Formal Development of Reactive Fault Tolerant Systems

Linas Laibinis and Elena Troubitsyna

Åbo Akademi, Department of Computer Science,
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
(Linas.Laibinis, Elena.Troubitsyna)@abo.fi

Abstract. Usually complex systems are controlled by an operator co-operating with a computer-based controller. The controlling software runs in continuous interaction with the operator and constantly reacts on operator's interruptions by dynamically adapting system behaviour. Simultaneously it catches the exceptions signalling about errors in the system components and performs error recovery. Since interruptions are asynchronous signals they might concurrently co-exist and conflict with exceptions. To ensure dependability of a dynamically adaptable system, we propose a formal approach for resolving conflicts and designing robust interruption and exception handlers. We present a formal specification pattern for designing components of layered control systems that contain interruption and exception handlers as an intrinsic part of the specification. We demonstrate how to develop a layered control system by recursive application of this pattern.

1 Introduction

In this paper we propose a formal approach to the development of dependable control systems. Control systems are the typical examples of reactive systems. Often they run in a constant interaction not only with the controlled physical application but also the operator. The operator participates in providing the control over an application by placing requests to execute certain services and often intervening in the service provision. As a response to the operator's intervention, the controller should adapt the behaviour of the system accordingly. The task of ensuring dependability of dynamically adaptable systems is two-fold: on the one hand, we should design the controller to be flexible enough to allow the operator's intervention; on the other hand, the controller should prevent potentially dangerous interventions in its service provision.

Design of dependable control systems usually spans over several engineering domains. Traditionally abstraction, modularisation and layered architecture are recognized to be effective ways to manage system complexity [14]. Though the components at each architectural layer are susceptible to specific kinds of faults, the mechanism of exception raising and handling can be used for error detection and recovery at each architectural layer.

However, since exceptions and interruptions are asynchronous signals, several exceptions and interruptions might occur simultaneously. Incorrect resolution of such conflicting situations might seriously jeopardize system dependability. In this paper we formally analyse the relationships between interruptions and exceptions and propose formal guidelines for designing robust interruption and exception handling. Moreover, we propose a formal approach to the development of reactive fault tolerant control systems in a layered manner.

Our approach is based on stepwise refinement of a formal system model in the B Method [1,2,15]. While developing a system by refinement, we start from an abstract specification and step by step incorporate implementation details into it until executable code is obtained. In this paper we propose a general pattern for specification and refinement of reactive layered systems in B. Our pattern contains exception and interruption handlers as an intrinsic part of the specification. We start from the specification of the system behaviour on the upper architectural layer and unfold layers by a recursive instantiation of the proposed specification pattern. Since our approach addresses the dependability aspects already in the early stages of system development, we argue that it has potential to enhance system dependability.

We proceed as follows: in Section 2 we discuss propagation of exceptions and interruptions in the layered control systems. In Section 3 we present a formal basis for designing interruption and exception handlers. In Section 4 we demonstrate our approach to formal development of fault tolerant reactive systems that contain interruption and exception handlers as an intrinsic part of their specifications. In Section 5 we summarize the proposed approach, discuss its possible extensions and overview the related work.

2 Exceptions and Interruptions in a Layered Architecture

Control systems. In this paper we focus on modelling dependable control systems. A general structure of a control system is given in Fig.1. A plant is a physical entity whose operation is being monitored and controlled by a computer-based controller. Often control over an application is provided in co-operation of computer with an operator – a human (or sometimes another computer-based system) participating in operating the system. The controller monitors plant's behaviour via the sensors and affects it via the actuators as shown in Fig. 1.

