:QName'' use=``required'' />
</xsd:complexType>
<xsd:complexType name=``wait''>
      <xsd:attribute name=``sender'' type=``xsd:QName'' minOccurs=0 />
      <xsd:attribute name=``message'' type=``xsd:QName'' use=``required'' />
</xsd:complexType>
<xsd:complexType name=``onInput''>
      <xsd:attribute name=``sender'' type=``xsd:QName'' minOccurs=0 />
      <xsd:attribute name=``message'' type=``xsd:QName'' use=``required'' />
```

---

[8] Note that when the participant is actually bound to a set of service instances, the call is by default interpreted as a multicast RPC returning a sequence of output messages. Extension of WSCAL to specify call to specific instances could further be integrated, although not addressed here.

[9] Note that when the participant is actually bound to a set of service instances, the sending of a message to the Web Service is by default interpreted as a multicast. Extension of WSCAL to specify the sending of the message to specific instances could further be integrated, although not addressed here

```
</xsd:complexType>
```

Finally, the **Raise** statement allows signalling an exception whose handling is specified using the traditional **Try** statement for defining exception handling scopes. The participant ultimately raises the exception if it is not handled locally, leading to coordinated exception handling, as supported by WSCAs. We get:

```
<xsd:complexType name=``raise''>
       <xsd:attribute name=``exception'' type=``xsd:QName'' use=``required''/>
</xsd:complexType>
<xsd:complexType name=``try''>
       <xsd:element ref=``Body'' use=``required'' >
       <xsd:element ref=``Exceptional'' maxOccurs=``unbounded''/>
</xsd:complexType>
```

Discussion

The specification of composite Web Services using WSCAL allows carrying out a number of analyses with respect to the correctness and the dependable behavior of composite services.

Except classical static type checking, the correctness of the composite service may be checked statically with respect to the usage of individual services: the pattern of interactions with a Web Service of any WSCA participant must conform with the WSCL specification of the Web Service. Conformance with the WSCL specification may further be automated through model checking, using e.g., CSP and associated FDR tool where the translation of WSCAL processes into CSP is quite straightforward given the definition of WSCAL. In general, powerful behavioral analyses of composite services may be achieved through translation of the WSCAL specifications into CSP.

Reasoning about the dependable behavior of composite Web Services lies in the precise characterization of the dependability properties that hold over the states of the individual Web Services after the execution of WSCAs. We are in particular interested in the specification of properties relating to the relaxed form of atomicity that is introduced by the exploitation of open-nested transactions within WSCA. Toward this goal, Appendix A provides the formal specification of base CA Actions extended with dynamic nesting, which will serve as a base ground for the formal specification of WSCAs.

1.4.2. Example

This section illustrates the specification of a composite Web Service using WSCAL, through the Travel Agent-related WSCAs that were introduced in Section 1.3, focusing more specifically on the specification of the *JointBooking* WSCA. We further do not give the specification of related WSDL and WSCL documents since samples may be found in the literature (see Section 1.2.1 for references) due to the common usage of the Travel Agent case study for illustrating Web Services.

The WCSA specification of interfaces of the *TravelAgent* composite Web Service and of the *Hotel* and *Airline* Web Services is given below. In particular, the two Web Services that are composed support open-nested transactions for the *reservation* and *book* operations, through the compensating *cancel* operations (see embedded definition of **Transactional**).

```
<WS name=``TravelAgentInterface''>
```

```
        <Interface hrefSchema=``http://travelagency.com/TravelAgent.wsdl''/>
        <Conversation hrefSchema=``http:// travelagency.com/TravelAgent.wscl''/>
</WS>

<WS name=``HotelInterface''>
        <Interface hrefSchema=``http://travelagency.com/Hotel.wsdl''/>
        <Conversation hrefSchema=``http:// travelagency.com/Hotel.wscl''/>
        <Transactional name=``Reservation''
                operation=``reservation'' compensate=``cancel''/>
        <Transactional name=``booking''
                operation=``book'' compensate=``cancel''/>
        Other transactional operations
</WS>

<WS name=``AirlineInterface''>
        <Interface hrefSchema=``http://travelagency.com/Airline.wsdl''/>
        <Conversation hrefSchema=``http:// travelagency.com/Airline.wscl''/>
        <Transactional name=``reservation''
                operation=``reservation'' compensate=``cancel''/>
        <Transactional name=``booking''
                operation=``booking'' compensate=``cancel''/>
        Other transactional operations
</WS>
```

A sample of the WSC element specifying the behavior of the `TravelAgent` composite service is given below, which directly follows from the informal presentation of Section 1.3 and WSCAL definition. The service in particular offers the *JointBooking* WSCA that coordinates booking over the *Hotel* and *Airline* Web Services, for which a single instance is dynamically retrieved upon invocation of the WSCA (see definition of **`Participants`**). Coordinated booking is achieved as discussed in Section 1.3, exploiting in particular open-nested transactions of participating Web Services. The two participants of the *JointBooking* WSCA have similar behavior. The participant first invokes the **`reservation`** operation of the Web Service to which it is bound and then books the proposed selection –if any- through call to **`book`**. Otherwise, the **`unavailable`** exception is raised by the `reservation` operation, leading to retry an alternative reservation, and ultimately propagating the **`unavailable`** exception for co-operative handling at the level of the WSCA. Finally, cooperative handling of **`unavailable`** by the WSCA amounts to cancel performed booking by the peer participant -if any.

```
<WSC name=``TravelAgentService''>
     <Abstract hrefSchema=``http://travelagency.com/TravelAgentInterface.wscal''/>
     <ExceptionTree hrefSchema=``http://travelagency.com/TAExceptionTree.xml''/>

     <Participants>
       <Participant name=``UserBrowser''>
              <Service hrefSchema==``http://travelagency.com/TAUser.wscal''/>
              <DynamicParticipant>
       </Participant>
       <Participant name=``AirlineService''>
              <Service hrefSchema==``http://travelagency.com/Airline.wscal''/>
              <DynamicParticipant>
       </Participant>
       <Participant name=``HotelService''>
              <Service hrefSchema==``http://travelagency.com/Hotel.wscal''/>
              <DynamicParticipant>
       </Participant>
     </Participants>

     <WSCA name=``TravelAgent'' operation=``TravelAgent''>
       Not detailed
     </WSCA>

     <WSCA name=``JointBooking'' operation=``HABooking''>
       <ParticipantWSCA name=``airline'' bind=``AirlineService''>
```

```
            <Input …/>
            <Output …/>
            <Fault name=``unavailable'' message=``unavailableMsg''/>
            <Behavior>
                  Not detailed, similar to hotel participant given below
            </Behavior>
      </ParticipantWSCA>
      <ParticipantWSCA name=``hotel'' bind=``HotelService''>
            <Input …/>
            <Output …/>
            <Fault name=``unavailable'' message=``unavailableMsg''/>
            <State>
                  Not detailed
            </State>
            <Behavior>
                  <Standard>
                        <Body>
                        <Try>
                        <Body>
                              <Comment text=``Try reserve a room matching the
                                    user's input''/>
                              <Call service=``true''
                                    operation=``reservation''
                                    input = ``…''
                                    output= ``…''/>
                              <Comment text=``book the room that was found,
                                    as given by the output message of the call
                                    to reservation''/>
                              <Call service=``true''
                                    operation=``book''
                                    input = ``…''
                                    output= ``…''/>
                              <Comment text=``Return booking information
                                    obtained as result to the invocation of
                                    booking''/>
                              <Return element=``…''/>
                        </Body>
                        <Exceptional handles=``unavailable''>
                              <Body>
                                    Retry booking, propagates unavailable
                                    otherwise
                              </Body>
                        </Exceptional>
                        </Try>
                        </Body>
                  </Standard>
                  <Exceptional handles=``unavailable''>
                        <Body>
                              <Choice>
                                    <cond=``reserved or booked''/>
                                    <Body>
                                          <Comment text=``compensate action''>
                                          <Call service=``true''
                                                operation=``cancel''
                                                input = ``…''
                                                output= ``…''/>
                                          <Raise exception=``unavailable''/>
                                    </Body>
                              </Choice>
                        </Body>
                  </Exceptional>
            </Behavior>
      </ParticipantWSCA>
    </WSCA>
    …
</WSC>
```

1.5. Execution of WSCA-based Services

As discussed in Section 1.2.1, the execution of a composite Web Service may be realized either in a centralized way or through peer-to-peer interactions. We undertake the former approach, as it does not require any additional support from the Web Services that are composed. A composite Web Service matching a given WSCAL specification may then be either implemented by the developer or generated from the specification, depending on the specific environment in which the service is to be deployed and the complexity of the service. The complexity of the service in particular comes from internal state management in the case of a stateful service, which is abstracted in WSCAL due to the focus on Web Services interactions. In the context of our work, we are more specifically interested in the generation of stateless composite services from WSCAL specification, including the integration of adequate support for increased quality of service. This section concentrates on the design of base support for generating composite Web Services, while it is part of our future work to further develop runtime support for enhanced quality of service. Prior to introduce the design of composite Web Services generation, we first recall base runtime support associated with Web Services.

1.5.1. Base Runtime Support for Web Services

The essential role of the base runtime support (referred to as middleware in the following) for Web Services is to deploy and undeploy services, and to manage the messages and Remote Procedure Calls (RPCs) to (i.e., the calls) and from (i.e., the replies to calls) its deployed services.

The *deploy operation* provided by the middleware makes available a local program implementation to all software running on network-connected nodes, and associates a universal identifier (composed by the local host name followed by a name for the service) used by remote clients as a reference to connect to the service and request its functionality. The *undeploy operation* offered by the middleware is the opposite functionality; it deletes the mapping between the universal identifier and the service implementation, making this one unavailable to its clients.

The middleware is based on SOAP-RPC with HTTP as binding protocol. This specification requires a container able to receive requests and send replies using the HTTP protocol. Thus, the middleware must include the implementation of the HTTP protocol to allow messages exchange between deployed services and their clients. Moreover, the SOAP-RPC specification defines XML as the protocol to code data sent inside HTTP messages. Thus, the middleware must also include an XML-parser to translate received messages, execute object method calls, build replies for clients, and possibly generate fault messages.

As already mentioned, there already exist various platforms supporting the above functionalities. Considering for instance Java-based platforms, the following software may be used:

- The Java 2 Standard Edition (J2SE)[10].

[10] http://java.sun.com/j2se.

- The Java Web Service Developer Pack (Java WSDP) that provides an implementation of SOAP specification, and all Web Services-related technologies and support to handle XML documents[11].

1.5.2. Generating Web Services from WSCAL specification

Given base middleware for running Web Services, supporting the generation of composite Web Services from WSCAL specification amounts to:

- Generating the service implementation to be run over the local runtime support.
- Enriching the base middleware for Web Services so as to support the functionalities introduced by WSCAL, i.e., dynamic binding with Web Services and WSCAs.

We are more specifically concentrating on the generation of Java-based services, although our solution applies as well to other platforms.

Generating Java implementation of a composite service from WSCA specification is quite direct. First, Java stub and a skeleton associated with the invocation of Web Services' operations may be generated from the corresponding WSDL specification, using existing tools such as the *WSDL2Java* compiler. Then, the translation of WSCAL statements into Java relies on dedicated runtime support for WSCAs and dynamic binding with Web Services, the former functionality being detailed in the next section.

Support for dynamic bindings amounts to offering a registry-based location Web Service for advertising and locating Web Services. Such a facility is already supported by UDDI. However, in our context, instances of Web Services should be retrieved with respect to the WSCAL specification of the services' interfaces, i.e., taking into account the WSCL specification and transactional behavior of the service. We are thus designing extension to UDDI-based services for offering a lookup operation that implement specification matching with respect to WSCAL specification of service interfaces. We are more specifically interested in defining a behavioral specification matching relationship that allows retrieving any service instance whose behavior refines the one of the target service from the standpoint of supported conversation and transactional behaviour. Alternatively, Web Services instances may be retrieved from their WSDL specification only, leading to assume that such services do not offer transactional operations. In addition, correctness of the interactions with the Web Service with respect to conversations assumed by the service's implementation cannot be enforced, possibly leading the Web Service to raise exceptions to its caller (i.e., WSCA participant in our context).

Finally, note that by advertising the WSCA specification of a composite Web Service through location services, service instances may be deployed in any environment integrating the above support for service generation.

---

[11] http://java.sun.com/webservices.

1.5.3. Java-based Runtime Support for WSCAs

There is a number of Java and Ada implementations of CA actions developed for different platforms, environments and applications[12]. Each of them typically offers a set of re-usable classes or patterns for the application programmers to apply while employing CA actions and a runtime support built on the top of a language runtime (sometimes combined with a distributed communication feature - e.g. with RMI for some Java implementations). A complete RMI Java framework was developed several years ago [Zorzo & Stroud 1999] and since then it has been applied in a number of industry-oriented case studies. It offers a number of classes (for defining actions, action participants, exception handlers) and a runtime support in a form of the action manager object. Recently it has been used for a preliminary experimental work on implementing a prototype TA system [Romanovsky *et al.* 2002]. In the course of this work it was extended to allow for a special type of action composition based on action participant forking/joining (which we extensively use in WSCA). A Java-based local runtime support for WSCA is under development now. It is built as an adaptation of this extended CA action Java framework. The task is simplified by the fact that we do not need distribution of action participants for WSCA, so all features related to RMI can be replaced in this framework by direct method calls with multiple threads executing on a single machine taking part in an action. The resulting product is a set of Java classes that can be used either while generating the Java application code or manually by a programmer.

1.6. Conclusion

Web Services are expected to become a major class of systems of systems in the near future. This chapter has introduced our work towards supporting the development of DSoSs in the context of the Web Service Architecture. Our approach primarily lies in the WSCAL XML-based language for the abstract specification of the dependable composition of Web Services, which builds upon the CA Actions concept for enforcing dependability. We have further introduced base design of middleware support for the automatic generation of composite Web Services from their WSCAL specification. Our current and future work is oriented towards the following complementary areas:

- Formal specification of WSCAL for enabling thorough reasoning about the behavior of composite Web Services regarding both the correctness of the composition and offered dependability properties. We are in particular aiming at offering associated tool support for automated analysis of the composite Web Services' behavior.
- Detailed design and implementation of base middleware support for the generation of composite Web Services from WSCAL specification. We are in particular investigating the development of a service for locating Web Services that implements a behavioral specification matching relationship based on refinement.

[12] http://www.cs.ncl.ac.uk/old/people/alexander.romanovsky/home.formal/caa.html

- Design and implementation of middleware support for increasing the quality of composite Web Services. We are in particular interesting in developing caching support that has been proven successful in the Web for enhancing response time.

As discussed in Section 1.2, there is extensive research work that is ongoing towards supporting the development of dependable composite Web Services, addressing the XML-based abstract specification of Web Services and of their composition, and transactional support for composite Web Services. Our contribution primarily comes from relying on forward error recovery instead of backward error recovery for specifying the behavior of composite Web Services in the presence of failures. Forward error recovery is further specified in terms of co-operative actions, building upon the CA Actions concept.

Chapter 2 - Structured Handling of On-Line Interface Upgrades in Integrating Dependable SoSs

Cliff Jones, Panos Periorellis, Alexander Romanovsky, Ian Welch *(University of Newcastle)*

The integration of complex systems out of existing systems is an active area of research and development. There are many practical situations in which the interfaces of the component systems, for example belonging to separate organisations, are changed dynamically and without notification. Usually SoSs developers deal with such situations off-line causing considerable downtime and undermining the quality of the service that SoSs are delivering [Romanovsky & Smith 2002]. In this chapter we propose an approach to handling such upgrades in a structured and disciplined fashion. All interface changes are viewed as abnormal events and general fault tolerance mechanisms (exception handling, in particular) are applied to dealing with them. The chapter outlines general ways of detecting such interface upgrades and recovering after them. An Internet Travel Agency is used as a case study throughout the chapter. An implementation demonstrating how the general approach proposed can be applied for dealing with some of the possible interface upgrades within this case study is discussed.

2.1. Introduction

A "System of Systems" (SoS) is built by interfacing to systems which might be under the control of organisations totally separate from that commissioning the overall SoS. (We will refer to the existing (separate) systems as "components" although this must not confuse the question of their separate ownership). In this situation, it is unrealistic to assume that all changes to the interfaces of such components will be notified. In fact, in many interesting cases, the organisation responsible for the components may not be aware of (all of) the systems using its component. One of the most challenging problems faced by researchers and developers constructing *dependable* systems of systems (DSoSs) is, therefore, dealing with on-line (or unanticipated) upgrades of component systems in a way which does not interrupt the availability of the overall SoS.

It is useful to contrast evolutionary (unanticipated) upgrades with the case where changes are programmed (anticipated). In the spirit of other work on dependable systems, the approach taken here is to catch as many changes as possible with exception handling mechanisms.

Dependable systems of systems are made up of loosely coupled, autonomous component systems whose owners may not be aware of the fact that their system is involved in a bigger system. The components can change without giving any warning (in some application areas, e.g. web services, this is a normal situation). The drivers for on-line software upgrading are well known: correcting bugs, improving (non-) functionality (e.g. improving performance, replacing an algorithm with a faster one), adding new features, and reacting to changes in the environment.

This chapter focuses on evolutionary changes that are typical in complex web applications which are built out of existing web services; we aim to propose a generally applicable approach. As a concrete example, we consider an Internet Travel Agency (TA) [Periorellis & Dobson

2001] case study (see Figure 2). The goal of the case study is to build a travel service that allows a client to book whole journeys without having to use multiple web services each of which only allows the client to book some component of a trip (e.g. a hotel room, a car, a flight). To achieve this we are developing fault tolerance techniques that can be used to build such emergent services that provide a service which none of its component systems are capable of delivering individually. Of course, the multiplicity of airlines, hotel chains etc. provides redundancy which makes it possible for a well-designed error-recovery mechanism to survive temporary or permanent interruptions of connection but the interest here is on surviving unanticipated interface changes. As not all the systems in our system of systems are owned by the same organisation, it is inevitable that they will change during the lifetime of the system and there is no guarantee that existing clients of those systems will be notified of the change.

When a component is upgraded without correct reconfiguration or upgrading of the enclosing system, problems similar to ones caused by faults occur, for example: loss of money, TA service failures, deterioration of the quality of TA service, misuse of component systems. Changes to components can occur at both the structural and semantic level. For example changes of a component system can result in a revision of the units in which parameters are measured (e.g. from Francs to Euro), in the number of parameters expected by an operation (e.g. when an airline introduces a new type of service), in the sequence of information to be exchanged between the TA and a component system (e.g. after upgrading a hotel booking server requires that a credit card number is introduced before the booking starts). In the extreme, components might cease to exist and new components must be accommodated.

Although some on-line upgrading schemes assume that interfaces of components always stay unchanged (e.g. [Tai *et al.* 2002]), we believe that in many application areas it is very likely that component interfaces will change and that this will happen without information being sent to all the users/clients. This is the nature of the Internet as well as the nature of many complex systems of systems in which components have different owners and belong to different organizations as shown in Figure 2. In some cases of course, there might be an internal notification of system changes but the semantics of the notification system might not be externally understood.

Although there are several existing partial approaches to these problems, they are not generally applicable in our context. For example, some solutions deal only with programmed change where all possible ways of upgrading are hard-wired into the design and information about upgrading is always passed between components. This does not work in our context in which we deal with pre-existing component systems but still want to be able to deal with interface upgrading in a safe and reasonable fashion. Other approaches that attempt to deal with unanticipated or evolutionary change in a way that makes dynamic reconfiguration transparent to the TA integrators[13] may be found in the AI field. However, our intention is not to hide changes from the application level. Our aim is to provide a solution that is application-specific and reliant on general approaches to dealing with abnormal situations. In particular, we will be building on existing research in fault tolerance and exception handling which offer disciplined and structured ways of dealing with errors of any types [Cristian 1995] at the application level.

[13] We use terms TA integrators and TA developers interchangeably.

Our overall aim is to propose structured multi-level mechanisms that assist developers in protecting the integrated DSoSs from interface changes and, if possible, in letting these DSoSs continue providing the required services.

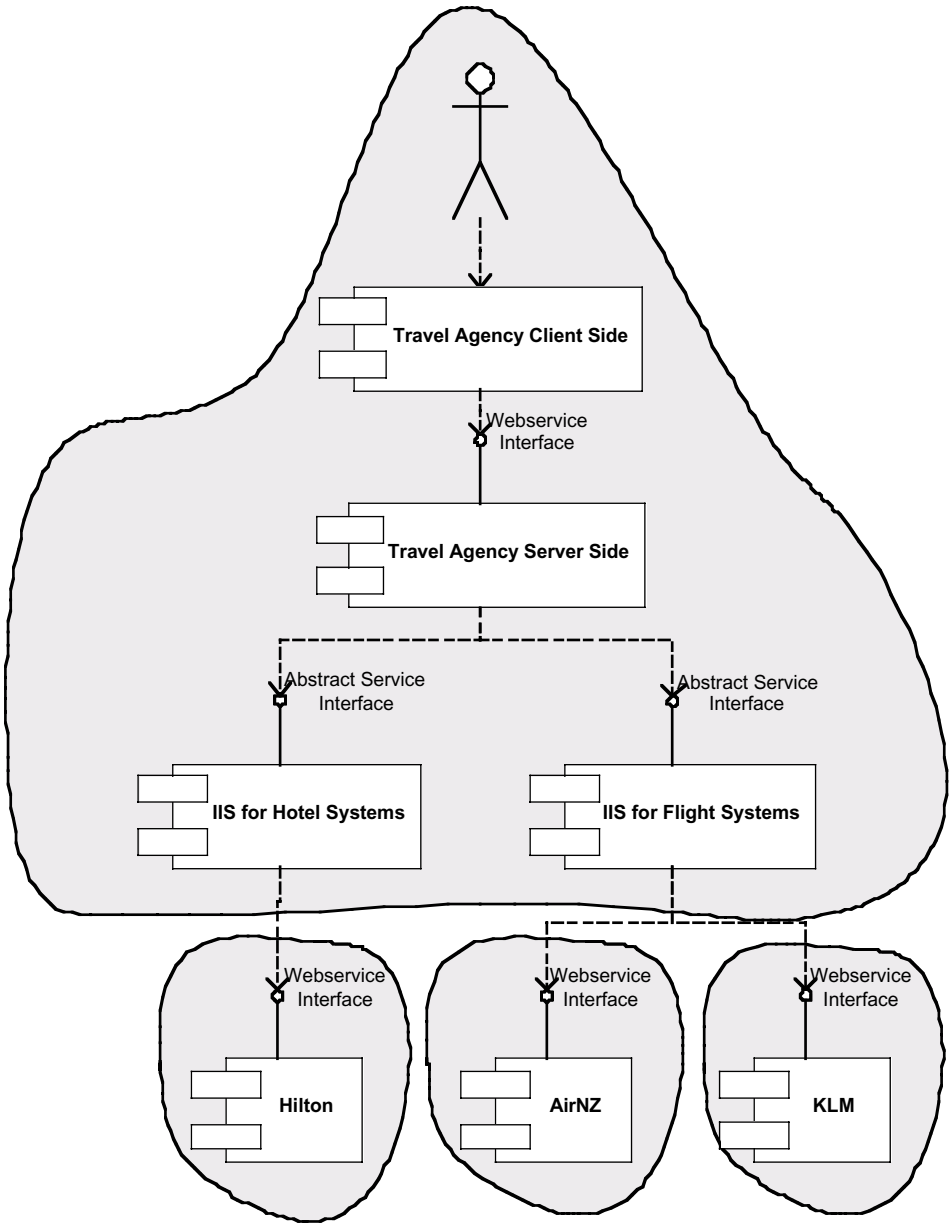


Figure 2 UML Component diagram showing the component systems that make up the Internet Travel Agency (TA). The grey areas indicate the fact that the component systems are under the control of different organisations. A user is shown interacting with the Travel Agency Client Side component that validates client side inputs and passes requests to the Travel Agency Server Side component. The Travel Agent Server Side component handles each request by invoking multiple Intermediate Interfacing Subsystems (IIS). Each IIS provides an abstract service interface for a particular service type, for example the Flight Systems IIS provides an abstract service interface for booking flights with systems such as AirNZ and KLM even though each of these systems has different webservice interfaces.

2.2. System Model

Integrators compose a DSoS from existing components (systems) that are connected by interfaces, glue code and additional (newly-developed) components where necessary. An interface is a set of named operations that can be invoked by clients [Szyperski 1997]. We assume that the integrators know the component interfaces. Knowledge of the interfaces can be derived from several sources: interfaces can be either published or discovered (there is a number of new techniques emerging in this area), programmer's guides, interfaces are first-class entities in a number of environments such as interpreters, component technologies (CORBA, EJB), languages (Java).

Besides integrators there are other roles played by humans involved in the composed system at runtime, for example: clients of the composed system, other clients of the components, etc.

We assume that component upgrade is out of our control: components are upgraded somehow (e.g. off-line) and if necessary their states are consistently transferred from the old version to the new version.

2.3. The Framework

2.3.1. Structured Fault Tolerance

We propose to use fault tolerance as the paradigm for dealing with interface changes: specific changes are clearly abnormal situations (even if the developers accept their occurrence is inevitable), and we view them as errors of the integrated DSoS in the terminology accepted in the dependability community [Laprie 1995]. In the following we focus on error detection and error recovery as two main phases in tolerating faults.

Error detection aims at earlier detection of interface changes to assist in protecting the whole system from the failures which they can cause. For example, it is possible that, because of an undetected change in the interface, an input parameter is misinterpreted (a year is interpreted as a number of days the client is intending to stay in a hotel) causing serious harm. Error recovery follows error detection and can consist of a number of levels: in the best case dynamically reconfiguring the component/system and in the worst with a safe failure notification and off-line recovery.

Our structured approach to dealing with interface changes relies on multilevel exception handling which should be incorporated into a DSoS. It is our intention to "promote" multilevel structuring of complex applications to make it easier for developers to deal with a number of problems, but our main focus here is structured handling of interface changes. The general idea is straightforward [Cristian 1995]: during DSoS design or integration, the developer identifies errors that can be detected at each level and develops handlers for them; if handling is not possible at this level, an exception is propagated to the higher level and responsibility for recovery is passed to this level. In addition to this general scheme, study of some examples suggests classifications of changes which can be used as check lists.

2.3.2. Error Detection

In nearly all cases, there is a need for meta-information to detect interface changes. Such meta-information is a non-functional description of the interfaces (and possibly of their upgrades), which may capture both structural and semantic information. Some languages and most middleware maintain structural meta-information, for example Java allows structural introspection and CORBA supports interface discovery via specialised repositories. However, at present there is little work on handling changes to semantic meta-information.

Meta-information for a component includes descriptions of:

- call points (interfaces), including input parameters (types, allowable defaults), output parameters (types, allowable defaults), pre- and post-conditions, exceptions to be propagated
- protocols: the sequences of calls to be executed to perform specific activities (e.g. cancel a flight, rent a car). A high-level scripting language can be used for this.

Interface changes can be detected either by comparing meta-description of old and new interfaces or if a component supports some mechanism to notify clients of changes. Another, less general, and as such less reliable, way of detecting such changes is by using general error detection features (some reasonable run-time type checking; pre- and post-conditions, or assertions of other types of checking parameters in the call points; protective component wrappers, etc.).

The intention should be to associate a rich set of exceptions with structural and semantic interface changes (changing the type of a parameter, new parameters, additional call points, changing call points, changing protocols, etc.); this would allow the system developers to handle them effectively.

2.3.3. Error Recovery

Error recovery can be supported through the use of:

- different handlers (at the same level) for different exceptions related to different types of interface changes
- multilevel handling.

2.3.3.1. Different Handlers

System developers should try and handle the following types of exception:

- changes of types of parameters, new parameter, missing parameter, new call point
- changes of the protocols, re-ordering, splitting, joining, adding, renaming and the removal of protocol events
- change of the meta-description language itself (if components provide us with such a meta-description of its interface)

- raising of new exceptions, if the protocol changes then new exceptions may also be raised during its execution.

To provide some motivational examples, consider the Travel Agent case study.

- A very simple interface change is where the currency in which prices are quoted changes. In this case, simple type information could show, for example, that the TA system requires a price in Pounds Sterling and the Car rental is being quoted in Norwegian Crowns. An exception handler can ask for a price in Euros which might be countered with an offer to quote in Dollars. Note that this process is not the same as reducing everything to a common unit (dollars?), finding agreement earlier can result in real savings in conversions.

- A previously functioning communication from the TA system to a hotel reservation system might raise an exception if a previously un-experienced query comes back as to whether the client wants a non-smoking room. Either of two general strategies might help here: the query could come marked with a default which will be applied if no response is given (an exception handler could accept this option) or the coded value might (on request from the exception handler) translate into an ASCII string which can be passed to the client for interpretation.

- Some of the most interesting changes and incompatibilities are likely to be protocol changes. An airline system might suddenly start putting its special offers before any information dialogue can be performed; the order in which information is exchanged between the TA and its suppliers of cars, flights etc. might change. Given enough meta-information, it is in principle, possible to resolve such changes but this is far more complex than laying out the order of fields in a record: it is the actual order of query and response which can evolve.

- In the extreme, the chosen meta-language might change. Even here, a higher-level exception handler might be able to recover if the meta-language is from a know repertoire.

- When an airline ceases to respond (exist?) the TA system must cope with the exception by offering a reduced service from the remaining airlines.

- Communication with new systems might be established if there is some agreement on meta-languages which can be handled.

In all of the above cases, the attempt is to use exception handling to keep the TA system running. Of course, notification of such changes might well be sent to developers; but the continuing function of the TA should not await their availability.

2.3.3.2. Multilevel Handling

Exceptions are propagated to a higher level if an exception is not explicitly handled or an attempt to handle the exception fails. This leads to a recursive system structuring with handlers being associated with different levels of a system. Possible handling strategies are:

- request a description of the new interface from the upgraded component

- renegotiate the new protocol with the component

- use a default value of the new parameters
- pass the unrecognised parameters to the end client (e.g. in ASCII)
- involve system operators into handling
- exclude the component from the operation
- execute safe stop of the whole system.

When designing handlers DSoS developers can apply the concepts of backward recovery, forward recovery or error compensation [Laprie 1995]. Backward recovery restores the system to its state before the error, for example the TA abandons (aborts) a set of partial bookings making up an itinerary if one of the components cannot satisfy a particular booking. Forward recovery finds a new system state from which the system can still operate correctly, for example where DSoS developers decide to involve people in handling interface changes: TA support/developers, TA users/clients, component support. Error compensation relies upon the system state containing enough redundancy to allow the masking of the error. An example of error compensation is the use of redundant components. For example, in the TA case study if the KLM server changes its interface and TA cannot deal with this, it ignores it but continues using servers of BA and AirFrance.

After the TA has been safely stopped or a component has been excluded, the TA support and developers can perform off-line analysis of the new interface of the component (cf. fault diagnosis in [Laprie 1995]). To improve the system performance and to make better use of the recent interface upgrades the TA application logic can be off-line modified when necessary following the ideas of fault treatment [Laprie 1995].

2.4. Representing Meaning

In order to communicate semantic information between two computers or in the case of the TA between the SoS and its providers we need a structured collection of information (meta-data) as well as a set of inference rules that can be used to conduct automated reasoning. Traditionally knowledge engineering [Hruska & Hashimoto 2000], as this process is often called, requires all participants to share the same definitions of concepts. In our case, for example, definitions of what is a trip or a flight as well as the parameters for each of these have to be defined and shared. The protocol for booking and paying for a trip or an item is also required. Detailed descriptions of the parameter types and their semantic information also need to be held in a shared knowledge base. Knowledge bases however and their usage does not necessarily make the system more flexible; quite the contrary. Requests would have to be performed under strict rules for inference and deduction. The SoS would have to process its metadata (globally shared data descriptions) in order to infer how to make a request for a particular method (i.e. booking a flight) and further more infer what parameters accompany this method and what is their meaning.

This process requires a well defined globally shared description of the domain in which the SoS operates. Such a definition is usually called ontological definition and the process is referred to as ontological modeling. Current developments in web architectures and distributed systems are working towards communicating meta-data information across components systems. XML for

example allows us to define our own tags in order to structure web pages and it is also widely used for structuring soap messages sent to components systems (web services). XML effectively allows the user to define its own tags the process of which is shared via a common document type definition which in turn enables both client and servers to interpret them. In order however to comprehend the semantics behind it then we need human intervention. To put it simply; a user for a share price component that accepts a string and returns the price of the share represented by the string may have a tag type stock price. So within the tags <stockprice> and </stockprice> the price of the share would be returned. Via XML we could communication between a provider and a consumer that a certain tag is of type String or Integer but that does not encapsulate its semantic information. This is where the resource definition framework (RDF) [W3C-RDF 2000] may help as it provides a technique for describing resources on the web. It is a framework for specifying metadata using XML syntax.

In conjunction with RDF interface descriptions could also bear and communicate their semantic information. All these technologies and disciplines however have not yet being put together for any meaningful application. They exist separately as hypertext and TCP/IP for example existed before the Internet. Using the current API's our choices are limited. The next section shows how Java API is used in our case study to deal with interface upgrading.

2.5. Java RMI Implementation

This section discusses how some part of the general framework presented above is being applied within our ongoing experimental work on implementing Internet TA [Periorellis & Dobson 2001]. Current API's allow us to carry out some work towards dealing with online dynamic upgrades, although there is significant work to be done not just in programming terms at the application level but in terms of providing an adequate API that would allow us to overcome certain technical difficulties.

Java RMI does not offer a full API for dynamic interfaces. However, it does support dynamic invocation when used in conjunction with the standard Java reflective API. The client does not need to maintain a local copy of a stub for a remote service, and neither does it need to maintain a local copy of the interface for the remote service. This is because Java RMI supports the automatic downloading of RMI stubs on demand, and once the stub has been downloaded then the standard Java reflection API can be used to discover and invoke the methods supported by the stub and therefore the remote service. The limitation of this approach is that if the stub changes during the lifetime of the client then a replacement stub cannot be downloaded. This is due to caching at the client side, as the replacement stub has the same name as the original stub then the cached copy is used instead of downloading the stub again.

The TA prototype is using Java RMI and the standard Java reflective API to dynamically compose the emerging service out of participating components. As the stubs can be downloaded and the interface of the stubs discovered at runtime this allows the SoS to determine the composition of the emerging service at runtime. In order to implement such a structure we need four machines: one to act as an RMI server that accepts requests for component systems (e.g. playing a role of a KLM server), a client (IIS in our case, see Figure 2) and a stub repository that makes the stubs available via the network (this could be a web server or an anonymous FTP server), and a machine that hosts the RMI registry. In our implementation we maintain the stubs

at the web server while the RMI server holds the actual implementations of the component systems, supporting classes and the interface description.

Each IIS only holds the names of the SoS component systems that it wraps. Each name is a human readable, implementation-independent reference that is registered with the RMI registry. This allows the location of SoS components to change without forcing changes to the implementation of the IIS. When the IIS invokes a SoS component service it queries the RMI registry for the stub that represents the SoS component service. The stub is transparently downloaded from the stub repository by the RMI infrastructure as the stub does exist locally. The IIS then uses the Java reflection API to discover and invoke methods on the stub and via RMI the SoS component system.

As each IIS provides a fixed abstract interface to the TA SoS then any changes to SoS component systems are localised to the IIS. The TA SoS and, via the TA SoS, any clients may be informed of unexpected changes to the component systems if extra information that is not captured by the abstract interface is required in order to complete a request. We foresee this being handled via our distributed exception handling scheme.

We already have an initial prototype that does not deal with server upgrading which can be accessed at http://ouston.ncl.ac.uk/main.htm. There are several avenues we are exploring right now that would allow some handling of online dynamic upgrades to SoS component systems. Although, changes to SoS component system interfaces that take place during the lifetime of a IIS are not visible via changes to the stub we can detect that some change has occurred by catching marshalling/unmarshalling or connection refused exceptions that will be caused by an upgrade. At present the best course of action that we can suggest is to restart the IIS and thereby force the local copy of the stub for the SoS component system to be refreshed. Once it has been refreshed then we can compare the interface of the new stub with a cached description of the old stub, this would allow the exact nature of the change to be detected and the appropriate handlers to be invoked. In this approach the actual stubs represent the meta-information used for handling interface upgrades. Assuming that we can find some technical solution to the caching problem then it would be possible to avoid restarting the IIS and therefore handling the effect of the upgrade would be more transparent.

Under some assumptions (e.g. the registry is updated before the server has been replaced with a new one) several scenarios are possible with respect to handling interface changes. For example:

- if a marshalling/unmarshalling exception is raised while accessing a KLM server we force the refresh of the local stub for the KLM server and compare its interface with a cached description of the KLM server in order to discover what has changed.

- if a connection refused exception is raised we can find out if we are trying to access the server in the middle of upgrading by going to the registry. This case clearly needs additional features because there is no guarantee that KLM updates the registry and the server atomically.

Our experience shows that Java and the RMI architecture in particular are not the most appropriate technologies for evaluating and implementing dynamic interface updates even though additional features such as the Java reflection API can be used to implement a limited

form of dynamic interface discovery and remote. In particular, they do not allow us to call an updated service as a means for handling because of local caching of the stubs. By catching some RMI service exceptions we can infer that a service upgrade has occurred and this can drive manual clearing of the cache via a restart of the RMI client. Alternatively another way of handling such situation is to exclude the upgraded service from the following execution until the client logs off.

There are two directions in which we can progress from here. The first one is to see if we can modify the Java/RMI infrastructure to force local refresh of the stub cache. The second one is to use modern Web technologies which offer much more flexible features for on-line dealing with interface descriptions and provide dynamic discovery and invocation as first-class features.

2.6. Related Work

The distributed computing community has considered the problems of maintaining meta-information for service discovery within the context of loosely coupled distributed systems such as DSoSs. Most middleware systems implement some form of object trading service, for example CORBA has an Object Trader Service, Jini has a Lookup Service, and .NET uses services provided by the Universal Discovery, Description and Integration (UDDI) project. Furthermore recent developments supported by the World Wide Web Consortium (W3C)[14] include a number of XML-base languages complementing UDDI and allowing Web service interfaces [W3C-WSDL 2001] and business-level conversations supported by such services (e.g. [W3C-WSCL 2002]) to be described. Object traders enable providers to advertise services by registering offered interfaces with a trading service. Clients locate a service by querying the trader using descriptions based on the structure of an interface and quantitative constraints [Szyperski 1997]. As with our proposed solution, object traders provide the ability to associate some meta-information with services. However, there is an assumption that once a client has found a service that uses a particular interface then that interface will remain static. Another difference is that we plan to maintain a richer set of meta-information with services that capture both structural and semantic information about interfaces such as versioning information, protocols, meta-information related to ontology and knowledge representation, dealing with abnormal situations while using the service, associating typical scenarios with the protocols, etc.

On the other hand, the object oriented database community has explicitly considered system evolution. They have developed schemes for schema evolution, schema versioning and class versioning. For example, in [Amann *et al.* 2000] schemata of multiple DBs are expressed in XML. In this approach the user's queries are written using a domain standard, that identifies the various entities and relationships, and for each data-source/base there is a mapping from that source entities to the domain standard. So, that a rewriting of the user's query to the various source formats can be done automatically. Our work differs in that in addition to structural changes we consider semantic changes such as protocol mismatches that occur when evolution takes place. Also the solutions proposed by this community tend to assume the existence of a centralised authority for enforcing control whereas we are working in the context of decentralised authority.

---

[14] http://www.w3.org/

There has been some work on resolving protocol mismatches in the area of component-based development. In [Vanderperren 2002] the concept of a component adaptor is introduced. It describes adaptations of the external behaviour independently of a specific API. When the adapter is applied to a composition of components the required adaptations can be automatically inserted. This is achieved through the application of algorithms that are based on finite automata theory. Our work differs in that we consider dynamic rather than build-time changes to protocols and we consider more wide ranging adaptation than just the renaming or addition of protocol events.

In our future work on the TA case study we intend to exploit this related work and some other features provided by modern component-oriented technologies and Internet technologies. Other useful features that can be used are language support for runtime reflection [Welch 2002], interface repositories and type libraries, and services such as CORBA's Meta-Object Facility that defines standard interfaces for defining and manipulating meta-models.

2.7. Concluding Remarks

This chapter has not proposed a totally general or efficient solution; our interest is in providing a pragmatic approach that explicitly uses a fault tolerance framework. Our work is motivated by real problems encountered when considering a case study where mismatches due to evolution must be dealt with at runtime. Although there are some existing approaches to this problem we do not try to hide evolution from the application developer but provide a framework for dealing with it dynamically.

Chapter 3 - From Error Detection to Recovery Wrappers

Manuel Rodriguez, Jean-Charles Fabre, Jean Arlat, Eric Marsden ***(LAAS-CNRS)***

3.1. Introduction

The use of commercial-off-the-shelf (COTS) software components, in particular real-time executive software (microkernels, conventional operating systems, middleware layers and virtual machines), is an attractive approach to realise systems that can be part of larger infrastructures, namely systems of systems. As far as dependability is concerned, their behaviour in the presence of faults remains an open issue. This issue has been investigated using fault injection techniques to help characterize their failure modes [Koopman & DeVale 1999, Arlat *et al.* 2002]. The observed failure modes reveal some weaknesses in the design and the implementation of the COTS software components, mainly because dependability was not a major concern in their development process. The need to master and maximize the coverage of fault assumptions is a crucial aspect of dependable real-time applications, not only regarding the real-time executive itself, but also regarding the selection of adequate fault tolerance strategies.

Improving COTS components from a dependability viewpoint has been often done by means of wrappers. To date, wrappers have been essentially applied to supplement the built-in error detection mechanisms and thus improve the overall error detection coverage (e.g., see [Voas 1998, Ghosh *et al.* 1999, Arlat *et al.* 2002]). We have extended the conventional notion of wrappers by means of recovery actions. The idea is not only to improve the signalling of errors, but also to define actions able to recover from transient faults or to put the system in a safe state. Clearly the objective is to improve the fault assumptions that can be made by an integrator of systems of systems. Indeed, distributed fault tolerance algorithms deployed in a system of systems are based on assumptions whose coverage must be made as high as possible. The detailed identification of the erroneous state provided by our formal specifications based approach enable these actions to be defined and implemented on various target software components.

As far as CORBA middleware-based systems are concerned, the work we have carried out by fault injection (cf. IC3 DSOS report) shows that CORBA implementations are very sensitive to corrupted inter-object requests between clients and servers. The results we obtained highlight the possible behaviour in the presence of faults of essential CORBA services when corrupted IIOP requests are received. The impact of these types of faults can be reduced through the use of simple wrappers, such as electronic signatures applied to IIOP requests at the application level. Another major source of problems, leading to the incorrect behaviour of a middleware layer like CORBA, is the reaction of the operating system support (microkernel, conventional operating system) to abnormal situations. For instance, some external and internal faults may lead the operating system layer to return error status codes and exceptions to the middleware. The way in which the middleware handles these inputs is really a major issue from a dependability viewpoint. From early experiments, we have observed that poor handling of such exceptional conditions may lead to a complete crash of the system or to error propagation to the application level. This is not surprising as middleware implementors sometimes neglect to consider every possible outcome of a call to the executive support, ignoring conditions that are very rare or unexpected (such as resource issues). It si thus important to make the executive support

as reliable as possible, since the middleware implementation does not implement any recovery actions to deal with corrupted behavior of the underlying operating system.

As a matter of fact, a middleware system can be wrapped from above, by means of simple wrappers preventing corrupted requests to be forwarded to target services, but also a middleware system must be wrapped from below to prevent the executive layer from disturbing the middleware layer.

In the work reported in this chapter we concentrate on this idea of recovery wrappers and illustrate the benefits that can be obtained in particular to maximize fault assumptions coverage. These extended wrappers are based on a framework previously described in the IC2 DSOS report.

For the sake of completeness we briefly describe this wrapping framework in Section 3.2, which focuses on the formal development of wrappers for error detection [Rodríguez *et al.* 2002b].

In the sequel we describe in detail how such wrappers can be enhanced with forward error recovery capabilities. They are based on a set of simple error handlers called recovery actions. The objective is that several elementary data modifications carried out concurrently by the recovery actions into the target component, be able to eliminate — or at least minimize — the errors detected by the wrappers. In addition, the recovery actions are implemented in a disciplined way using reflective concepts [Maes 1987].

We also show how the temporal costs of both the detection and the recovery capabilities of the wrappers can be made compatible with the hard deadlines of a target real-time application, while maximizing the error detection and recovery coverage of the wrappers. To do so, we first extend the schedulability test of the application. Then, instead of using a theoretic fault model (like in [Burns *et al.* 1999]), we study experimentally, by means of software implemented fault injection (SWIFI) [Carreira *et al.* 1998], the impact of errors on the activation profile of the wrappers.

Accordingly, the rest of this chapter is structured as follows. The fault tolerance wrapping framework is briefly presented in Section 3.2. Section 3.3 describes the notion of recovery action and how it fits within the framework defined. Section 3.4 shows how the wrapping framework can be applied to real-time microkernel based systems. We elaborate here on the concept of reflective real-time executive — an essential notion to observe and control the microkernel —, and we also illustrate the behavior of the wrappers both in the absence and in the presence of faults. The case study provided in Section 3.5 consists of a real-time application running on a COTS real-time microkernel. The kernel is encapsulated with a large number of wrappers. Selected results derived from fault injection experiments are provided and discussed. These results help us analyze how the temporal overheads of the wrappers can be made compatible with the hard deadlines of the application. Section 3.6 provides a discussion on the proposed wrapping technology, its interest within middleware-based systems and some other wrapping aspects in the context of DSoS. Section 3.7 concludes the Chapter.

3.2. Wrapping Framework for Fault Tolerance

The fault tolerance wrapping framework (Figure 3) is composed of five elements: the *specification* (a formal description of the system requirements), the *wrappers* (the executable version of the specification), the *runtime checker* (a platform for the execution of the wrappers), the *observation & control* (a software layer providing interceptors and methods to observe and modify the behavior of the system), and the *Target Software Component* or TSC (the component of the system that is to be wrapped).

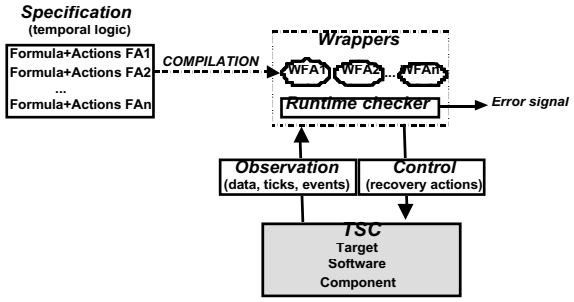


Figure 3 Overall framework

It is worth noting that while this framework is generic and can be used with any TSC, our work focuses on its application to real-time microkernel-based systems.

To specify system requirements, we have defined a formal language based on temporal logic, called *Clock and Event driven Temporal Logic* (CETL), whose detailed syntax and semantics are provided in [Rodríguez *et al.* 2002b]. The main particularity of this temporal logic is that it embodies the notions of *discrete time* (in the form of *clock triggers*) and *event*. As any temporal logic, it is built from temporal operators and a first order logic. The temporal operators defined are $\bigcirc$ (next clock trigger or asynchronous event), $\mathbf{O}$ (next clock trigger), $\Box$ (next asynchronous event), $\diamondsuit[e]$ (some time for asynchronous event e), $\diamondsuit[e]^{<k}$ (some time before k clock triggers for asynchronous event e). The first order logic consists of boolean predicates defining standard logical ($\wedge, \neg$, etc.), relational ($<, \leq$, etc.) and arithmetic ($+, -$, etc.) operators.

The *specification* of the TSC is given as a set of temporal logic formulas (*FA1*, *FA2*, …, *FAn* in Figure 3) which describe formally the parts of the TSC behavior that is to be controlled. A formula is based on a logical implication, composed of an antecedent and a consequent (see example in Section 3.4.1). Because of their particular structure, the formulas of the specification are referred to as *statements*. In addition, the statements also specify the recovery actions that are to be executed when a predicate of the consequent is violated.

The *wrappers* are the executable version of the statements from the specification. Their role is to detect timing and value errors of the TSC operation at runtime, and to recover from errors by properly executing the recovery actions. We have developed a compiler that automatically translates each statement into a single wrapper in C language (e.g., *FA1* into *WFA1*, *FA2* into *WFA2*, etc.). An example of a wrapper is provided in Section 3.4.1.

The *runtime checker* is a sort of virtual machine in charge of executing the wrappers (*WFA1*, *WFA2*, …, *WFAn* in the figure). Essentially, it is an interpreter of temporal logic that provides an interface with services for managing the temporal operators (Table 1a) as well as the predicates of the antecedent and the consequent of a statement (Table 1b).

Services (C language)	*Meaning*
(a) Management of temporal operators *(F is a temporal logic formula, and e is an event identifier)*	
`NEXT (k, F, context);`	$\bigcirc^k$ (F)
`NEXT_CLOCK (k, F, context);`	$\mathbf{O}^k$ (F)
`NEXT_EVENT (k, F, context);`	$\circledcirc^k$ (F)
`SOMETIME (e, F, context);`	$\diamond$[e] (F)
`BOUND SOMETIME (e, k, F, context);`	$\diamond[e]^{<k}$(F)
(b) Management of predicates	
`CONDITION (predicates);`	• *Evaluates predicates of the antecedent*
`ASSERT (predicate, recovery_action);`	• *Evaluates a predicate of the consequent. It signals an error only if after the execution of recovery action the predicate is false.*
(c) Management of the wrapper context	
`NEW_CONTEXT ();`	• *Creates a new context from a static memory pool*
`CONTEXT_SET (value, context, index);`	• *Updates context[index] with parameter value, and returns parameter value*
`CONTEXT GET (context, index)`	• *Returns the contents of context[index]*

Table 1 Services provided by the runtime checker

Note that the runtime checker signals an error if a predicate of the consequent of a statement is false after the execution of its associated recovery action (service *ASSERT*, see also Section 3.3). Wrappers are executed concurrently by the runtime checker. Concurrency is made possible thanks to the decomposition of a wrapper into functions that are to be executed at different instants. Internally, the runtime checker maintains a form of process context block for each wrapper, which characterizes the global state of the wrapper along its different executions. Such an information is referred to as *wrapper context*, whose corresponding runtime checker services are listed in Table 1c.

The *observation & control layer* allows the wrappers to observe and control the behavior of the TSC. The *observation* part of this layer is in charge of providing the necessary TSC information to the runtime checker and the wrappers. Such an information may consist of messages [Diaz *et al.* 1994], event occurrences [Mok & Liu 1997], signals [Savor & Seviora 1997], or states [Schneider 1998]. Indeed, it depends very much on the formalism used to describe the TSC requirements. In our case, the temporal logic used is built from predicates that describe the internal state of the TSC at different instants of time signaled by clock triggers and event occurrences. Accordingly, the type of information we need to observe correspond to internal TSC data, clock triggers (or *ticks*) and asynchronous events, as indicated in Figure 3. Conversely, the *control* part of the observation & control layer allows the wrappers to modify the behavior of the TSC when an error is detected. This part is composed of a set of *recovery actions*, each of them in charge of performing an elementary modification into the TSC (e.g., the insertion of a task into a queue or the substitution of the running task by another task). The objective is that several elementary modifications carried out by a set of recovery actions into the TSC be able to eliminate (or at least minimize) the error detected by the wrappers. Note that the observation & control layer makes the runtime checker and the wrappers independent from the particular implementation of the underlying TSC. In other words, when different implementations of the same TSC are to be tested (e.g., different implementations of the same POSIX interface), only the observation & control layer must be modified. The defined wrapping framework is compatible with any particular type of observation & control layer provided. As an example, in Sec-

tion 3.4.3 we used a *reflective* approach [Maes 1987] to develop such a layer for a TSC consisting of a real-time microkernel.

3.3. The Recovery Actions

A recovery action consists of a distinct implementation (i.e., a variant) of a basic function of the TSC. It is thus characterized by its *diversified implementation* and its *minimal functionality*. As an example, let us consider the yielding of the running thread, which is a function common to every software executive. With respect to the set of services provided by a software executive (synchronization, temporization, etc.) the functionality of this function is minimal. In addition, considering the particular implementation provided for such a function by a given software executive, we can develop a related recovery action whose implementation is different.

The execution model of the recovery actions is described in Figure 4. This model corresponds to the internal algorithm developed by service *ASSERT* of the runtime checker.

When a predicate of a wrapper is violated, the runtime checker raises an error signal as long as such a predicate is not associated to any recovery action. Otherwise, the corresponding recovery action is executed. At the end of the execution, the violated predicate is checked again. If it is still false, then the runtime checker will signal an error. However, if the predicate has become true, it can be assumed that the error has been (possibly) corrected by the recovery action, and the wrapper can resume its execution. Since various wrappers are running concurrently, this execution model is carried out for various predicates from different wrappers being checked altogether. As wrappers execute concurrently, they help prevent the propagation of errors. Indeed, the rationale is that several elementary modifications carried out concurrently by the recovery actions into the TSC, are able to eliminate — or at least minimize — the errors at the origin of a violation of the specification.

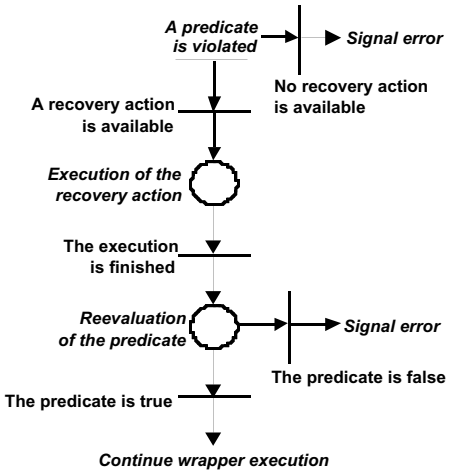


Figure 4 Execution model of the recovery actions (service *ASSERT*)

Note however that although the recovery actions can guarantee the satisfaction of the specification by means of an alternative implementation of certain elementary functions of the system (e.g., yielding the running thread), the error responsible for a predicate violation can however propagate, as wrappers cannot cover all faulty situations.

The implementation of the recovery actions is highly dependent on the target system. Indeed, they are based on the forward error recovery model. In Section 3.4.2, we provide an example of implementation of recovery actions.

3.4. Application to Real-Time Microkernel Based Systems

3.4.1. Example of a Wrapper

As an example of a wrapper for a microkernel, let consider a typical kernel service, namely *Create*. Figure 5 provides a statement specifying the creation of higher priority tasks carried out by service *Create*. A comprehensive temporal logic specification of the various services provided by real-time microkernels can be found in [Rodríguez *et al.* 2000].

Statement	***Create*** $\Box$ ($\diamond$[$\uparrow$*Create*] (*thb = created_th* $\wedge$ *tha = running* $\wedge$ *prio (thb) > prio (tha)* $\wedge$ $\diamond$[$\uparrow$*signal*] (*signaled_th == thb* $\wedge$ *running == tha*)) $\Rightarrow$ $\circledcirc$ (*event* == $\downarrow$*context_switch* $\wedge$ *running == thb* $\wedge$ *tha* $\in$ *ready (prio(tha))*))
Recovery Actions	*RA (running == thb) = changeRunning (thb)* *RA (tha* $\in$ *ready (prio(tha)) = insertThInFrontReadyQ (tha, prio(tha))*

Figure 5 Statement *Create* with its associated recovery actions

The interpretation of this statement (referred to as *statement Create*) is as follows. When the running task, represented by *tha*, requests the creation of a higher priority task *thb*, the kernel routine corresponding to service *Create* (indicated by event $\uparrow$*Create*) is executed. Some time later, the kernel inserts the newly created task *thb* into the ready queue (event $\uparrow$*signal*). As the child task has a higher priority than its parent, the latter is preempted after a context switch operation (event $\downarrow$*context_switch*). As a result, child task *thb* is elected to run (predicate *running == thb*), while parent task *tha* is inserted back into the ready queue (predicate *tha* $\in$ *ready (prio(tha))*).

The *antecedent* of statement *Create* is represented by the term:

$$\diamond[\uparrow Create]\ (thb = created\_th \wedge tha = running \wedge prio\ (thb) > prio\ (tha) \wedge \diamond[\uparrow signal]\ (signaled\_th == thb \wedge running == tha\ ))$$

while its consequent corresponds to the term:

$$\circledcirc\ (event == \downarrow context\_switch \wedge running == thb \wedge tha \in ready\ (prio(tha))).$$

Because of operator always ($\Box$), the implication *antecedent* $\Rightarrow$ *consequent* must be satisfied in all the computations of the target system. In general, an implication is satisfied either when the antecedent is false (irrespective of the consequent), or when both the antecedent and the consequent are true. Therefore, the implication is violated *only* when the antecedent is true while the consequent is false. In other words, an error is detected in a system computation by statement *Create* when, at the occurrence of event $\uparrow$*Create*, the antecedent is true while the consequent is false.

Two recovery actions have been associated to this wrapper:

- *changeRunning (thb)*, which will try to substitute the running task by task *thb* whenever predicate running == *thb* is violated;
- *insertThInFrontReadyQ (tha, prio(tha))*, which will try to insert task *tha* at the front of its priority level ready queue whenever predicate *tha ∈ ready (prio(tha))* is violated.

We have developed a compiler that automatically translates a statement into its corresponding wrapper. The wrapper generated by the compiler for statement *Create* (referred to as *wrapper Create*) is shown in Figure 6. In the wrapper code, routines *ANT_1* and *ANT_2* represent the antecedent, while routine *CON* represents the consequent.

Each temporal operator of statement *Create* corresponds to a call to a temporal operator service of the runtime checker (e.g., *SOMETIME, NEXT_EVENT*). The actual values of the kernel variables are obtained by executing a *get_* instruction (e.g., *get_created_th*, *get_running*). Predicates of the antecedent are assessed by service *CONDITION*, while those from the consequent are evaluated by service *ASSERT*. The definition of auxiliary variables is allowed within statements. For instance, statement *Create* defines auxiliary variables *tha* and *thb*, which are respectively assigned the identifiers of the running task and of the created task. Finally, variable *context* points to a structure storing the context of the wrapper. The wrapper context is composed of the set of auxiliary variables defined within a statement. For instance, the context of statement *Create* is composed of auxiliary variables *tha* and *thb*. The runtime checker holds the wrapper context between the execution of the different routines of the wrapper.