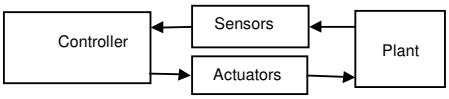

Fig.1 General structure of a control system

Layered architecture. Usually development of a control system spans over several engineering domains, such as mechanical engineering, software engineering, human-computer interface etc. It is widely recognized that a layered architecture is preferable in designing such complex systems since it allows the developers to map real-world domains into software layers [14]. The lowest layer confines real-time subsystems which directly communicate with sensors and actuators – the electro-mechanical devices used to monitor and control the plant. These subsystems cyclically execute the standard control loop consisting of reading the sensors, and assigning the new states to the actuators. The layer above contains the components that encapsulate the detailed behaviour of the lowest level subsystems by providing abstract interfaces to them. The component server (often called the service director) is on the highest level of hierarchy. It serves as an interface between the operator and the components.

There are several ways in which the operator can interact with the system. Normally s/he places the requests to execute certain services. A service is an encapsulation of a set of operations to be executed by the components. Upon receiving a request to execute a service, the component server at first translates (decomposes) the service into the corresponding sequence of operations. Then it initiates and monitors the execution of

operations by placing corresponding requests on the components. In their turns, the requested components further decompose these operations into the lower level operations. These operations are to be executed by real-time subsystems at the lowest layer of hierarchy. Upon completion of each operation, the requested subsystem notifies the requesting component about success of the execution. The component continues to place the requests on the subsystems until completion of the requested operation. Then it ceases its autonomous functioning and notifies the component server about success of the execution. The behaviour of the components follows the same general pattern: the component is initially "dormant" but becomes active upon receiving a request to execute a certain operation. In the active mode the component autonomously executes a operation until completion. Then it returns the acknowledgement to the requesting component and becomes inactive again. The communication between components can be graphically represented as shown in Fig.2.

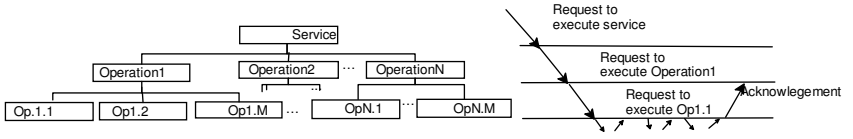

Fig.2. Architecture of a layered system

Exceptions. While describing the communication between the layers of a control system, we assumed so far that the system is fault-free, i.e., after receiving a request to execute an operation, the component eventually successfully completes it. However, occurrence of errors might prevent a component from providing a required operation correctly. Hence, while designing a controller, we should specify means for tolerating fault of various natures. In this paper we focus on hardware faults and human errors.

The main goal of introducing fault tolerance is to design a system in such a way that that faults of components do not result in system failure [3,4,11]. A fault of a component manifests itself as an error [3]. Upon detection of an error, error recovery should be performed. Error recovery is an attempt to restore a fault-free system state or at least preclude system failure. Hence components of fault tolerant controllers should be able to detect the errors and notify the requesting component, so that error recovery can be initiated. This behaviour is achieved via the mechanism of exception raising and handling [5]. Observe that for each component (except the lowest level subsystems) we can identify two classes of exceptions:

1. *generated exceptions*: the exceptions raised by the component itself upon detection of an error,
2. *propagated exceptions*: the exceptions raised at the lower layer but propagated to the component for handling.

The generated exceptions are propagated upwards (to the requesting component) for handling. Usually the component that has raised an exception ceases its autonomous functioning. Such a behaviour models the fact that the component is unable to handle the erroneous situation. With each component we associate a class of errors from which the component attempts to recover by itself. If the component receives a propagated exception signalling about error from this class, then it initiates error recovery by requesting certain lower layer operations. Otherwise the component propagates the exception further up in the hierarchy. Hence certain errors will be propagated to the

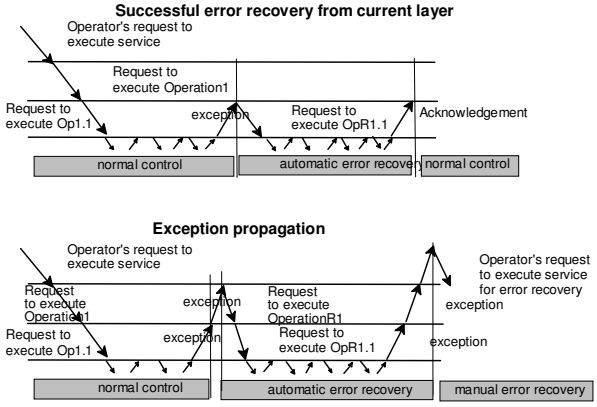

Fig.3. Exceptions in a layered architecture

operator, so s/he could initiate manual error recovery. This behaviour is graphically represented in Fig.3.

Interruptions. While discussing service provision in a layered architecture we assumed that the system accepts the operator's request to execute a certain service after it has completed execution of a previous service, i.e., in an idle state. However, often the operator needs to intervene in the service execution. For instance, s/he might change "on-the-fly" the parameters of the currently executing service or cancel it, suspend and resume service provision etc. Such interventions are usually called interruptions. Observe that a request to execute a service can also be seen as the special case of interruption.

Interruptions are dual to exceptions. They arrive from the uppermost architectural layers and are "propagated" downwards, to the currently active layer. Upon receiving an interruption, the currently active component takes appropriate actions required to handle it. The component can either

- change the local state to adjust the execution and then resume its work, or
- generate the requests to execute certain lower-layer subservices (which might be seen as a special case of error recovery), or
- "realize" that the interruption should be handled on a higher layer of hierarchy. Then it would generate the corresponding exception, which is then propagated upwards.

The behaviour of the system while handling interruption is graphically represented in Fig.4.

While designing dependable systems, we need to analyse the impact of interruptions and exceptions on system dependability. In the next section we will address this issue in details.

3 Formal Analysis of Interruption and Exception Handling

Ensuring dependability of the systems, which can adapt their behaviour in response to operator's interruptions, is a complex task. It involves establishing a proper balance between system flexibility and dependability. For instance, if the behaviour of the autopilot deems to be faulty, the controlling software of an aircraft should allow the pilot to interrupt the autopilot and resume manual control; on the other hand, software should

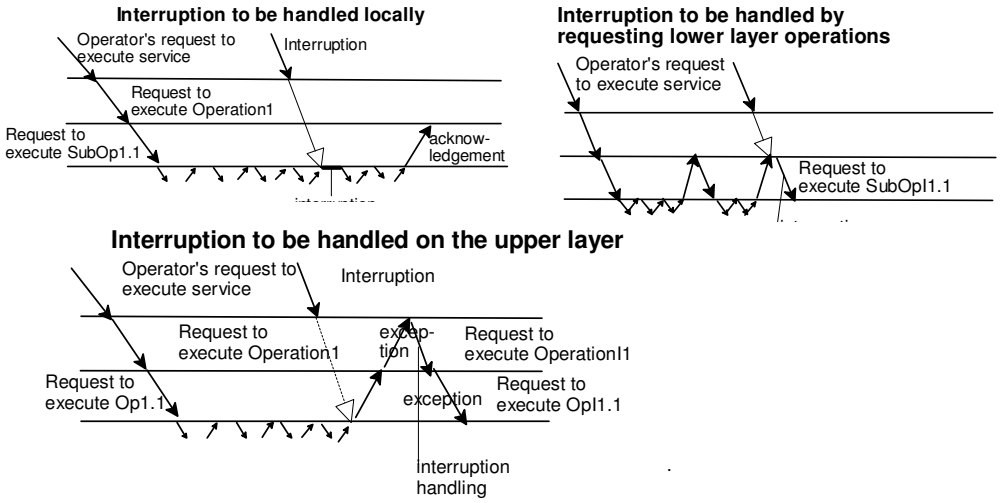

Fig.4. Interruptions in a layered architecture

disallow the pilot to interrupt automatic control in a potentially dangerous way. Next we consider a formal basis for designing dependable reactive systems.