```
int start () {
    return SOMETIME (ev_begin_Create, (void*) ANT_1, null);
}
int ANT_1 (Context* context) {
    int thb = get_created_th ();
    int tha = get_running ();
    CONDITION (prio (thb) > prio (tha));
    context = NEW_CONTEXT ();                          /* Create a wrapper context */
    CONTEXT_SET (thb, context, 1);                     /* Update the context */
    CONTEXT_SET (tha, context, 2);
    return SOMETIME (ev_begin_signal, (void*) ANT_2, context);
}
int ANT_2 (Context* context) {
    int thb = CONTEXT_GET (context, 1);                /* Retrieve context data */
    int tha = CONTEXT_GET (context, 2);
    int running = get_running ();                      /* Obtain kernel data */
    int signaled_th = get_signaled_th ();
    CONDITION (signaled_th == thb && running == tha);  /* Request the evaluation of predicates */
    return NEXT_EVENT (1, (void*) CON, context);       /* Set the next temporal operator */
}
int CON (Context* context) {
    int thb = CONTEXT_GET (context, 1);
    int tha = CONTEXT_GET (context, 2);
    ASSERT (get_event_id () == ev_end_context_switch, NULL);
    ASSERT (get_running () == thb, changeRunning (thb));
    ASSERT (isInReadyQ (tha, ready (prio (tha))),
        insertThInFrontReadyQ (tha, prio (tha)));
}
```

Figure 6 Wrapper *Create*

3.4.2. Example of Implementation of Recovery Actions

Figure 7 provides the implementation of the recovery action *changeRunning* (defined by statement *Create* in Figure 5) for the Chorus microkernel.

This recovery action provides a different implementation (i.e., a variant) of the kernel function responsible for yielding the running thread, and is executed whenever a failure affecting such a

function occurs (e.g., violation of predicate *running == thb* of statement *Create* in Figure 5). The recovery action is decomposed into the following steps: kernel synchronization (1-2), thread identification (3-4), updating memory and internal kernel structures (5-8), and hardware context-switch (9-11).

```
int changeRunning (int threadId) {
    Assert (!KERN_INTR_LOCK_IS_HOLD_ANY());                    /* 1 - no kernel locks must be held */
    Assert (SWITCH_IS_HEALTHY());                              /* 2- context switch must be permitted */
    Thread* th = ThreadTable::publicLidCheck(threadId);        /* 3 - find the corresponding kernel thread */
    if (th == NULL) return FALSE;                              /* 4 - kernel thread not found, so return */
    removeThFromRQ(schedThread((Thread*)th));                  /* 5 - remove new thread from ready queue */
    if (memSwitchRequired) {
        memSwitch((Thread*) currentThread, (Thread*)th);       /* 6 - perform a memory switch if necessary */
    }
    currentThread = (Thread*)th;                               /* 7 - the running thread variable is updated */
    CurrentTss->esp0 = (unsigned int) (int)th->thrStack;       /* 8 - the stack of the running thread is updated*/
    int thThrSwitchCtx = (int)th->thrSwitchCtx;                /* 9 - process context block of the new thread */
    int oldThrSwitchCtx = (int)&(((Thread_f*)currentThread)->thrSwitchCtx); /*10 - context of the current thread */
    context_switch (oldThrSwitchCtx, thThrSwitchCtx);          /* 11 - perform a context switch */
}
```

Figure 7 Recovery action *changeRunning* for the Chorus microkernel

3.4.3. *Reflective* Real-Time Microkernel

In this section, we describe how the internal state of a microkernel can be observed and controlled using *reflection* [Maes 1987]. In a reflective approach, the target system delivers events to the wrappers, whereas the wrappers get the necessary additional information from the target system. In reflective terms, the former relates to the notion of *reification*, and the latter to the notion of *introspection*. In addition, reflection also allows the behavior of the target system to be controlled using mechanisms based on the concept of *intercession*. These notions are refined in the next paragraphs.

In a reflective system [Kiczales *et al.* 1991, Fabre & Pérennou 1998], a clear distinction is made between the so-called *base-level*, running the target system, and the *metalevel*, responsible for controlling and updating the behavior of the target system. Information is provided from the base-level to the metalevel, that becomes *metalevel data* or *metainformation*. Any change in the metainformation is reflected to the base-level. The distinction made between the base-level and the metalevel provides a clear separation of concerns between the functional aspects handled at the base-level and the non-functional aspects (here, error detection and error recovery) handled at the metalevel.

Figure 8 details the various layers, components and mechanisms that make up the reflective framework. This framework complies with and extends the principles introduced in [Arlat *et al.* 2002].

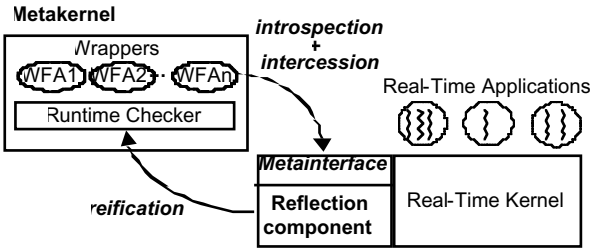


Figure 8 Reflective framework

The base-level of our reflective framework is the real-time microkernel, while the metalevel is called the *metakernel*, composed of both the wrappers and the runtime checker. The association of both layers leads to the notion of *reflective real-time microkernel*. The kernel can be observed and controlled through the so-called *reflection component*, which is a special component added to the target microkernel. The reflection component is responsible for the management of the intercepted events (i.e., reification), the observation of internal items (i.e., introspection), and the required actions down into the real-time kernel (i.e., intercession). The reified events are delivered as *upcalls* to the metakernel, whereas introspection and intercession are provided by the reflection component through the so-called *metainterface*. The metainterface is defined as a set of services providing access to the necessary information from and actions into the real-time kernel.

As far as *reification* is concerned, events are delivered to the metakernel using *upcalls*. An upcall is a jump instruction inserted into the kernel that diverts the execution flow from the kernel to the metakernel. Such an upcall is similar to a jump assembly code instruction, i.e., it does not trigger any context switch. For example, statement *Create* defines event ↑*Create*, which corresponds to the start of the kernel service that carries out the execution of system call *Create*. Accordingly, an upcall is inserted at the beginning of the *Create* routine of the kernel, which takes as an input parameter the identifier of event ↑*Create* (see `service_create` below).

When the kernel enters routine `service_create`, the upcall is executed and diverts execution to the runtime checker. Events ↑*signal* and ↓*context_switch* of statement *Create* are reified in a similar way.

Clock ticks can also be reified by inserting an upcall at the beginning of the clock handler routine of the kernel, as indicated hereafter (see `clock_handler` below).