Preventing incorrect interventions. At first we observe that the interface of the system is the medium via which the operator requests and interrupts services. Hence design of the system interface should obey the principles of designing human-computer interfaces for error prevention (the study of those is outside of the scope of this paper). Usually the resulting system interface is dynamic, i.e., the set of the requests and interruptions which the system accepts from the operator varies depending on the internal system state.

Assume that $I = \{I_1, I_2, .. I_M\}$ is a complete set of interruptions and requests which the operator can send to the system via its interface. We define the function

$$blocking: I \rightarrow \mathbf{P}I,$$

which for each interruption I_j, $j:1..M$, returns a subset of interruptions or requests which can be next accepted by the system. For instance, if the operator has sent the interruption *"Pause"* then only the interruptions *"Continue"* and *"Abort"* can be accepted next. Let us observe that some interruptions or requests are non-blocking ($blocking(I_k)=I$, for some k), i.e., they do not disable other interruptions. The function *blocking* explicitly defines how to dynamically adapt the system interface to enforce the correct operator's behavior.

In this paper for the sake of simplicity we defined the function *blocking* in a static, i.e., independent of the system state way. The function *blocking_dyn*

$$blocking\_dyn: (I, \sigma) \rightarrow \mathbf{P}I,$$

where σ is the system state (e.g., the set of system variables), modifies *blocking* by taking into account the current system state while defining the subset of requests and interruptions that can be accepted next.

Next we study how the controller should prevent the incorrect intervention into its service provision. We define the relationships between interruptions and the controller's operations. Namely, for each operation of the controller, we identify the subset of interruptions which must be immediately handled and the subset of interruptions

handling of which should be postponed until execution of the current operation is completed.

Assume that C is a component in our layered architecture. Assume also that $OpC = \{OpC_1, OpC_2, \ldots OpC_N\}$ is a set of operations provided by C and I is a set of interruptions and requests accepted by the overall system. For each operation in OpC, we analyse the consequences of interrupting its execution by each of the interruption from I. As a result of such analysis, we define a function

$$atomicC : OpC \times I \rightarrow Bool$$

such that, for any $i: 1..N$ and $j: 1..M$, $atomicC(OpC_i, I_j)$ is $FALSE$ if interruption I_j received during the execution of OpC_i should be handled immediately, and $TRUE$ if handling of I_j should be postponed until completion of OpC_i. In the latter case the operation $OPCi$ is atomic, i.e., uninterruptible by the interruption I.

Since interruptions are asynchronous events and sometimes interruption handling is postponed, the interruptions can be queued for handling. Let us consider the following situation: after unsuccessful attempts to recover from error by sending certain interruption, the operator decides to cancel the execution of the current service. If the interruptions are handled in the "first-in-first-out" order, then the service will be cancelled only after all previous interruptions are handled, i.e., with a delay. This might be undesirable or even dangerous. Hence we need to distinguish between the levels of criticality of different interruptions, for instance, to ensure that, if an interruption is an attempt to preclude a dangerous system failure, it is handled with the highest priority.

We define the function

$$I\_EVAL: I \rightarrow NAT$$

which assigns priorities to interruptions and requests. The greater the value of $I\_EVAL(I_j)$, where $j:1..M$, the higher degree of priority of handling the interruption I_j. While designing the system, we ensure that the interruptions are handled according to the priority assigned by I_EVAL.

Interruptions versus exceptions. Above we analysed the principles of interruption handling. However, our analysis would be incomplete if we omit consideration of the relationship between interruptions and exceptions. Indeed, since interruptions are asynchronous events, they might co-exist and "collide" with exceptions, e.g., when an interruption is caught simultaneously with the exception indicating an erroneous situation. Dealing with concurrent arrival of several signals from different sources has been recognized as a serious problem that has not received sufficient attention [4]. Incorrect handling of these signals might lead to the unexpected system behaviour and, as a consequence, can seriously jeopardize system dependability. To resolve such potentially dangerous situations, handling of simultaneous signals should be designed in a structured and rigorous way.

Let C be a component on a certain layer of our layered architecture. Let $EXC\_C = \{Exc_1, Exc_2, \ldots Exc_N\}$ be a set of exceptions that can be propagated to C from the lower layer components. We define the function

$$E\_EVALC : EXC\_C \rightarrow NAT$$

which assigns a certain criticality level to each exception which component receives. By defining E_EVALC for each component of our system we assign a certain priority to each exception to be handled by the system.

Let us consider now an active component C, which has currently caught the exception Exc_1 and interruptions I_1, I_2, I_3 such that $I\_EVAL(I_1) > I\_EVAL(I_2) > I\_EVAL(I_3)$. Then

- if the interruption I_1 is more critical than the exception, i.e., $I\_EVAL(I_1) \geq E\_EVALC(Exc_1)$ then the next signal to be handled is the interruption I_1,
- if the exception is more critical than the interruptions, i.e., $E\_EVALC(Exc_1) > I\_EVAL(I_1)$ then the next signal to be handled is the exception Exc_1.

Upon completion of handling the most critical signal, the caught signals are evaluated in the same way. Then the decision which signal should be handled next is made again.

Observe that the functions *E_EVALC* and *I_EVAL* can be extended in a similar way as the function *blocking*. This would allow the systems to take into account its current state while making the decision about criticality of exceptions and interruptions.

Interruption and exception propagation. Let us now discuss the design of interruption and exception handlers. We identify three classes of interruptions:

- the interruptions, whose handling can be done locally, i.e., by changing local variables of the currently active component,
- those, whose handling requires to invoke the lower layer operations, and
- those, whose handling is possible only on some higher layer (the received interruption is converted into an exception to be propagated upward).

The identified classes are disjoint. The proposed classification is complete in a sense that it defines all possible types of system responses on interruptions.

We define the function (for each component *C*)

$$I\_STATUS\_C: I \rightarrow \{Local, Down, Up\}$$

which, for any interruption, defines the type of the required handling.

In the similar way we define the types of exceptions as exceptions signalling about

- successful termination of requested service,
- recoverable error, or
- unrecoverable error.

The corresponding function (for each component *C*)

$$E\_STATUS\_C: EXC\_C \rightarrow \{Ok, Recov, Unrecov\}$$

defines the type of each exception and acknowledgement. If the acknowledgement notifies about successful termination then the normal control flow continues. If the exception signals about recoverable error then error recovery from the current layer is attempted. Otherwise, the exception is propagated upward in the hierarchy.

In the latter case, we need to define the rules for converting unrecoverable propagated exceptions and interruptions into generated exceptions of the current component. After conversion, the corresponding exception of the current component is raised and propagated up. For every pair of components (C_i, C_j) such that C_i is a requesting component (client) and C_j is a requested component from the lower layer, we define the function

$$E\_\ CONV_{ij}: EXC\_C_i \rightarrow EXCg\_\ C_j$$

where EXC_C_i is the set of propagated and $EXCg\_C_j$ is the set of generated exceptions of the corresponding components. The function $E\_\ CONV_{ij}$ converts propagated exceptions of C_i into generated exceptions of C_j.

In a similar way, we design interruption handling converting the received interruption into an exception to be propagated upwards. For every component *C,* we define the function

$$I\_CONV: I \rightarrow EXC\_Cg$$

Let us observe that our exception handling has hierarchical structure: while designing exception handling we follow the principle "the more critical an error is, the higher the layer that should handle its exception". This principle should be utilized while defining the functions *E_EVALC* (for each component *C*) and *I_EVAL*.

In this section we formalized the principles of designing interruption and exception handling in a layered architecture. In the next section we present our approach to specification and refinement of dependable reactive systems.