```
service_create (...) {
      upcall (ev_begin_create);
...
}
```

```
clock_handler (...) {
      upcall (clock_tick);
...
}
```

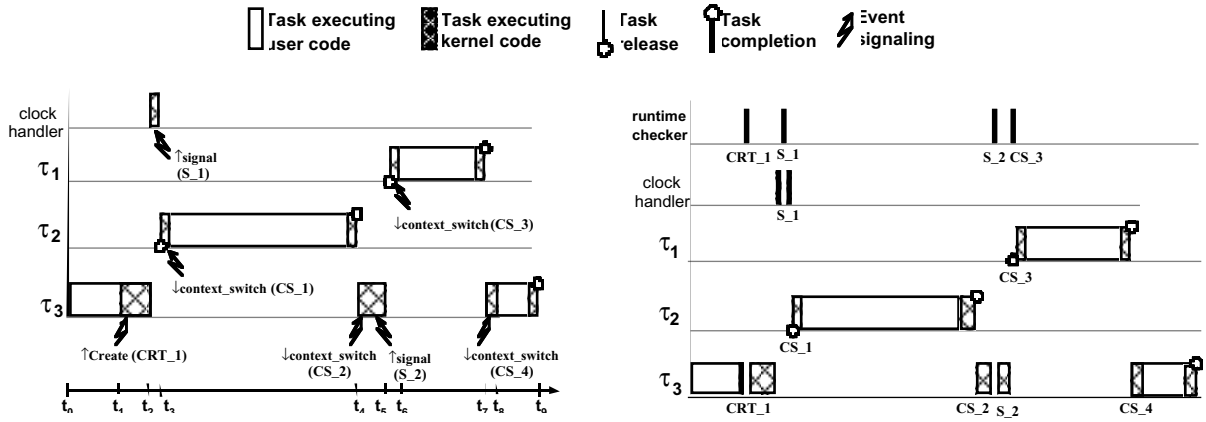
On the other hand, *introspection* and *intercession* facilities are provided through the metainterface. The definition of the metainterface is derived from the specification. Indeed, the specification points out the necessary events, data structures and functions that must be observed and controlled. To illustrate this point, Table 2 lists the set of services of the metainterface for statement *Create*.

Temporal logic	*Metainterface*		
	Instrospection		
created_th	`int`	`get_created_th`	`()`
running	`int`	`get_running`	`()`
signaled_th	`int`	`get_signaled_th`	`()`
event	`int`	`get_event_id`	`()`
prio (th)	`int`	`prio`	`(int th)`
ready (level)	`int`	`ready`	`(int level)`
th ∈ ready	`int`	`isInReadyQ`	`(int th)`
	Intercession		
running == th	`int`	`changeRunning`	`(int th)`
th ∈ ready (p)	`int`	`insertThInFrontReadyQ`	`(int th, int p)`

Table 2 Metainterface necessary to wrapper *Create*

3.4.4. Wrapper Execution while the Microkernel Behaves Correctly

This section illustrates the execution of the wrappers when the microkernel is not affected by errors. The application considered is represented in Figure 9 by a set of real-time tasks executing concurrently and requesting kernel service *Create*.



a) Original execution of tasks b) Execution of wrappers by the runtime checker

Figure 9 Execution of wrapper *Create* by the runtime checker

Figure 9a represents the original execution of the tasks together with the events triggered into the microkernel, while Figure 9b represents the same set of tasks extended with the runtime checker, which executes wrapper *Create*. The detailed behavior of the original set of tasks represented in Figure 9a is described in Table 3.

t_0	Task τ_3 is already running in user mode.
t_1	τ_3 requests the creation of a higher priority task (τ_1) by means of system call *Create*. Event CRT_1 is triggered when task τ_3 enters service *Create* in kernel mode.
t_2	A periodic tick interrupt leads to the execution of the clock handler of the kernel, which will move periodic task τ_2 from a suspend queue to the ready queue. A signal event (S_1) is triggered just before task τ_2 is inserted into the ready queue. Note that the clock handler does not preempt task τ_3 (i.e., there is no context-switch), but it simply executes at the highest priority on behalf of task τ_3.
t_3	Higher priority task τ_2 preempts task τ_3. Task τ_2 obtains the processor after a context switch (event CS_1).
t_3-t_4	Task τ_2 executes.
t_4	Task τ_2 suspends and task τ_3 is given the processor after a context switch (event CS_2). Task τ_3 continues execution of service *Create* in kernel mode.
t_5	Creation of task τ_1 is completed (event S_2).
t_6	Because priority of τ_1 is higher than priority of τ_3, the latter is preempted by its child, which obtains the processor after a context switch (event CS_3).
t_6-t_7	Child task τ_1 executes.
t_7	Child task ends execution. Its parent (τ_3) obtains the processor after a context switch (event CS_4).
t_8	Task τ_3 finishes executing service *Create* in kernel mode, and continues execution in user mode.
t_9	Task τ_3 ends execution and suspends.

Table 3 Original execution of tasks

Wrapper *Create* is executed by the runtime checker during the intervals represented inFigure 9b, labeled by the kernel event at the origin of the activation of the runtime checker. Remember that the runtime checker is a sort of virtual machine in charge of executing the wrappers, and is activated after the occurrence of an event triggered within the target system (denoting a state change). Note also that the runtime checker does not preempt, but simply diverts the execution flow of the task running at the moment of its activation, so no context switch is triggered. In other words, the runtime checker executes at the highest priority on behalf of the running task. In consequence, checks carried out by the wrappers by means of the runtime checker do not modify the original scheduling of tasks, as shown in Figure 9b.

Each activation of the runtime checker leads thus to the execution of one or several wrappers concurrently. The steps followed by the runtime checker to execute wrapper *Create*, as well as the checks performed by this wrapper, are detailed in Table 4.

Time ↓

Events		***Runtime checker actions***				
Activated	Expected type	Wrapper	Routine	Cxt	Expressions checked	Result
…	CRT					
CRT_1	CRT	Create	ANT_1		prio (τ1) > prio (τ3)	TRUE
S_1	S, CRT	Create	ANT_2	c1	signaled_th == τ1 $\wedge$ running == τ3	FALSE
CS_1	S, CRT					
CS_2	S, CRT					
S_2	S, CRT	Create	ANT_2	c1	signaled_th == τ1 $\wedge$ running == τ3	TRUE
CS_3	Any	Create	CON	c1	event == CS running == τ1 τ3 $\in$ ready (prio(τ3))	TRUE TRUE TRUE
CS_4	CRT					
…	CRT					

Table 4 Event occurrences and actions carried out by the runtime checker to verify *Create*

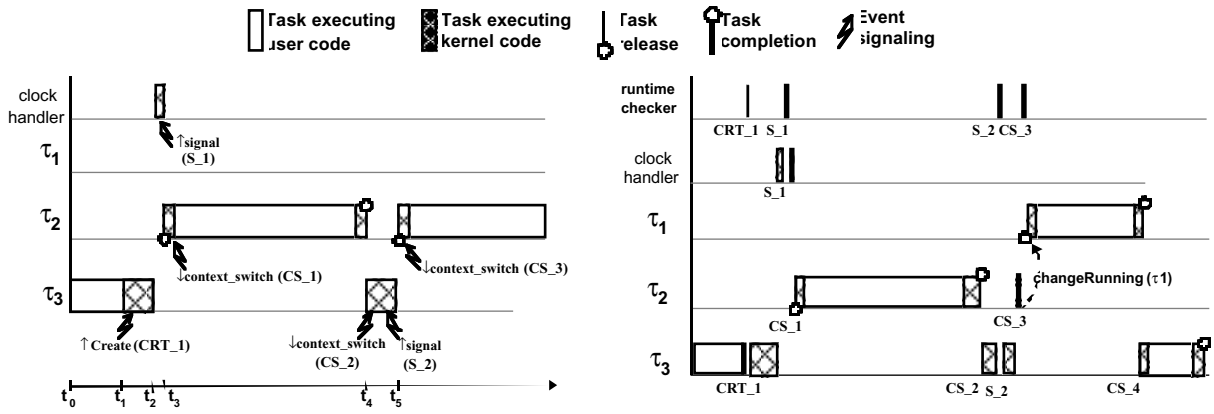
Column *Activated* contains the various events triggered during the execution of the system, while column *Expected type* corresponds to the type of events expected by the runtime checker at a given moment. Columns *Wrapper*, *Routine* and *Ctx* refer respectively to the name of the wrapper activated, to the wrapper routine executed, and to the wrapper context used. Column *Expressions checked* specifies the verifications performed by the wrappers. Note that auxiliary variables *tha* and *thb* of Figure 5 have been substituted in this column by the task identifier they represent (τ_1, τ_2, etc.), depending on the information contained into the corresponding wrapper context.

- Initially, given that Create is the only wrapper installed (for the sake of clarity), the single type of event expected by the runtime checker is ↑Create (denoted CRT).
- At the occurrence of event CRT_1, routine ANT_1 of wrapper Create is executed. As the child task (τ_1) has higher priority than its parent (τ_3), event ↑signal (S) is setup by the wrapper as an expected event. Context c1 is at this moment allocated with auxiliary variables tha = τ_3 and thb = τ_1. Next, the runtime checker suspends and waits for events ↑Create and ↑signal.
- Event S_1 (which signals task τ_2) triggers routine ANT_2 of wrapper Create under wrapper context c1. Given that the signaled task (τ_2) is not the child of task τ_3 (namely, task τ_1), the checked expression is false. Thus, the runtime checker suspends and keeps waiting for events ↑Create and ↑signal.
- Events CS_1 and CS_2 are ignored since they are not expected by the unique wrapper installed.
- Event S_2 (which signals τ_1) triggers routine ANT_2 of wrapper Create under wrapper context c1. The antecedent of Create is then evaluated to true, since the task signaled during the execution of τ_3 is indeed τ_1. The runtime checker waits then for the occurrence of any event.
- At the occurrence of event CS_3, the consequent of Create is evaluated under context c1. It is verified that the event triggered is a context switch, that the running task is τ_1, and that task τ_3 has been preempted into the ready queue. Since these expressions are evaluated to true, statement Create succeeds and no error is thus signaled.
- Finally, event CS_4 is ignored since the running checker is waiting for event ↑Create.

Note that if task τ_2 had requested a task creation, two instances of the same wrapper would have been executed concurrently. For the sake of conciseness, the example illustrates intentionally a single wrapper instance.

3.4.5. Wrapper Execution when an Error Impacts the Microkernel

In this section, we illustrate the behavior of the wrappers when the microkernel is affected by an error (Figure 10). Such an error leads to an incorrect scheduling of the tasks (Figure 10a). Then, wrapper *Create* successfully correct the error by means of its associated recovery actions (Figure 10b)



a) Incorrect execution of tasks b) Detection and recovery of the error by the wrappers

Figure 10 Example of an error detected and recovered by the wrappers

As depicted in Figure 10a (to be compared to Figure 9a), after the occurrence of event *signal* at instant t_5, the kernel has already inserted the newly created task τ_1 into the ready queue. Given that τ_1 is the highest priority task, it should be elected to run. However, because of an error affecting the kernel scheduler, the lower priority task τ_2 becomes running instead, just after a context switch. This leads in the sequel to an incorrect scheduling of the tasks (to be compared to the correct scheduling presented in Figure 9a). This error is successfully corrected by recovery action *changeRunning* of wrapper *Create*, as illustrated in Figure 10b and in Table 5.

Time

Events		***Runtime checker actions***				
Activated	Expected type	Wrapper	Routine	Cxt	Expressions checked	Result
…	CRT					
CRT_1	CRT	Create	ANT_1		prio (τ1) > prio (τ3)	TRUE
S_1	S, CRT	Create	ANT_2	c1	signaled_th == τ1 ∧ running == τ3	FALSE
CS_1	S, CRT					
CS_2	S, CRT					
S_2	S, CRT	Create	ANT_2	c1	signaled_th == τ1 ∧ running == τ3	TRUE
CS_3	Any	Create	CON	c1	event == CS	TRUE
					running == τ1	FALSE
					changeRunning (τ1)	
					running == τ1	TRUE
					τ3 ∈ ready (prio(τ3))	TRUE
CS_4	CRT					
…	CRT					

Table 5 Event occurrences and actions carried out by the runtime checker during the recovery of an error

As described in Table 5, the completion of context switch CS_3 leads to the execution of routine CON from wrapper *Create*. After checking the activated event, the wrapper compares the identifier of the running task (τ_2) with that of task τ_1. As they are different tasks, this check fails, and the wrapper triggers the recovery action *changeRunning* (τ_1). Such an action will try to put the system into a (possibly) correct state, by modifying the necessary tables, structures and registers of both the microkernel and the processor so that task τ_1 becomes the newly running task (an implementation example is provided in Figure 7). When the action finishes, the wrapper verifies *again* the identifier of the running task. Both Table 5 and Figure 10b illustrate the

case in which the recovery action *succeeds* (see Figure 9b for proof check). Next, the wrapper engages the verification of the last predicate, and the evaluation of statement *Create* is finally satisfied. As illustrated in Figure 10b, the scheduling of the tasks after the execution of the recovery action is correct (to be compared to the correct scheduling presented in Figure 9b).

3.5. Case Study

We characterize the error detection and recovery coverage and the performance of wrappers in a real-time system consisting of the *Chorus* microkernel [Chorus 1997] and the *mine drainage control system application* [Burns & Wellings 1997]. This is a typical runtime workload that we have already used in other types of studies [Rodríguez *et al.* 2002a].

The Chorus kernel was encapsulated with a set of error detection and recovery wrappers generated from an extended kernel specification, which encompassed the specification defined in [Rodríguez *et al.* 2000]. In total, 31 wrappers were used, corresponding to 18 scheduling statements, 2 timer statements and 11 synchronization statements. Also, a set of 18 recovery actions was implemented and used by the wrappers.

The mine drainage control system application has been used by a number of authors (e.g., [Burns & Lister 1991, Joseph 1996]). The detailed description of the application can be found in [Burns & Wellings 1997]. Table 6 contains the main attributes of the tasks that compose the application.

Task	Type	Priority	T (ms)	B (ms)	C (ms)	D (ms)	R (ms)
CH4 Sensor	Periodic	10	80	3	12	30	17.16
CO Sensor	Periodic	8	100	3	10	60	37.23
Air-Flow Sensor	Periodic	7	100	3	10	100	47.27
Water-Flow Sensor	Periodic	9	1000	3	10	40	27.19
HLW Handler	Sporadic	6	6000	3	20	200	67.32

T: Minimum inter-arrival time (task period) B: WC blocking time C: WC execution time D: Deadline R: WC response time

Table 6 Attributes and worst-case response times of tasks

The application is used to control a mining environment. The objective is to pump to the surface mine water collected in a sump at the bottom of the shaft. The main safety requirement is that the pump should not be operated when the level of methane gas in the mine reaches a high value due to the risk of explosion. The level of methane is monitored by task CH4 Sensor. Other environment parameters monitored are the level of carbon monoxide (task CO Sensor) and the flow of air in the mine (task Air-Flow Sensor). The flow of water in the pipes of the pump is checked by task Water-Flow Sensor, whereas the water levels in the sump are detected by task Hlw Handler.

3.5.1. Assessment by Fault Injection

MAFALDA-RT [Rodríguez *et al.* 2002a] was used to assess the efficacy and the performance of the kernel wrappers. The tool has been developed to encompass the assessment by fault

injection of both hard and soft real-time systems. It provides a facility to eliminate time intrusiveness by controlling the hardware clock of the target system and the external devices (e.g., sensors and actuators). Such a facility was used to eliminate the temporal overhead introduced both by the tool itself and by the wrappers. Therefore, the tasks were not aware neither of the execution of the tool nor of the wrappers from a temporal viewpoint. Note that we are using the wrappers in a testbed system, not in the final system; we are thus interested in evaluating wrapper coverage and wrapper performance without increasing the original execution time of the tasks. A full account of the MAFALDA-RT tool can be found in [Rodríguez *et al.* 2002a].

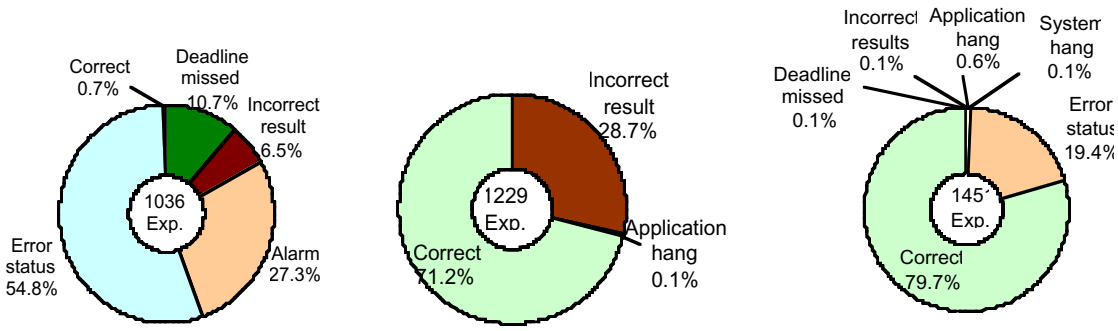
We carried out three different fault injection campaigns, in which the targets of the injected faults were the scheculing component (campaign sSCH), the timers component (campaign sTIM), and the Priority Ceiling Protocol component (campaign pPCP) of the microkernel. Table 7 briefly describes the types of faults injected in each campaign. It is worth noting that MAFALDA-RT systematically selects randomly the target of injection, and that it checks whether the corrupted element is accessed during the experiment, i.e., whether the fault is actually activated (only activated faults are considered).

Campaign	Injected faults
sSCH	Substitution of the running task by the next highest priority ready task.
sTIM	(#1) Random corruption by single bit-flip of the expiration time of a randomly selected sporadic timer. (#2) Avoiding once the insertion of a randomly selected sporadic timer into the timeout queue. (#3) Avoiding once the deletion of a randomly selected timer from the timeout queue. (#4) Random corruption by single bit-flip of the expiration time of a randomly selected periodic timer. (#5) Avoiding once the insertion of a randomly selected periodic timer into the timeout queue. (#6) Avoiding once the expiration of a randomly selected timer.
pPCP	Random corruption by single bit-flip of PCP system calls parameters during a call invocation.

Table 7 Faults injected (only one fault type injected per experiment at a random time)

3.5.1.1. Assessment when the Kernel is not Wrapped

Figure 11 reports the distribution of the *first* fault manifestations observed in the experiments (*#Exp.*) of the campaigns carried out in the non-wrapped version of the system.



a) sSCH b) sTIM c) pPCP

- The most critical situation occurs when an error propagates to the application, making it fail either in the time or in the value domain. Timing failures are represented by classes *Deadline missed*, *Application hang* and *System hang*, while value failures are represented by class *Incorrect result*.
- The error detection mechanisms of the microkernel are represented by classes *Alarm*, *Error status* and *Exception*.
- Class *Correct* represents the case when both the time production and the value of the application results are correct.

Figure 11 *First* fault manifestations observed for the set of experiments (*Exp.*) of each campaign

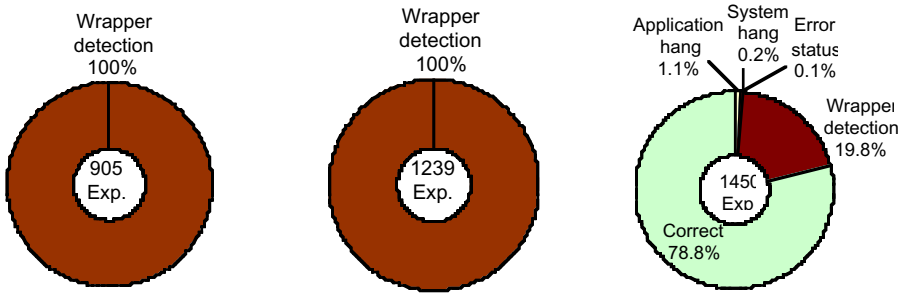
Faults injected in campaign sSCH provoked the substitution of the running task by a lower priority task while the former task was leaving a critical section. The impact of this error was catastrophic to the application, which missed deadlines and delivered incorrect results in respectively 10.7% and 6.5% of the cases. Most errors were however signaled by means of a built-in error detection mechanism. Indeed, an alarm detected a deadline missed in 27.3% of the cases, while 54.8% of the times the error status mechanism signaled the presence of an error affecting a synchronization system call.

Conversely, 28.7% of the faults injected in campaign sTIM led the application to issue incorrect results. Indeed, faults #5 and #6 prevented periodic tasks CH4 Sensor and CO Sensor from being released, while they are required to the correct computation of results. Nevertheless, 71.2% of the injected faults did not lead to any observable failure at the application level. After a careful analysis, we observed that this was due to: i) the internal redundancy of the timeout value of the timers, making injections #1 and #4 ineffective; ii) the elimination of alarms while the concerned tasks do not miss their deadline (injection #2); iii) the triggering of alarms that are not related to any application task (injection #3).

Few errors impaired the system when the parameters of the synchronization system calls were corrupted (campaign pPCP), because of the high percentage of correct experiments observed (79.7%). This is mostly due to the corruption of unused bits within parameters (randomly selection by the fault injection tool). Conversely, the consistency checks implemented within the kernel API (represented by class *error status*) detected most errors (19.4%). Few errors (0.9%) could thus propagate and lead to the failure of the application.

3.5.1.2. Assessment when the Kernel is Wrapped for Error Detection

We carried out the same fault injection campaigns using *only* the *error detection* capabilities of the wrappers. The distribution of the first fault manifestations observed are reported in Figure 12.



a) sSCH b) sTIM c) pPCP

Figure 12 *First* fault manifestations observed when wrappers are used for error detection

From the 31 wrappers installed, we report in Table 8 those that detected an error *first* in an experiment, namely *Timer_1*, *Take_winlock*, *Take_1*, *Give_1* and *Give_owner*. Their whole specifications are provided in [Rodríguez *et al.* 2002a], and their role is briefly explained in Table 8. We also observed that wrapper *Create* (Figure 6) did not detect any error. Indeed, all the tasks were created at the beginning of the experiments (i.e., wrapper *Create* was *only* activated at the beginning of the experiment), while faults were injected in the middle of the experiments. Hence, wrapper *Create* could not detect the errors caused by these faults.

Wrapper	*sSCH*	*sTIM*	*pPCP*
Timer_1		1239 *100 %*	
Take_winlock	306 *34%*		
Take_1	305 *34%*		
Give_owner			287 *100 %*
Give_1	294 *32%*		

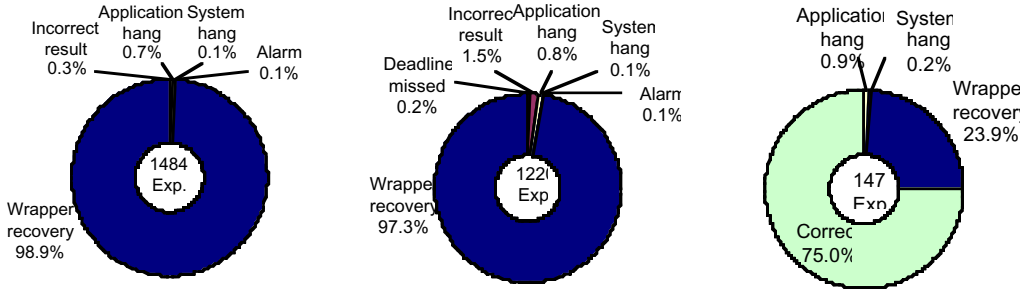
- *Timer_1* checks the kernel service responsible for setting timers.
- *Take_winlock* and *Take_1* check the kernel service of the PCP responsible for assigning a critical section.
- *Give_owner* and *Give_1* check the kernel service of the PCP responsible for releasing a critical section.

Table 8 *First* wrappers detecting an error

The error detection coverage provided by wrappers *Timer_1*, *Take_winlock*, *Take_1* and *Give_1* in campaigns sSCH and sTIM is perfect. Indeed, they have systematically intercepted all the errors at the origin of the fault manifestations of Figures 10a and 10b. Every failure has been avoided by these wrappers. Problems previously detected by means of an error detection mechanism of the kernel, are now notified by a wrapper with a shorter latency. Finally, latent errors within the kernel that did not previously lead to any observable failure, are now signaled by a wrapper. Conversely, in campaign pPCP, wrapper *Give_owner* detected, with a shorter latency, the same class of errors previously detected by means of an error status, involving the corruption of a parameter handling a critical section identifier. The remaining failure rate of this campaign, although not important, could not be further reduced by the wrappers.

3.5.1.3. Assessment when the Kernel is Wrapped for Error Detection and Error Recovery

The campaigns shown in Figure 12 only considered the error detection capabilities of the wrappers. In the sequel, we concentrate on the capacity of the wrappers to put the system in a (possibly) correct state by means of the recovery actions. Accordingly, we activated the error recovery capabilities of the wrappers, and we carried out again the same fault injection campaigns (see Figure 13).



a) sSCH b) sTIM c) pPCP

Figure 13 *First* fault manifestations observed when wrappers are used for error detection and recovery

Under the identical experimental conditions used, the wrappers detecting an error are the same as before (see Table 8). However, in this case, after the detection of an error, these wrappers executed a recovery action. We observed that every time a predicate of these wrappers was violated as a consequence of an error, such a predicate systematically became *true* after the execution of its associated recovery action, i.e., the recovery actions were locally successful. However, we only classed an experiment as a *wrapper recovery* (see Figure 13) when both the activated recovery actions were successful and no errors were observed in the sequel of the experiment. On the contrary, when the activated recovery actions were locally successful but one or several errors were observed in the sequel of the experiment, the latter was classified depending on the type of the *first* error then observed. These considerations are reflected in Figure 13, which reports the distribution of the first fault manifestations observed when the wrappers were equipped with the recovery actions. In addition, from the 18 recovery actions implemented, we report in Table 9 those that were activated *first* in an experiment classed as a wrapper recovery, namely *changeRunning*, *insertToInTimeoutQ*, *setTimeoutTicks* and *setTimeoutTicks*. Their role is also briefly explained in the table.

Recovery action	*sSCH*	*sTIM*	*pPCP*
`changeRunning`	1467 *100%*		
`insertToInTimeoutQ`		783 *66%*	
`setTimeoutTicks`		397 *34%*	
`changeOwnerCS`			353 *100%*

- `changeRunning` — Changes the running task.
- `insertToInTimeoutQ` — Inserts a timer into the timer queue.
- `setTimeoutTicks` — Changes the number of ticks (i.e., timeout) of a timer.
- `changeOwnerCS` — Changes the critical section of the owner task (PCP).

Table 9 *First* activated recovery actions

Figure 13 shows that all the errors appeared in campaigns sSCH and sTIM affecting respectively 98.9% and 97.3% of the experiments were efficiently corrected by the recovery actions. However, they could not prevent residual errors (that were not corrected) from provoking the failure of the system in 1.1% of the cases in sSCH, and in 2.6% in sTIM. Conversely, we observed that *all* the errors previously detected by a wrapper in campaign pPCP, were here corrected by means of recovery action *changeOwnerCS*.

3.5.2. Integration of Wrappers into a Real-Time System

In the fault injection campaigns carried out in Section 3.5.1, we have eliminated (thanks to MAFALDA-RT) the temporal overhead induced by the wrappers so as to characterize at best their error detection and recovery coverage, without perturbing the behavior of the target application. The quantitative measures obtained from this study also allow us to determine the set of wrappers optimizing the ratio coverage vs. temporal overhead that can be successfully integrated into the target real-time system in operation.

In general, a fault-tolerant mechanism can be integrated into a real-time system only when, in spite of the additional overheads it provokes, none of the *hard* deadlines of the real-time application is missed; in other words, when a feasible scheduling exists. For safety-critical real-time systems, such a feasible scheduling has to be guaranteed off-line (i.e., before the system is put into operation) by means of a schedulability test. The earlier this integration takes place in the development process of the target real-time system, the more it is readily to be achieved, because both application and wrappers can be developed together.

In the sequel, we show how the temporal cost of the wrappers can be made compatible with the hard deadlines of the target real-time application, while maximizing the error detection and recovery coverage of the wrappers. Neither the original system (application, kernel, hardware, etc.) nor its temporal constraints (deadlines, WCETs, etc.) are modified, which means that wrappers are to be integrated into an already developed system. First, we extend the schedulability test of the application to take into account the overheads caused by the wrappers. Then, we use the fault injection experiments carried out in Section 3.5.1 to analyze the influence of errors on the behavior of the wrappers. This allows us to experimentally calculate their overhead. Finally, we obtain the sets of wrappers that provide the best error detection and recovery coverage, while guaranteeing task deadlines.

3.5.2.1. Integration of Wrappers into the Schedulability Test

The schedulability test used to verify the satisfaction of deadlines in the mine drainage application [Burns & Wellings 1997] is related to a dynamic, preemptive scheduler with a fixed priority assignment scheme for a uniprocessor architecture-based system. However, this test can be distinguished from similar tests of its kind by the fact that it is issued from a complex system model that takes into account temporal costs imposed by the underlying operating system (context-switches, interrupts, clock ticks, etc.). We have further refined such a model and extended the schedulability test with the execution times of the wrappers (Figure 14).

As we have illustrated in Section 3.4.4, the wrappers execute at the highest priority on behalf of the running task. In consequence, the response time of a task increases not only because of the

wrappers activated during its execution (term W_i), but also because of those activated during the execution of higher priority tasks (term W_j). Term W_x represents thus the worst-case execution time of the wrappers in an instance of task *x*. The schedulability test is satisfied if and only if the worst-case response time of every task is lower than or equal to its deadline.

$$\forall i \in tasks, R_i \leq D_i$$

$$R_i = W_i + CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \left(W_j + CS^1 + CS^2 + C_j \right)$$

$$+ \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH + \left\lceil \frac{R_i}{T_{clk}} \right\rceil CT^c + \sum_{g \in \Gamma p} \left\lceil \frac{R_i}{T_g} \right\rceil CT^s$$

W_i	**Maximum execution time of the wrappers in an instance of task *i*.**
R_i	Worst case response time of task *i*.
D_i	Deadline of task *i*.
CS^1, CS^2	Maximum execution time of the initial and final parts of a context-switch.
C_i	Worst-case execution time (WCET) of task *i*.
B_i	Worst-case blocking time of task *i*.
$hp(i)$	Set of tasks whose priority is higher than *i*.
T_j	Period (or minimum inter-arrival time) of task *j*.
$\Gamma s, \Gamma p$	Set of sporadic and periodic tasks.
IH	Maximum execution time of an interrupt handler.
T_{clk}	Clock period.
CT^c	Maximum execution time of the clock handler.
CT^s	Overhead of moving a single periodic task to the dispatch queue.

Figure 14 Schedulability test

(CS1 = 0.016 ms, CS2 = 0.012 ms, Tclk = 10 ms, CTc = 0.01 ms, CTs = 0.03 ms)

In the following sections, it is worth noting that term $''_x$ is measured experimentally: the maximum value of the execution time of the wrappers is measured *directly* from the target system, and assigned to $''_x$. This technique does not guarantee however that the so-obtained value derives from the worst-case scenario of execution, and hence that it effectively corresponds to the worst-case value. The same problem occurs today as far as the determination of worst-case overheads of operating systems is concerned [Colin & Puaut 2001]. Indeed, using techniques based on static code analysis in these cases is still an open problem that was out of the scope of our work.

3.5.2.2. Integration of Error Detection Only

When wrappers are used for error detection *only* (see Section 3.5.1.2), their maximum overhead appears in the absence of faults. Indeed, when an error is detected by a wrapper, the system (and so the wrapper) is usually stopped (fail-safe behavior); the overhead induced by the wrapper in this case is thus lower that if it had finished its execution.

First, we have measured experimentally in the absence of faults, the maximum overhead generated by the whole set of wrappers into a task instance. Such measures are reported in column *W (ALL)* of Table 10.

Tasks	W (*ALL*)	W (*ACT-TIM*)	W (*ACT-PCP*)
CH4 Sensor	18.16	2.22	7.47
CO Sensor	7.50	2.19	2.48
Air-Flow Sensor	7.51	2.21	2.48
Water-Flow Sensor	9.82	2.05	3.61
HLW Handler	9.66	1.50	3.73

Table 10 Maximum overhead of the wrappers in a task instance (ms)

Next, we have integrated such measurements into the schedulability test of Figure 14 and we have calculated the worst-case response time of the tasks, reported in column *R (ALL)* of Table 11.

T ches	D	R (*ALL*)	R (*ACT-TIM*)	R (*ACT-PCP*)
CH4 Sensor	30	***35.34***	19.38	24.64
CO Sensor	60	***72.76***	43.72	50.83
Air-Flow Sensor	100	***138.18***	55.97	63.35
Water-Flow Sensor	40	***55.21***	31.49	38.29
HLW Handler	200	***281.16***	77.53	131.81

Table 11 Worst-case response times of the tasks in the presence of the wrappers (ms)

According to the results, in the worst-case scenario, all tasks miss their deadline when the whole set of wrappers is used.

We have applied the same procedure to more restricted sets of wrappers, formed by the combination of the wrappers that were activated as a consequence of an error (namely, wrappers *Timer_1*, *Take_gainlock*, *Take_1*, *Give_owner* and *Give_1*, reported in Table 8). In other words, we have eliminated those wrappers that did not contributed to increase the error detection coverage, but that caused a considerably temporal overhead. According to our analyses, the sets of wrappers that satisfy the schedulability test, while maximizing the error detection coverage, are the following:

- Set ACT-TIM: Formed by the single activated wrapper checking temporization, namely wrapper Timer_1. The error detection coverage it provides in campaign sTIM is maintained (i.e., it is the same as that reported in Figure 12).
- Set ACT-PCP: Formed by the activated wrappers checking synchronization, namely wrappers Take_gainlock, Take_1, Give_owner and Give_1. The error detection coverage provided by this set in campaigns sSCH and pPCP is preserved (see Figure 12).

The temporal overheads of these sets (*W (ACT-TIM)* et *W (ACT-PCP)*) are reported in Table 10, and the response times of tasks in their presence (*R (ACT-TIM)* et *R (ACT-PCP)*) in Table 11.

Note that the wrappers are being integrated into a target system whose conception and development are not modified. By modifying some timing constraints of the system, for instance by increasing the deadline of task *Water-Flow Sensor* by only 5 ms, the set of wrappers resulting from the union of sets *ACT-TIM* and *ACT-PCP* satisfies the schedulability test.

3.5.2.3. Integration of Error Detection and Error Recovery

When wrappers are used for error detection and error recovery (see Section 3.5.1.3), their overhead depends on the number of errors they detect. Indeed, the greater the number of detected errors, the greater the number of executed recovery actions, and so the greater the overhead caused by the wrappers. Note that in this case, the fault model used is a key factor. Instead of using a theoretic fault model (like in [Burns *et al.* 1999]), we use the experimentally injected faults in Section 3.5.1 to analyze the behavior profile of the wrappers in the presence of faults.

First, we measured, for each campaign, the maximum overhead that *all* the executed recovery actions induced in a task instance (Table 12). Clearly, this overhead depends on the activation profile of the recovery actions in the presence of faults. Then, we have analyzed which of these activation profiles can be combined to the wrapper sets identified in the previous section (Table 13). The objective is that, despite the extra overhead imposed by the recovery actions, tasks deadlines be guaranteed and the error detection and recovery coverage of the wrappers be maximized. Taking into account the overheads reported in Table 12, the schedulability test is satisfied for the wrapper sets and the activated recovery actions presented in Table 13.

Tasks	sSCH	sTIM	pPCP
CH4 Sensor	2.51	0.02	0.16
CO Sensor	0.10	0.05	0.05
Air-Flow Sensor	0.10	0.03	0.05
Water-Flow Sensor	0.15	0.02	0.09
HLW Handler	0.13	0.01	0.07

Table 12 Temporal overheads induced by all the activated recovery actions (ms)

ACT-TIM + sTIM
ACT-PCP + pPCP
ACT-sSCH + sSCH

Table 13 Wrapper sets and activated recovery actions that satisfy the schedulability test

Despite the additional overheads generated by the recovery actions, wrapper set *ACT-TIM* guarantees the satisfaction of all the task's deadlines in campaign sTIM, while procuring the same ratios of error detection and recovery previously measured (see Figure 13). The same can be said when set *ACT-PCP* is used in campaign pPCP. Concerning campaign sSCH, it was necessary to define a more restricted wrapper set, namely *ACT-sSCH*, composed of wrappers *Take_gainlock*, *Take_1* and *Give_1*. Nevertheless, such a set provides the same error detection and recovery coverage previously observed in campaign sSCH (see Figure 13).

3.6. Discussion

3.6.1. General Discussion

In this chapter, we have shown that the conventional notion of error detection wrappers can be extended to the inclusion of recovery actions to constitute fault tolerance wrappers. We also showed that the temporal overhead of such wrappers can be made compatible with the hard deadlines of a target application, while maximizing error detection and recovery coverage.

The efficiency of the fault tolerance wrappers is due to the fact that they are developed from precise temporal logic specifications. At runtime, they can confidently diagnose the source of the problem that altered the state of the system and trigger the most appropriate recovery actions. The concurrent execution of the wrappers helps both check the system behavior from several angles and prevent the propagation of errors. The evaluation of the assertions carried out by the wrappers on a real-time microkernel relies on a reflective approach to capture events (notion of reification), to obtain internal data (notion of introspection), and to execute the recovery actions (notion of intercession).

From a dependability viewpoint, the results obtained in the case study by using fault injection show the real benefit procured by this new form of comprehensive wrappers. However, since wrappers are to be integrated into a real-time system, the temporal overheads they introduce must also be taken into account. Although this aspect can be addressed as early as possible in the design of the real-time application, necessary tradeoffs must be decided regarding the expected coverage procured by a set of wrappers and the performance overheads that can be accepted. This decision is left open to the real-time system designers. Regarding this aspect, we have provided an experimental methodology to support such an analysis that is based on fault injection to assess the impact of errors in the overheads of the wrappers, and to determine the set of wrappers that meets hard task deadlines while maximizing the error detection and recovery coverage.

The wrapping methodology presented in the previous chapters can be applied to a CORBA-based system at three levels: at the interface between the application and the middleware, at the interface between the middleware and the operating system, and at the interface between the middleware and the network. In each case, the wrappers mediate between the components interacting at the interface, ensuring that their respective expectations are met, even in the presence of faults. The types of properties that can be verified by the wrappers, and the techniques for checking and meeting the properties, are different in each case, and are discussed in the next three subsections.

3.6.2. Wrapping the Interface between the Application and the Middleware

In the same way as an operating system may fail when subjected to invalid parameters in a system call, a middleware implementation may fail if the application passes invalid parameters to a CORBA method call. Consider for example the `object_to_string` method that is defined by the CORBA specification, and which may be invoked in a C++ program as follows:

```
orb->object_to_string(obj)
```

Some middleware implementations crash when the object reference `obj` is invalid. In particular, the Ballista project has investigated the impact of this class of faults (that simulates primarily software faults) on a number of middleware implementations, and found a considerable number of robustness failures [Pan *et al.* 2001]. The authors found that the addition of simple wrappers protecting these methods was sufficient to improve their robustness with respect to these fault classes. The technique they used to implement the wrappers was intrusive, since it consisted of modifying the code of the middleware implementation to add error checking of the parameters to certain CORBA methods. However, similar forms of error checking could be implemented using less intrusive techniques, such as hooking into the symbol resolution mechanism used with shared libraries on Unix-like systems.

Unfortunately, this form of wrapping can only be applied to a limited subset of the functionality provided by an ORB, because the interface between application code and code from the middleware is in general difficult to identify. In a system running on a microkernel, there is a clear separation between user space and kernel space (often enforced with the help of the memory management hardware). Accordingly, the set of system calls implemented by the kernel constitutes an explicit interface between the two layers. In contrast, in a middleware-based system, application code is generally more closely intertwined with code from the middleware. In a CORBA-based middleware, for example, code that is automatically generated from OMG IDL interfaces (the stubs and skeletons, which control the process in which a programming language method call is transformed into a remote method invocation), is linked together with code written by the application programmer. The form of this automatically generated code depends both on the programming language used at the application level and on the choices made by the ORB implementor, which makes it difficult to isolate a specific interface where wrapping could be applied systematically. In consequence, any wrapping technique that operates at this level would be specific to a programming language and to a particular middleware implementation (and possibly a specific version).

3.6.3. Wrapping the Interface between the Middleware and the Operating System

A middleware implementation depends on services provided by the underlying operating system. Another The interface between the middleware and the operating system is a potential error propagation channel in a middleware-based system. If the operating system behaves in an incorrect or unexpected manner, the robustness of the system may be significantly impaired.

There are a number of ways in which errors in the operating system, or unexpected situations in the operating environment, may propagate to the middleware. The first is via the return code of a system call[15]. For instance, a write() system call used by the middleware to send data over a network stream returns a status code indicating the number of bytes that were actually written to the network, or an error code. In the nominal case, the number of bytes written is equal to the length of the argument to the system call, but the caller should also check for a partial write of

[15] In the following we use terminology for POSIX-like operating systems. The same ideas could be applied to most modern operating systems.

the data. The system call can also result in a number of different error codes, indicating that the network connection has been closed, or that a low-level error occurred, or that the system call was interrupted and should be retried. Other system calls for memory allocation or access to stable storage may fail due to exhaustion of system resources.

A robust middleware implementation should check the return codes from all system calls, and take appropriate measures for each possible return code. For instance, the failure of a write() system call to a network socket should be signaled to the application level by a COMM_FAILURE exception. An EINTR return code from a system call indicates that the system call was interrupted, and should be replayed.

Unexpected return codes are not the only error propagation channel from the operating system to the middleware. The kernel may also send unexpected signals to the middleware, causing it to fail, or omit to send signals that were expected by the middleware (for instance to inform it that a timer has expired). The kernel may also fail in the time domain, by not responding to requests within the time span expected by the application (for instance, when using the NFS distributed file system, when a network file server becomes inaccessible, and processes using files on the system will typically be blocked until the file server starts responding again). Other kernel services that can cause failure are the threading and synchronization primitives.

Preliminary results obtained by fault injection have shown that the behaviour of a CORBA implementation can be significantly disturbed when kernel functions are corrupted. For instance, when the operating system's memory resources are exhausted, the MICO implementation of the CORBA Event Service voluntarily aborts, instead of the more robust alternative of signalling a NO_MEMORY exception to the client. This is an illustration of how fault injection experiments can allow the identification of a specific software fault, which is easy to correct.

If a middleware implementation is being integrated into a DSoS as a COTS component, it is difficult to modify the middleware to correct these classes of robustness weaknesses. It is much easier to apply wrapping techniques to the operating system, in order to improve its failure modes and to force them to better match those expected by the middleware. The case study reported in this chapter demonstrates that wrapping of executive software components is able to ensure the validity of relatively sophisticated properties (such as the scheduling behaviour of threads); this technique can improve the operating system’s failure profile as perceived by the middleware, thus enhancing the overall robustness of the system.

3.6.4. Wrapping the Interface between the Middleware and Remote Objects

In [Marsden & Fabre 2001], we showed that CORBA middleware implementations are quite sensitive to corrupt messages arriving over the network, in certain cases crashing upon reception of an incorrectly formatted method invocation. While this interaction occurs via services provided by the operating system (reads and writes on socket streams), we distinguish this interface from the one addressed in the previous paragraph, because in the present case it is the data received via the system call that may cause a problem, rather than an error code returned by the system call.

Two forms of wrapping are possible to enhance the robustness of a DSoS using a CORBA-based communications infrastructure, with respect to these classes of faults:

- Add a checksum to all IIOP messages, allowing a wrapper to reject any messages where the checksum is incorrect. This type of wrapper can be implemented in a transparent way using the Portable Interceptor mechanism that was standardized in CORBA 2.4. The checksum is added to the service context (a section of the header of an IIOP message) by a client-side interceptor, and checked in the receiver by a server-side interceptor. If the checksum does not match the message contents, the server-side interceptor raises a COMM_FAILURE exception rather than propagating the message to the servant. However, this approach is not able to protect against malicious faults.

- Use an additional level of encapsulation at the network level, by using IIOP over SSL (*Secure Sockets Layer*) or TLS (*Transport Layer Security*). This provides protection from accidental faults, since the TLS decryption phase will detect transport-level corruption, and also from malicious faults, since messages whose signature is invalid would be rejected by the server-side interceptor. However, the performance overhead of this form of wrapper is non-negligible.

The implementation of wrappers between remote objects must be based on interception facilities that can be provided by metaobject protocols. Metaobjects controlling object interactions can thus be added to both client and server side, thus enabling enhanced assertions to be developed.

3.7. Conclusion

The above described wrapping technology enables software components to be adjusted from a non functional viewpoint to the expected assumptions made by companion components. This is the case for conventional middleware like CORBA that often make the assumption that underlying operating system facilities behave as specified. This is a strong assumption, even in the absence of faults, but this assumption has a very weak coverage in the presence of faults.

The work presented in this chapter aim at providing a generic framework and tools to develop additional error detection and recovery mechanisms that make a *Target Software Component* compliant with assumptions made by its users, who may be other system components in a system of systems. We believe that the ability to make simplifying assumptions on the failure modes of component systems greatly facilitates the process of building of a dependable system of systems, and that the use of wrapping technology, as presented in this chapter, is a particularly attractive way of obtaining well-encapsulated component systems, even in the more difficult case of COTS or legacy systems.

References

Chapter 1 - Dependable Composition of Web Services

[Abrial 1996] J. R. Abrial. The B Book – Assigning Programs to Meanings. *Cambridge University Press*. 1996.

[Benatallah *et al.* 2002] B. Benatallah, M. Dumas, M-C. Fauvet, and F.A. Rabhi. Towards patterns of Web Services composition. In Patterns an skeletons for parallel en distributed computing. Springer, 2002.

[BPML] A. Arkin. Business Process Modeling Language, BPML 1.0 Last Call Working Draft, 2002.

[Casati *et al.* 2001] F. Casati, M. Sayal, and M-C. Shan. Developing E-services for composing E-services. In Proc. Of CAISE'2001, LNCS 2068, pages 171-186, 2001.

[Fauvet *et al.* 2001] M-C. Fauvet, M. Dumas, B. Benatallah, and H-Y. Paik. Peer-to-peer traced execution of composite services. In proc. Of TES'2001, LNCS 2193, pages 103-117, 2001.

[Florescu *et al.* 2002] D. Florescu, A. Grunhagen, and D. Kossmann. XL: An XML language for web service specification and composition. In Proceedings of the WWW'02 Conference, 2002.

[Kuno *et al.* 2001] H. Kuno, M. Lemon, A. Karp, and D. Beringer. Conversations+Interfaces= Business logic. In Proc. Of TES' 2001, LNCS 2193, pages 30-43, 2001.

[Mikalsen *et al.* 2002] T. Mikalsen, S. Tai and I. Rouvellou. Transactional attitudes: Reliable composition of autonomous Web services. In proc. Of ISDN'2002, 2002.

[MS-NET] Microsoft. .NET. http://msdn.microsoft.com/net/.

[Narayanan & Mc Ilraith 2002] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In Proceedings of the WWW'02 Conference, 2002.

[OASIS-BTP] OASIS Committee Specification. Business Transaction Protocol, Version 1.0, 2002.

[OMG-WS] OMG. Corba Web Services. OMG TC Document orbos/2001-06-07. http://www.omg.org. 2001.

[Romanovsky *et al.* 2002] A. Romanovsky, P. Periorellis, and A.F. Zorzo. On Structuring Integrated Web Applications for Fault Tolerance. Technical Report 765, Department of Computing Science, University of Newcastle upon Tyne, 2002

[UDDI] UDDI. UDDI, Version 2.0, API Specification. Technical report, 2002. http://www.uddi.org.

[SUN-J2EE] Sun Microsystems Inc. Java 2 Platform, Enterprise Edition (J2EE). http://java.sun.com/j2ee/.

[W3C-SOAP] W3C. Soap version 1.2. Technical report, The World Wide Web Consortium, 2002. http://www.w3.org/2000/xp/Group/.

[W3C-THP] W3C. Tentative Protocol Part 1: White Paper, 2001.

[W3C-WSCL] W3C. Web services conversation language (WSCL), version 1.0. Technical report, The World Wide Web Consortium, 2002. http://www.w3.org/TR/wscl10/.

[W3C-WSDL] W3C. Web services description language (WSDL), version 1.1. Technical report, The World Wide Web Consortium, 2001. http://www.w3.org/TR/wsdl, Working draft version 1.2 available at http://www.w3.org/TR/2002/WD-wsdl12-20020709.

[W3C-XML] W3C. Second Edition of the Extensible Markup Language (XML). 1.0 Specification. W3C Recommendation. http://www.w3.org/TR/2000/REC-xml-2001006. 2000.

[WSCI] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawagushi, D. Orchard, S. Poligliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimeck. Web Service Choreography Interface 1.0.

[WSFL] F. Leymann. Web Services Flow Language (WSFL 1.0). IBM Software Group. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf. 2001.

[XLANG] S. Thatte. XLANG: Web Services for Business Process Design. Microsoft Corporation. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm. 2001.

[Xu *et al.* 1995] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. *Proceedings of the IEEE Symposium on Fault Tolerant Computing*. 1995.

[Yang & Papazoglou 2002] J. Yang and P. Papazoglou. Web component: A substrate for web service reuse and composition. In Proceedings of CAISE'02, pages 21-36, 2002

[Zorzo & Stroud 1999] A.F. Zorzo and R.J. Stroud. An Object-Oriented Framework for Dependable Multiparty Interactions. In proc. of Conf. on Object-Oriented Programming, Systems & Applications (OOPSLA'99), ACM Sigplan Notices, 34(10), pages 435-446, 1999.

Chapter 2 – Structured Handling of On-Line Interface Upgrades in Integrating Dependable SoSs

[Amann *et al.* 2000] B. Amann, I. Fundulaki, M.Scholl. Integrating ontologies and thesauri for RDF schema creation and metadata querying. International Journal of Digital Libraries, 3, 3, pp. 221–236, 2000.

[Cristian 1995] F. Cristian. Exception Handling and Tolerance of Software Faults. In Lyu, M.R. (ed.): Software Fault Tolerance. Wiley, pp. 81-107, 1995.

[Hruska & Hashimoto 2000] T. Hruska and H. Hashimoto (eds), Knowledge Based Software Engineering, Ios Press June 2000.

[Laprie 1995] J.-C. Laprie. Dependable Computing: Concepts, Limits, Challenges. Proc. of the 25th Int. Symposium On Fault-Tolerant Computing. IEEE CS Press. Pasadena, CA. pp. 42-54. 1995.

[Periorellis & Dobson 2001] P. Periorellis, J.E. Dobson. Case Study Problem Analysis. The Travel Agency Problem. Technical Deliverable. Dependable Systems of Systems Project (IST-1999-11585). University of Newcastle upon Tyne. UK. 37 p. 2001. www.newcastle.research.ec.org/dsos/

[Romanovsky & Smith 2002] A. Romanovsky, I. Smith. Dependable On-line Upgrading of Distributed Systems. In Proc. of COMPSAC 2002. 26-29 August 2002, Oxford, UK. IEEE CS Press. pp. 975-976. 2002.

[Szyperski 1997] C. Szyperski. Component Software. ACM Press. 1997.

[Tai *et al.* 2002] A.T. Tai, K.S. Tso, L. Alkalai, S.N. Chau, W.H. Sanders. Low-Cost Error Containment and Recovery for Onboard Guarded Software Upgrading and Beyond. IEEE TC-51, 2, pp. 121-137. 2002.

[Vanderperren 2002] W. Vanderperren. A Pattern Based Approach to Separate Tangled Concerns in Component Based Development. Proc. of the 1st AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, held in conjunction with the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002). pp. 71-75. 2002.

[Welch 2002] I. Welch. A Reflective Security Architecture for Applications. PhD Thesis. Department of Computing, University of Newcastle upon Tyne (in preparation).

[W3C-RDF 2000] W3C. Resource Description Framework (RDF). RDF Specification Development. 2000. http://www.w3.org/RDF/.

[W3C-WSCL 2002] W3C. Web services conversation language (WSCL), version 1.0. The World Wide Web Consortium, 2002. http://www.w3.org/TR/wscl10/.

[W3C-WSDL 2001] W3C. Web services description language (WSDL), version 1.1. The World Wide Web Consortium, 2001. http://www.w3.org/TR/wsdl, Working draft version 1.2 http://www.w3.org/TR/2002/WD-wsdl12-20020709

Chapter 3 – From Error Detection to Recovery Wrappers

[Arlat *et al.* 2002] J. Arlat, J.-C. Fabre, M. Rodríguez , F. Salles, "Dependability of COTS Microkernel-Based Systems", *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 138-163, 2002.

[Burns & Lister 1991] A. Burns , A. M. Lister, "A Framework for Building Dependable Systems", *The Computer Journal*, vol. 34, no. 2, pp. 173-181, 1991.

[Burns *et al.* 1999] A. Burns, S. Punnekkat, L. Strigini , D. R. Wright, "Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems", in Proc. *7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA'99)*, San Jose, CA (USA), pp. 361-378, 1999.

[Burns & Wellings 1997] A. Burns, A. J. Wellings, *Real-time Systems and their Programming Languages*, Addison Wesley, 1997.

[Carreira *et al.* 1998] J. Carreira, H. Madeira , J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125-136, 1998.

[Chorus 1997] Chorus Systems, "CHORUS/ClassiX release 3 - Technical Overview", Technical Report no. CS/TR-96-119.12, Chorus Systems, 1997 (www.sun.com/chorusos).

[Colin & Puaut 2001] A. Colin , I. Puaut, "Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System", in Proc. *13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, pp. 191-198, 2001.

[Diaz *et al.* 1994] M. Diaz, G. Juanole , J.-P. Courtiat, "Observer--A Concept for Formal On-Line Validation of Distributed Systems", *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 900-913, 1994.

[Fabre & Pérennou 1998] J.-C. Fabre , T. Pérennou, "A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach", *IEEE Transactions on Computers, Special Issue on Dependability of Computing Systems*, pp. 78-95, 1998.

[Ghosh *et al.* 1999] A. K. Ghosh, M. Schmid , F. Hill, "Wrapping Windows NT Sofware for Robustness", in Proc. *29th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-29)*, Madison, WI (USA), pp. 344-347, 1999.

[Joseph 1996] M. Joseph, *Real-Time Systems: Specification, Verification and Analysis*, Prentice-Hall, 1996.

[Kiczales *et al.* 1991] G. Kiczales, J. d. Rivières , D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Koopman & DeVale 1999] P. J. Koopman , J. DeVale, "Comparing the Robustness of POSIX Operating Systems", in Proc. *29th IEEE International Symposium on Fault-Tolerant Computing (FTCS-29)*, Madison, WI (USA), pp. 30-37, 1999.

[Maes 1987] P. Maes, “Concepts and Experiments in Computational Reflection”, in Proc. *OOPSLA'87*, Orlando, FL (USA), pp. 147-155, 1987.

[Marsden & Fabre 2001] “Failure analysis of an ORB in presence of faults”, DSoS deliverable IC3, 2001.

[Mok & Liu 1997] A. K. Mok , G. Liu, “Efficient Run-Time Monitoring of Timing Constraints”, in Proc. *3rd Real-Time Technology and Applications Symposium*, Montral, Canada, 1997.

[Pan *et al.* 2001] J. Pan, P. Koopman, D. Siewiorek, Y. Huang, R. Gruber , M. Jiang, “Robustness Testing and Hardening of CORBA ORB Implementations”, in Proc. *IEEE International Conference on Dependable Systems and Networks (DSN 2001)*, Goteborg (Sweden), pp. 141-150, 2001.

[Rodríguez *et al.* 2002a] M. Rodríguez, A. Albinet , J. Arlat, “MAFALDA-RT: A Tool for Dependability Assessment of Real-Time Systems”, in Proc. *IEEE International Conference on Dependable Systems and Networks (DSN 2002)*, Washington DC (USA) (to appear), 2002a.

[Rodríguez *et al.* 2000] M. Rodríguez, J.-C. Fabre , J. Arlat, “Formal Specification for Building Robust Real-time Microkernels”, in Proc. *21st IEEE Real-Time Systems Symposium (RTSS 2000)*, Orlando, Florida (USA), pp. 119-128, 2000.

[Rodríguez *et al.* 2002b] M. Rodríguez, J.-C. Fabre , J. Arlat, “Wrapping Real-Time Systems from Temporal Logic Specifications”, in Proc. *4th European Dependable Computing Conference (EDCC-4)* , Oct. 2002, Toulouse (France), 2002b (to appear).

[Savor & Seviora 1997] T. Savor , R. E. Seviora, “An Approach to Automatic Detection of Software Failures in Real-Time Systems.”, in Proc. *IEEE Real-Time Technology and Applications Symposium*, pp. 136-146, 1997.

[Schneider 1998] F. Schneider, “Enforceable Security Policies”, no. TR98-1664, Department of Computer Science, Cornell University, Ithaca, NY (USA), 1998.

[Voas 1998] J. M. Voas, “Certifying Off-the-Shelf Software Components”, *Computer*, pp. 53-59, 1998.

Appendix A. Formalisation of Coordinated Atomic Actions in B

Ferda Tartanoglu, Nicole Levy, Valérie Issarny *(INRIA)*

This part of the deliverable introduces a formal specification of Coordinated Atomic (CA) actions [Xu *et al.* 1995] and of their composition introduced in Chapter 1. The formalisation allows for rigorous reasoning about dependable behavior of the systems of systems integrated using this structuring paradigm.

A.1. The B Formal Method

We have used the B formal method [Abrial 1996] which is a model-based method built on set theory and predicate logic and extended by generalised substitutions. Specifications in B are represented by abstract machines encapsulating operations and states.

Generally speaking, the B method allows us to write abstract machines and refinements over them. At the end of the refinement process an implementation can be written that corresponds to an executable code. However, in our model, we have only developed the initial abstract machines.

Proofs are an essential part of the model: the idea is to prove that all operations preserve the invariants of the machine and that the implementations and refinements preserve the invariants and the behaviour of the initial abstract machine.

There are various tools that help writing and proving B specifications. The main of them are B-Tool [B-Core] and Atelier B [Atelier B]. Both tools include a type checker, an animator, a proof obligation generator, theorem provers, code translators and documentation facilities. Atelier B has been used in our investigation, however the notation we used is compatible with both of them.

A.2. Modeling CA Actions

The system is modelled by four abstract machines (Figure A.1). The first of which is the CONSTANTS machine which includes common constants used by all other machines which access the CONSTANTS machine by using the SEES clause.

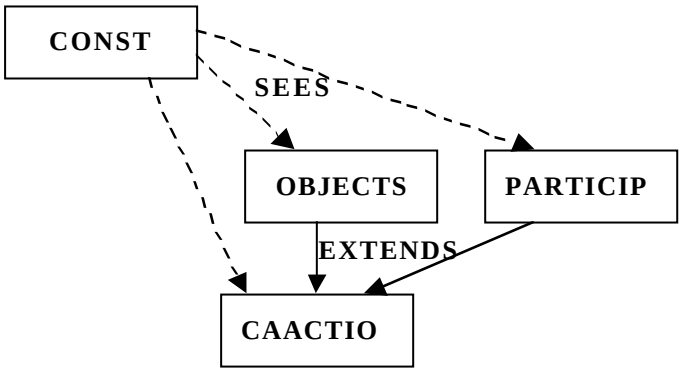


Figure A.1 Abstract machines

MACHINE
CONST
SETS
PARTICIPANT;
CAACTION_STATE = {*caa_normal*, *caa_exceptional*};
PARTICIPANT_STATE;
CAACTION;
OBJECT;
VALUE
CONSTANTS
init_val, *begin_val*, *commit_val*, *abort_val*,
normal, *waiting*, *exceptional*, *EXCEPTIONAL_STATE*, *compute_exception*
PROPERTIES
$init\_val \in VALUE \wedge$
$begin\_val \in VALUE \wedge$
$commit\_val \in VALUE \wedge$
$abort\_val \in VALUE \wedge$
$normal \in PARTICIPANT\_STATE \wedge$
$waiting \in PARTICIPANT\_STATE \wedge$
$EXCEPTIONAL\_STATE \subseteq PARTICIPANT\_STATE \wedge$
$exceptional \in PARTICIPANT\_STATE \wedge$
$exceptional \in EXCEPTIONAL\_STATE \wedge$
$compute\_exception : (PARTICIPANT\_STATE \times \mathrm{P}(PARTICIPANT\_STATE)) \rightarrow EXCEPTIONAL\_STATE$
END

Other abstract machines are PARTICIPANTS, OBJECTS and CAACTIONS. The CAACTIONS abstract machine extends (with the EXTENDS clause) the other two. Modularisation of this kind makes syntax simpler and facilitates proofs of the obligations.

The PARTICIPANTS abstract machine includes all operations related to action participants including their activations, state changes or removals.

MACHINE
PARTICIPANTS
SEES
CONST
VARIABLES
participant, *participant_state*, *participant_value*, *initial_values*
INVARIANT
/* active participants */
$participant \subseteq PARTICIPANT \wedge$

/* associates to each participant its state (normal, exceptional or waiting) */
$participant\_state \in PARTICIPANT \rightarrow \mathbf{seq}(PARTICIPANT\_STATE) \wedge$

/*associates a value to each active participant */
$participant\_value \in PARTICIPANT \rightarrow VALUE \wedge$

/* memorize the initial values of participants for recovery */
$initial\_values \in PARTICIPANT \rightarrow \mathbf{seq}(VALUE) \wedge$

/* CONSTRAINTS */
dom(*participant_state*) = *participant* ∧
dom(*participant_value*) = *participant* ∧
dom(*initial_values*) = *participant*
INITIALISATION
participant := ∅ ||
participant_state := ∅ ||
participant_value := ∅ ||
initial_values := ∅
OPERATIONS
add_new_participants(*epar*) =
PRE
epar ⊆ *PARTICIPANT* ∧
epar ∩ *participant* = ∅
THEN
participant := *participant* ∪ *epar* ||
participant_state := *participant_state* ⩤ *epar*×{[*normal*]} ||
participant_value := *participant_value* ∪ *epar*×{*init_val*} ||
initial_values := *initial_values* ∪ *epar*×{[*init_val*]}
END;
add_nested_participants(*epar*) =
PRE
epar ⊆ *PARTICIPANT* ∧
epar ⊆ *participant*
THEN
participant_state := *participant_state* ⩤ { *pa*, *sps* | *pa*∈ *epar* ∧ *pa*∈ **dom**(*participant_state*) ∧
sps∈ **seq**(*PARTICIPANT_STATE*) ∧
sps= **participant_state**(*pa*) ← *normal* } ||
initial_values := *initial_values* ⩤ { *pa*, *sval* | *pa*∈ *epar* ∧ *pa*∈ **dom**(*participant_value*) ∧
sval∈ **seq**(*VALUE*) ∧
sval= **initial_values**(*pa*) ← *init_val* }
END;
add_composed_participants(*pa*, *epar*) =
PRE
pa∈ *participant* ∧
epar ⊆ *PARTICIPANT* ∧
epar ∩ *participant* = ∅
THEN
participant := *participant* ∪ *epar* ||
participant_state := (*participant_state* ∪ *epar*×{[*normal*]})
⩤ {*pa* ↦ (*participant_state*(*pa*) ← *waiting*)} ||
participant_value := *participant_value* ∪ *epar*×{*init_val*} ||
initial_values := *initial_values* ∪ *epar*×{[*init_val*]}
END;
set_participant_value(*pa*, *val*) =
PRE
pa∈ *participant* ∧
val∈ *VALUE* ∧
pa ∈ **dom**(*participant_value*)
THEN
participant_value := *participant_value* ⩤ {*pa* ↦ *val*}
END;
set_participants_state (*epar*, *stat*) =

PRE
epar ⊆ *participant* ∧
stat ∈ *PARTICIPANT_STATE* ∧
epar ⊆ **dom**(*participant_state*)
THEN
participant_state := *participant_state* ⩤ { *pa*, *sps* | *pa*∈ *epar* ∧ *sps*∈ **seq**(*PARTICIPANT_STATE*) ∧
sps= **front**(*participant_state*(*pa*)) ← **stat** }
END;
remove_update_participants_states(*epar*, *stat*) =
PRE
epar ⊆ *participant* ∧
stat ∈ *PARTICIPANT_STATE* ∧
epar ⊆ **dom**(*participant_state*) ∧
epar ⊆ **dom**(*initial_values*)
THEN
participant_state := *participant_state* ⩤ { *pa*, *sps* | *pa*∈ *epar* ∧ *sps*∈ **seq**(*PARTICIPANT_STATE*) ∧
sps= **front**(**front**(*participant_state*(*pa*))) ← **stat** } ||
initial_values := *initial_values*
⩤ { *pa*, *sval* | *pa*∈ *epar* ∧ *sval*∈ **seq**(*VALUE*) ∧ *sval*= **front**(*initial_values*(*pa*)) }
END;
delete_participants(*epar*) =
PRE
epar ⊆ *participant* ∧
epar ⊆ **dom**(*participant_state*) ∧
epar ⊆ **dom**(*participant_value*) ∧
epar ⊆ **dom**(*initial_values*)
THEN
participant := *participant* - *epar* ||
participant_state := *epar* ⩤ *participant_state* ||
participant_value := *epar* ⩤ *participant_value* ||
initial_values := *epar* ⩤ *initial_values*
END;
remove_composed_participants(*epar*, *pa*) =
/* set pa state to its previous state */
PRE
epar ⊆ *participant* ∧
pa∈ *participant* ∧
pa ∉ *epar* ∧
epar ⊆ **dom**(*participant_state*)∧
epar ⊆ **dom**(*participant_value*) ∧
epar ⊆ **dom**(*initial_values*)
THEN
participant := *participant* - *epar* ||
participant_state := (*epar* ⩤ *participant_state*) ⩤ {*pa* ↦ **front**(*participant_state*(*pa*)) } ||
participant_value := *epar* ⩤ *participant_value* ||
initial_values := *epar* ⩤ *initial_values*
END;
remove_composed_participants_exceptional(*epar*, *pa*) =
/* remove previous state of pa and set it to exceptional */
PRE
epar ⊆ *participant* ∧
pa∈ *participant* ∧
pa ∉ *epar* ∧

epar ⊆ **dom**(*participant_state*) ∧
epar ⊆ **dom**(*participant_value*) ∧
epar ⊆ **dom**(*initial_values*)

THEN

participant := *participant* - *epar* ||
participant_state := (*epar* ⩤ *participant_state*)
⩤ {*pa* ↦ (**front**(**front**(*participant_state*(*pa*))) ← *exceptional*) } ||
participant_value := *epar* ⩤ *participant_value* ||
initial_values := *epar* ⩤ *initial_values*

END;

remove_participant_state(*epar*) =

PRE

epar ⊆ *participant* ∧
epar ⊆ **dom**(*participant_state*) ∧
epar ⊆ **dom**(*initial_values*)

THEN

participant_state := *participant_state* ⩤ { *pa*, *sps* | *pa*∈ *epar* ∧ *sps*∈ **seq**(*PARTICIPANT_STATE*) ∧
sps= **front**(*participant_state*(*pa*))} ||
initial_values := *initial_values* ⩤ { *pa*, *sval* | *pa*∈ *epar* ∧ *sval*∈ **seq**(*VALUE*) ∧
sval= **front**(*initial_values*(*pa*))}

END;

remove_participant_state_and_value(*epar*) =

PRE

epar ⊆ *participant* ∧
epar ⊆ **dom**(*participant_state*) ∧
epar ⊆ **dom**(*participant_value*) ∧
epar ⊆ **dom**(*initial_values*)

THEN

participant_state := *participant_state* ⩤ { *pa*, *sps* | *pa*∈ *epar*∧ *sps*∈ **seq**(*PARTICIPANT_STATE*) ∧
sps= **front**(*participant_state*(*pa*))} ||
participant_value := *participant_value* ⩤ { *pa*, *val* | *pa*∈ *epar* ∧ *val*∈ *VALUE* ∧
val= **last**(*initial_values*(*pa*))} ||
/* participants recover their initial values */
initial_values := *initial_values*⩤ { *pa*, *sval* | *pa*∈ *epar* ∧ *sval*∈ **seq**(*VALUE*) ∧
sval= **front**(*initial_values*(*pa*))}

END

END

The OBJECTS machine specifies the operations on external objects including the transactional operations that these objects export.

MACHINE

OBJECTS

SEES

CONST

VARIABLES

values, *object*

INVARIANT

/* any object has a value at any time */
values ∈ *OBJECT* → *VALUE* ∧
/* external objects associated to a CA action */
object ⊆ *OBJECT*

INITIALISATION

values := *OBJECT* × {*init_val*} ||

object := ∅
OPERATIONS
val ← **read_object**(*obj*) =
PRE
obj ∈ *object*
THEN
val := *values*(*obj*)
END;
write_object(*obj*,*val*) =
PRE
obj ∈ *object* ∧
val ∈ *VALUE* ∧
obj ∈ **dom**(*values*)
THEN
values(*obj*) := *val*
END;
/* begin transaction on objects */
add_objects(*objs*) =
PRE
objs ⊆ *OBJECT* ∧
objs ∩ *object* = ∅
THEN
values := *values* ⩤ *objs*×{*begin_val*} ||
object := *object* ∪ *objs*
END;
terminate_transaction(*objs*,*val*) =
PRE
objs ⊆ *object* ∧
val ∈ *VALUE*
THEN
values := *values* ⩤ *objs*×{*val*} ||
object := *object* - *objs*
END;
terminate_nested_transaction(*objs*,*val*) =
PRE
objs ⊆ *object* ∧
val ∈ *VALUE*
THEN
values := *values* ⩤ *objs*×{*val*}
END
END

The CAACTIONS abstract machine includes operations used for creating and terminating CA actions, action nesting and composing, and also the exceptional state operations.

MACHINE
CAAction
SEES
CONST
EXTENDS
PARTICIPANTS,
OBJECTS
VARIABLES
caaction, *caaction_state*, *caaction_particip*, *particip_caaction*,

caaction_ext_objects, is_nested, is_composed

INVARIANT

/* TYPES */

/* active CA actions */
caaction $\subseteq$ *CAACTION* $\wedge$

/* associates to each active CA action its state (normal, exceptional) */
caaction_state $\in$ *caaction* $\rightarrow$ *CAACTION_STATE* $\wedge$

/* associates to each active aprticipant the sequence of nested CA actiond to which it participates */
caaction_particip $\in$ *participant* $\rightarrow$ **seq**(*caaction*) $\wedge$

/* associates to each CA action its participants */
particip_caaction $\in$ *caaction* $\rightarrow$ P (*participant*) $\wedge$

/* external object accessed from a CA action */
caaction_ext_objects $\in$ *caaction* $\leftrightarrow$ *object* $\wedge$

/* (caa1 |-> caa2) : caa1 is nested in caa2 */
is_nested $\in$ *caaction* $\leftrightarrow$ *caaction* $\wedge$

/* (pa |-> caa) : participant pa is waiting for the composed CA action caa */
is_composed $\in$ *participant* $\nrightarrow$ *caaction* $\wedge$

/* CONSTRAINTS */

/* any participant active in a CA action can be active in another CA action
only if this latter is nested in the former */
$\forall$(*pa*,*caa1*, *caa2*).((*pa* $\in$ *participant* $\wedge$ *caa1* $\in$ *caaction* $\wedge$ *caa2* $\in$ *caaction* $\wedge$
{*caa1*,*caa2*} $\subseteq$ **ran**(*caaction_particip*(*pa*)))
$\Rightarrow$ ((*caa1*,*caa2*) $\in$ *is_nested* $\vee$ (*caa2*,*caa1*) $\in$ *is_nested*)) $\wedge$

/* any CA action nested in another CA action cannot have external participants */
$\forall$(*caa1*, *caa2*).((*caa1* $\in$ *caaction* $\wedge$ *caa2* $\in$ *caaction* $\wedge$ (*caa1*,*caa2*) $\in$ *is_nested*)
$\Rightarrow$ *particip_caaction*(*caa1*) $\subseteq$ *particip_caaction*(*caa2*)) $\wedge$

/* any external object accessed from two CA actions implies they are nested one another */
$\forall$(*obj*,*caa1*,*caa2*).((*obj* $\in$ *object* $\wedge$ *caa1* $\in$ *caaction* $\wedge$ *caa2* $\in$ *caaction* $\wedge$
obj $\in$ *caaction_ext_objects*[{*caa1*}] $\cap$ *caaction_ext_objects*[{*caa2*}])
$\Rightarrow$ ((*caa1*,*caa2*) $\in$ *is_nested* $\vee$ (*caa2*,*caa1*) $\in$ *is_nested*)) $\wedge$

/* relation between caaction_particip and particip_caaction */
$\forall$(*pa*).(*pa* $\in$ *participant*
$\Rightarrow$ **ran**(*caaction_particip*(*pa*))={*caa*| *caa* $\in$ *caaction* $\wedge$ *pa* $\in$ *particip_caaction*(*caa*)}) $\wedge$

/* a CA action can only be in an exceptional state if all of its participants are also in the same state
or a participant may be waiting and in this case, before composing a CA action, the participant was
in an exceptional state */
$\forall$(*caa*). (*caa* $\in$ *caaction* $\wedge$ *caaction_state*(*caa*)= *caa_exceptional*
$\Rightarrow$ $\forall$(*pa*).(*pa* $\in$ *particip_caaction*(*caa*)
$\Rightarrow$ ((**last**(*participant_state*(*pa*)) $\in$ *EXCEPTIONAL_STATE*) $\vee$
((**last**(*participant_state*(*pa*)) = *waiting* $\wedge$
last(**front**(*participant_state*(*pa*))) $\in$

EXCEPTIONAL_STATE))))) ∧

/* a composed CA action is not nested in another CA action */
∀(*caa*). (*caa*∈ *caaction* ∧ *caa* ∈ **ran**(*is_composed*) ⇒ *caa* ∉ **dom**(*is_nested*)) ∧

/* (pa |-> caa) : is_composed implies that pa is in a waiting state
and to be in such a state means that pa is waiting for a composed CA action */
∀(*pa*,*caa*).(*pa*∈ *participant* ∧ *caa*∈ *caaction* ∧ (*pa* ↦ *caa*) ∈ *is_composed*
⇒ **last**(*participant_state*(*pa*)) = *waiting*) ∧

∀(*pa*).(*pa*∈ *participant* ∧ *pa* ∉ **dom**(*is_composed*) ⇒ **last**(*participant_state*(*pa*)) ≠ *waiting*)

INITIALISATION
caaction := ∅ ||
caaction_state := ∅ ||
caaction_particip := ∅ ||
particip_caaction := ∅ ||
caaction_ext_objects := ∅ ||
is_nested := ∅ ||
is_composed := ∅
OPERATIONS
/*** CREATE CA ACTIONS **********/
create_external_caaction(*caa*, *epar*, *exto*) =
PRE
caa∈ *CAACTION* ∧
caa ∉ *caaction* ∧
epar ⊆ *PARTICIPANT* ∧
epar ∩ *participant* =∅ ∧
exto ⊆ *OBJECT* ∧
exto ∩ *object* = ∅
THEN
caaction := *caaction* ∪ {*caa*} ||
caaction_state := *caaction_state* ∪ {*caa* ↦ *caa_normal*} ||
caaction_particip := *caaction_particip* ∪ *epar*×{[*caa*]} ||
particip_caaction := *particip_caaction* ∪ {*caa* ↦ *epar*} ||
add_new_participants(*epar*) ||
caaction_ext_objects := *caaction_ext_objects* ∪ {*caa*}×*exto* ||
add_objects(*exto*)
END;
create_nested_caaction(*caa1*, *caa2*, *epar*, *exto*) =
PRE
caa1∈ *CAACTION* ∧
caa2∈ *caaction* ∧
epar ⊆ *participant* ∧
exto ⊆ *object* ∧
epar ⊆ *particip_caaction*(*caa2*) ∧
caa1∉ *caaction* ∧
/* external objects of nested CA action is a subset of external objects of the containing CA action */
∀*obj*.(*obj*∈ *exto* ⇒ *obj*∈ *caaction_ext_objects*[{*caa2*}]) ∧
/* all participants to caa1 are in the same state which is not waiting */
card(**ran**({ *pa*, *stat* | *pa*∈ *epar* ∧ *state*∈ *PARTICIPANT_STATE* ∧
stat= **last**(*participant_state*(*pa*))})) = 1 ∧
∀(*pa*).(*pa*∈ *epar* ⇒ **last**(*participant_state*(*pa*)) ≠ *waiting*) ∧

/* all participants of the nested CA action are in the same containing CA action */
card(**ran**({ *pa, ca* | *pa*∈ *epar* ∧ *ca*∈ *CAACTION* ∧ *ca*= **last**(*caaction_particip*(*pa*)) })) = 1
THEN
caaction := *caaction* ∪ {*caa1*} ||
caaction_state := *caaction_state* ∪ {*caa1* ↦ *caa_normal*} ||
caaction_particip := *caaction_particip*
⩤ { *pa, scaa* | *pa*∈ *epar* ∧ *scaa*∈ **seq**(*caaction*) ∧ *scaa*= **caaction_particip**(*pa*) ← *caa1* } ||
particip_caaction := *particip_caaction* ∪ {*caa1* ↦ *epar*} ||
caaction_ext_objects := *caaction_ext_objects* ∪ {*caa1*}×*exto* ||
is_nested := *is_nested* ∪ {*caa1* ↦ *caa2*} ||
add_nested_participants(*epar*)
END;
create_composed_caaction(*pa, caa, epar, exto*) =
PRE
pa∈ *participant* ∧
caa∈ *CAACTION* ∧
caa ∉ *caaction* ∧
epar ⊆ *PARTICIPANT* ∧
epar ∩ *participant* =∅ ∧
exto ⊆ *OBJECT* ∧
exto ∩ *object* = ∅
THEN
caaction := *caaction* ∪ {*caa*} ||
caaction_state := *caaction_state* ∪ {*caa* ↦ *caa_normal*} ||
caaction_particip := *caaction_particip* ∪ *epar*×{[*caa*]} ||
particip_caaction := *particip_caaction* ∪ {*caa* ↦ *epar*} ||
caaction_ext_objects := *caaction_ext_objects* ∪ {*caa*}×*exto* ||
is_composed := *is_composed* ∪ {*pa* ↦ *caa*} ||
add_composed_participants(*pa, epar*) ||
add_objects(*exto*)
END;
/*** NORMAL MODE OPERATIONS **********/
send_message(*pa1, pa2, val*) =
PRE
pa1∈ *participant* ∧
pa2∈ *participant* ∧
val∈ *VALUE* ∧
/* the seqence of CA actions must be the same : messages between
participants of the same CA action */
caaction_particip(*pa1*) = *caaction_particip*(*pa2*) ∧
last(*participant_state*(*pa1*)) = **last**(*participant_state*(*pa2*)) ∧
last(*participant_state*(*pa1*)) ≠ *waiting* ∧
last(*participant_state*(*pa2*)) ≠ *waiting*
THEN
set_participant_value(*pa2,val*)
END;
read_value ← **read_ext_object**(*pa,eobj*) =
PRE
pa∈ *participant* ∧
eobj ∈ *object* ∧
eobj∈ *caaction_ext_objects*[{ **last**(*caaction_particip*(*pa*))}]
THEN

read_value ← **read_object**(*eobj*)
END;
write_ext_object(*pa*,*eobj*,*funct*) =
PRE
pa∈ *participant* ∧
eobj ∈ *object* ∧
funct ∈ *VALUE* → *VALUE* ∧
eobj∈ *caaction_ext_objects*[{ **last**(*caaction_particip*(*pa*)) }]
THEN
write_object(*eobj*, *funct*(*values*(*eobj*)))
END;
/******** EXCEPTION RAISE **********/
raise_exception (*pa*, *excep*) =
PRE
pa∈ *participant* ∧
excep∈ *EXCEPTIONAL_STATE* ∧
last(*participant_state*(*pa*)) = *normal* ∧
caaction_state(**last**(*caaction_particip*(*pa*))) = *caa_normal*
THEN
set_participants_state({*pa*}, *excep*)
END;
/* propagate exception to all participants */
propagate_exception(*pa*) =
PRE
pa∈ *participant* ∧
last(*participant_state*(*pa*)) ∈ *EXCEPTIONAL_STATE* ∧
caaction_state(**last**(*caaction_particip*(*pa*))) = *caa_normal* ∧
∀*pacaa*.(*pacaa* ∈ *particip_caaction*(**last**(*caaction_particip*(*pa*)))
⇒ **last**(*participant_state*(*pacaa*)) ≠ *waiting*) ∧
/* all participants must be in the same CA action */
card(**ran**({*paa*,*caa* | *paa* ∈ *particip_caaction*(**last**(*caaction_particip*(*pa*))) ∧
caa ∈ *CAACTION* ∧ *caa* = **last**(*caaction_particip*(*paa*)) })) = 1
THEN
LET *pasost* **BE**
pasost = { *paac*, *sta* | *paac*∈ *particip_caaction*(**last**(*caaction_particip*(*pa*))) ∧
sta∈ *PARTICIPANT_STATE* ∧ *sta*= **last**(*participant_state*(*paac*))
}
IN
LET *stat* **BE**
stat = *compute_exception*(**last**(*participant_state*(*pa*)),**ran**(*pasost*))
IN
set_participants_state
(*particip_caaction*(**last**(*caaction_particip*(*pa*))), *stat*) ||
caaction_state(**last**(*caaction_particip*(*pa*))) := *caa_exceptional*
END
END
END;
/********** TERMINATE IN NORMAL STATE **********/
terminate_caaction(*caa*) =
PRE
caa∈ *caaction* ∧
caa ∉ **dom**(*is_nested*) ∧
caa ∉ **ran**(*is_nested*) ∧

caa ∉ **ran**(*is_composed*) ∧
caaction_state(*caa*) = *caa_normal* ∧
/*all participants must be in a normal state */
∀(*pa*).(*pa*∈ *particip_caaction*(*caa*) ⇒ **last**(*participant_state*(*pa*)) = *normal*)

THEN
LET *epar* **BE**
epar =*particip_caaction*(*caa*)
IN
/* commit and remove objects */
terminate_transaction(*caaction_ext_objects*[{*caa*}], *commit_val*) ||
delete_participants(*epar*) ||
caaction := *caaction* - {*caa*} ||
caaction_state := {*caa*} ⩤ *caaction_state* ||
caaction_particip := *epar* ⩤ *caaction_particip* ||
particip_caaction := {*caa*} ⩤ *particip_caaction* ||
caaction_ext_objects := {*caa*} ⩤ *caaction_ext_objects*
END
END;
terminate_nested_caaction(*caa*) =
PRE
caa∈ *caaction* ∧
caa∈ **dom**(*is_nested*) ∧
caa ∉ **ran**(*is_nested*) ∧
caaction_state(*caa*) = *caa_normal* ∧
/*all participants must be in normal state */
∀(*pa*).(*pa*∈ *particip_caaction*(*caa*) ⇒ **last**(*participant_state*(*pa*)) = *normal*)
THEN
LET *epar* **BE**
epar =*particip_caaction*(*caa*)
IN
/* all external objects are commited */
terminate_nested_transaction(*caaction_ext_objects*[{*caa*}], *commit_val*) ||
/* the participants recover their initial state */
remove_participant_state(*epar*) ||
caaction := *caaction* - {*caa*} ||
caaction_state := {*caa*} ⩤ *caaction_state* ||
caaction_particip := *caaction_particip*
⩤ { *pa*, *scaa* | *pa*∈ *epar* ∧ *scaa*∈ **seq**(*caaction*) ∧
scaa= **front**(*caaction_particip*(*pa*)) } ||
particip_caaction := {*caa*} ⩤ *particip_caaction* ||
caaction_ext_objects := {*caa*} ⩤ *caaction_ext_objects*
END
END;
terminate_composed_caaction (*caa*, *pa*) =
PRE
caa∈ *caaction* ∧
pa∈ *participant* ∧
pa ∉ *particip_caaction*(*caa*) ∧
pa∈ **dom**(*is_composed*) ∧
caa = *is_composed*(*pa*) ∧
caa ∉ **dom**(*is_nested*) ∧
caa ∉ **ran**(*is_nested*) ∧
caaction_state(*caa*) = *caa_normal* ∧