4. Specification and Refinement of Reactive Fault Tolerant Systems

It is widely accepted that high degree of dependability of the system can only be achieved if dependability consideration starts from the early stages of system development [11,17]. We demonstrate how to specify layered control systems in such a way that mechanisms for interruption and exception handling become an intrinsic part of their specification. We start by a brief introduction into the B Method – our formal development framework.

The B Method. The B Method [1,15] (further referred to as B) is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [13]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [16], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

The development methodology adopted by B is based on stepwise refinement [1]. While developing a system by refinement, we start from an abstract formal specification and transform it into an implementable program by a number of correctness preserving steps, called *refinements*. A formal specification is a mathematical model of the required behaviour of a (part of) system. In B a specification is represented by a set of modules, called Abstract Machines. An abstract machine encapsulates state and operations of the specification and as a concept is similar to module or package.

Each machine is uniquely identified by its name. The state variables of the machine are declared in the VARIABLES clause and initialized in the INITIALISATION clause. The variables in B are strongly typed by constraining predicates of the INVARIANT clause. All types in B are represented by non-empty sets. We can also define local types as *deferred sets*. In this case we just introduce a new name for a type, postponing actual definition until some later development stage.

The operations of the machine are defined in the OPERATIONS clause. There are two standard ways to describe an operation in B: either by the preconditioned operation PRE cond THEN body END or the guarded operation SELECT cond THEN body END. Here cond is a state predicate, and body is a B statement. If cond is satisfied, the

behaviour of both the precondition operation and the guarded operation corresponds to the execution of their bodies. However, if cond is false, then the precondition operation leads to a crash (i.e., unpredictable or even non-terminating behaviour) of the system, while the behaviour of the guarded operation is immaterial since it will be not executed. The preconditioned operations are used to describe operations that will be implemented as procedures modelling requests. The guarded operations are used to specify event-based systems and will model autonomous behaviour.

B statements that we are using to describe a state change in operations have the following syntax:

```
S  ==  x := e | IF cond THEN S1 ELSE S2 END | S1 ; S2 |
       x :: T  | S1 || S2 | ANY z WHERE cond THEN S END  |  ...
```

The first three constructs – assignment, conditional statement and sequential composition (used only in refinements) have the standard meaning. The remaining constructs allow us to model nondeterministic or parallel behaviour in a specification. Usually they are not implementable so they have to be refined (replaced) with executable constructs at some point of program development. The detailed description of the B statements can be found elsewhere [1,15].

The B method provides us with mechanisms for structuring the system architecture by modularization. The modules (machines) can be composed by means of several mechanisms providing different forms of encapsulation. For instance, if the machine C INCLUDES the machine D then all variables and operations of D are visible in C. However, to guarantee internal consistency (and hence independent verification and reuse) of D, the machine C can change the variables of D only via the operations of D. In addition, the invariant properties of D are included into the invariant of C. To make the operations of D available through the interface of C, we should list then in the PROMOTE clause of C. If D promotes all its operations to C then C is an extension of D which can be specified by the EXTENDS mechanism.

Refinement and layered architecture. Refinement is a technique to incorporate implementation details into a specification. In general, the refinement process can be seen as a way to reduce nondeterminism of the abstract specification, to replace the abstract mathematical data structures by the data structures implementable on a computer and to introduce underspecified design decisions. In the Abstract Machine Notation (AMN), the results of the intermediate development stages – the refinement machines – have essentially the same structure as the more abstract specifications. In addition, they explicitly state which specifications they refine.

Each refinement step should be formally verified by discharging (proving) certain proof obligations. Since verification of refinement is done by proofs rather than state exploration, the stepwise refinement technique is free of the state explosion problem and hence is well suited for the development of complex systems. In this paper we demonstrate how refinement facilitates development of systems structured in a layered manner.