```
            /*all participants must be in a normal state */
          ∀(pacaa).(pacaa∈ particip_caaction(caa) ⇒ last(participant_state(pacaa)) = normal )
   THEN
          LET epar BE
                epar = particip_caaction(caa)
          IN
                  /* all external objects are commited */
                terminate_transaction(caaction_ext_objects[{caa}],commit_val) ||
                  /* set to previous state */
                remove_composed_participants (epar, pa) ||
                caaction := caaction - {caa} ||
                caaction_state := {caa} ⩤ caaction_state ||
                caaction_particip := epar ⩤ caaction_particip ||
                particip_caaction := {caa} ⩤ particip_caaction ||
                caaction_ext_objects := {caa} ⩤ caaction_ext_objects ||
                is_composed := is_composed - {pa ↦ caa}
          END
   END;
/**** ABORT OPERATION **********************/
   abort_caaction(caa) =
   PRE
          caa ∈ caaction ∧
          caa ∉ dom(is_nested) ∧
          caa ∉ ran(is_nested) ∧
          caa ∉ ran(is_composed)
   THEN
          LET epar BE
                epar =particip_caaction(caa)
          IN
                  /* all external objects are aborted */
                terminate_transaction(caaction_ext_objects[{caa}], abort_val) ||
                delete_participants(epar) ||
                caaction := caaction - {caa} ||
                caaction_state := {caa} ⩤ caaction_state ||
                caaction_particip := epar ⩤ caaction_particip ||
                particip_caaction := {caa} ⩤ particip_caaction ||
                caaction_ext_objects := {caa} ⩤ caaction_ext_objects
          END
   END;
   abort_nested_caaction(caa) =
   PRE
          caa∈ caaction ∧
          caa∈ dom(is_nested) ∧
          caa ∉ ran(is_nested) ∧
          ∀pa.(pa ∈ particip_caaction(caa) ⇒ last(participant_state(pa)) ≠ waiting )
   THEN
          LET epar BE
                epar = particip_caaction(caa)
          IN
                  /* all external objects are aborted */
                terminate_nested_transaction(caaction_ext_objects[{caa}],abort_val) ||
                remove_participant_state_and_value(epar) ||
                caaction := caaction - {caa} ||
```

```
                caaction_state := {caa} ⩤ caaction_state ||
                caaction_particip := caaction_particip
                        ⩤ {pa,scaa| pa ∈ epar ∧ scaa∈ seq(caaction) ∧
                                scaa = front(caaction_particip(pa))} ||
                particip_caaction := {caa} ⩤ particip_caaction ||
                caaction_ext_objects := {caa} ⩤ caaction_ext_objects
        END
    END;
    abort_composed_caaction(caa,pa) =
    PRE
        caa∈ caaction ∧
        pa∈ participant ∧
        pa∈ dom(is_composed) ∧
        caa = is_composed(pa) ∧
        caa ∉ dom(is_nested) ∧
        caa ∉ ran(is_nested)
    THEN
        LET epar BE
            epar = particip_caaction(caa)
        IN
            /* all external objects are aborted */
            terminate_transaction(caaction_ext_objects[{caa}],abort_val) ||
            /* remove last state and set to exceptional */
            remove_composed_participants_exceptional(epar, pa) ||
            caaction := caaction - {caa} ||
            caaction_state := {caa} ⩤ caaction_state ||
            caaction_particip := epar ⩤ caaction_particip ||
            particip_caaction := {caa} ⩤ particip_caaction ||
            caaction_ext_objects := {caa} ⩤ caaction_ext_objects ||
            is_composed := is_composed - {pa ↦ caa}
        END
    END;
/**** EXCEPTIONAL TERMINATION **********************/
    terminate_caaction_exceptional(caa) =
    PRE
        caa ∈ caaction ∧
        caa ∉ dom(is_nested) ∧
        caa ∉ ran(is_nested) ∧
        caa ∉ ran(is_composed) ∧
        caaction_state(caa) = caa_exceptional ∧
        ∀pa.(pa ∈ particip_caaction(caa) ⇒ last(participant_state(pa)) ∈ EXCEPTIONAL_STATE )
    THEN
        LET epar BE
            epar =particip_caaction(caa)
        IN
            /* all external objects are aborted */
            terminate_transaction(caaction_ext_objects[{caa}], abort_val) ||
            delete_participants (epar) ||
            caaction := caaction - {caa} ||
            caaction_state := {caa} ⩤ caaction_state ||
            caaction_particip := epar ⩤ caaction_particip ||
            particip_caaction := {caa} ⩤ particip_caaction ||
            caaction_ext_objects := {caa} ⩤ caaction_ext_objects
```

```
        END
  END;
  terminate_nested_caaction_exceptional(caa) =
  PRE
        caa∈ caaction ∧
        caa∈ dom(is_nested) ∧
        caa∉ ran(is_nested) ∧
        caaction_state(caa) = caa_exceptional ∧
        ∀pa.(pa ∈ particip_caaction(caa) ⇒ last(participant_state(pa)) ∈ EXCEPTIONAL_STATE )
  THEN
        LET epar BE
              epar =particip_caaction(caa)
        IN
                /* all external objects are aborted */
              terminate_nested_transaction(caaction_ext_objects[{caa}], abort_val) ||
                /* the participants state is set to exceptional
                   (the exception is signalled outside the CA action) */
              remove_update_participants_states(epar, exceptional)||
              caaction := caaction - {caa} ||
              caaction_state := {caa} ⩤ caaction_state ||
              caaction_particip := caaction_particip
                        ⩤ { pa, scaa | pa ∈ epar ∧ scaa∈ seq(caaction) ∧
                                  scaa= front(caaction_particip(pa)) } ||
              particip_caaction := {caa} ⩤ particip_caaction ||
              caaction_ext_objects := {caa} ⩤ caaction_ext_objects
        END
  END;
  terminate_composed_caaction_exceptional(caa, pa) =
  PRE
        caa∈ caaction ∧
        pa∈ participant ∧
        pa∈ dom(is_composed) ∧
        caa = is_composed(pa) ∧
        caa ∉ dom(is_nested) ∧
        caa ∉ ran(is_nested) ∧
        caaction_state(caa) = caa_exceptional ∧
        ∀pacaa.(pacaa ∈ particip_caaction(caa)
              ⇒ last(participant_state(pacaa)) ∈ EXCEPTIONAL_STATE )
  THEN
        LET epar BE
              epar = particip_caaction(caa)
        IN
                /* all external objects are aborted */
              terminate_transaction(caaction_ext_objects[{caa}],abort_val) ||
              caaction := caaction - {caa} ||
              caaction_state := {caa} ⩤ caaction_state ||
                /* remove last state and set to exceptional */
              remove_composed_participants_exceptional(epar,pa) ||
              caaction_particip := epar ⩤ caaction_particip ||
              particip_caaction := {caa} ⩤ particip_caaction ||
              caaction_ext_objects := {caa} ⩤ caaction_ext_objects ||
              is_composed := is_composed - {pa ↦ caa}
        END
```

```
  END
END
```

A.3. Proofs

The tool has generated 837 proof obligations (416 are obvious and 421 non-obvious). Approximately 58% of the 421 non-obvious obligations are proved automatically. The remaining obligations should be proved interactively. The main reason that all the obligations have not been proved automatically is the fact that we have not written operations and invariants according to the capabilities of the tool. Especially the tool provides very poor results in case of using sequences that we have used a lot in order to have a less complex syntax. On the other hand, due to time limitations we have not started interactive proofs which require writing special lemmas, moreover that on each change to the abstract machine operations, variables or invariants, most of them should be rewritten and/or reproved.

A.4. Discussion

The model is built under an assumption that there is an underlying nested support that guarantees the transactional behaviour of the external objects. For each nested CA action calling an external object the support starts a nested transaction on the external object and commits or aborts it on action termination. If the containing CA action aborts after the nested one commits, the support ensures that all nested operations are aborted.

A participant that makes a call to create a composite action enters a waiting state: it neither accepts any message from other participants nor it can be part of a nested action until the composed action is terminated. It restores its state to "normal" or "exceptional" according to the composite CA action outcome. If the composite CA action returns an exception or is aborted or the operation timeouts, the participant raises an internal exception which is either handled locally or when it is not possible, is propagated to all participants and the participant enters the "exceptional" state. External objects accessed by the composite CA action, or recursively, by other composite CA actions invoked by it, are unknown to the participant of the main containing CA action, as well the possibility that other CA actions may have accessed to them.

We have addressed the formal specification of Coordinated Atomic Actions enriched with the notion of composite actions. We are now working on the formalization of Web Service Composition Actions introduced in Chapter 2 based on this experience.

References

[Abrial 1996] J. R. Abrial. The B Book - Assigning Programs to Meanings. Cambridge University Press. 1996.

[Atelier B] Atelier B. Clearsy (France) http://www.atelierb.societe.com/index_uk.html

[B-Core] The B-Tool. B-Core (UK) http://www.b-core.com/btool.html

[Xu *et al.* 1995] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud and Z. Wu. “Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error

Recovery" in *Proc. the 25th International Symposium on Fault Tolerant Computing* (FTCS-25), Pasadena, California, pp.499 – 509, 1995.