Let us observe that the schematic representation of communication between the components of a layered system represented in Fig.2 can also be seen as the scheme of atomicity refinement. Indeed, each layer decomposes a higher layer operation into a set of operations of smaller granularity. The decomposition continues iteratively until the lowest layer is reached. At this layer the operations are considered to be not further decomposable. From the architectural perspective, an abstract specification is a “folded” representation of the system structure. The system behaviour is specified in terms of

large atomic services at the component server layer. Each refinement step adds (or "unfolds") an architectural layer in the downward direction. Large atomic services are decomposed into operations of smaller granularity. Refinement process continues until the whole architectural hierarchy is built. We argue that the refinement process conducted in such a way allows us to obtain a realistic model of fault tolerant reactive systems. Indeed, by iterative refinement of atomicity we eventually arrive at modelling exceptions and interruptions arriving practically at any instance of time, i.e., before and after execution of each operation of the finest granularity. The proposed refinement process is illustrated in Fig.5, where we outline the development pattern instantiated to a three-layered system.

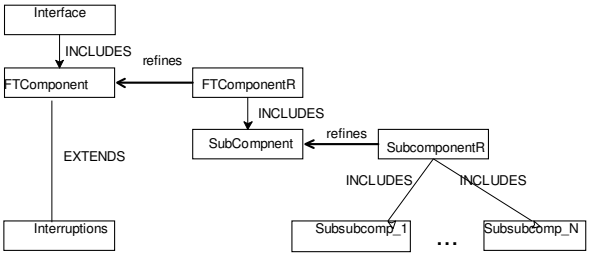

Fig.5. Refinement process

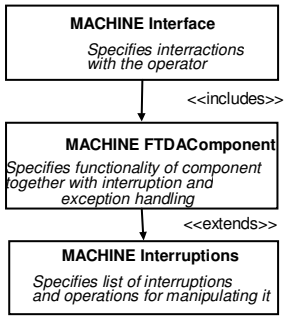

Fig.6. Abstract specification

Abstract specification in B. The initial abstract specification consists of three machines (see Fig.6). The machine Interface models the interface of the system. The machine Interruptions describes the data structure representing the interruptions. The machine Component contains the abstract specification of component server, i.e., it models services which the system provides for the operator as well as interruption and exception handling on the component server layer. The interactions of the operator with the interface result in invocations of the operations of FTDAComponent. Hence Interface INCLUDES FTDAComponent.

Each refinement step leads to creating components at the lower layer by including their specifications into the refinement of the corresponding components at the previous layer. Observe, that in the abstract specification the operator activates a component server by placing a request to execute a service. At the next refinement step we refine the specification of the component server to model placing requests on the lower layer component and introduce the specification of the lower layer component together with interruption and exception handling performed by it. Since the behaviour of the components on each layer follows the same pattern, the refinement process is essentially the recursive instantiation of the abstract specification pattern. Next we present the specification on the machines defining specification patterns in details.

The specification of Interruptions contains the data structure modelling interruptions and the operations for manipulating them. The operation add_interruption inserts arriving interruptions in the sequence Inter modelling a list of interruptions to be handled. The list Inter is sorted according to the criticality assigned to interruptions by the function *I_EVAL* and the operation add_interruptions preserves this order. The handled interruptions are removed from the list via the operation remove_interruption. In addition, remove_interruption removes interruptions that have become redundant or irrelevant after handling the last interruption.

```
MACHINE  Interruptions
SETS INTERRUPTIONS
VARIABLES Inter
INVARIANT  Inter : seq(INTERRUPTIONS) &  interruptions are sorted by their criticality
INITIALISATION Inter := [ ]
OPERATIONS
  add_interruption (inter)=  PRE inter is valid THEN
     add inter to Inter and ensure that interruptions remain sorted by criticality  END;

  remove_interruption (inter) =  PRE Inter is not empty THEN
    remove inter and irrelevant interruptions from Inter  END
END
```

The machine Interface given below

```
MACHINE  Interface
INCLUDES ServiceComponent
VARIABLES current
INVARIANT  current: INTERRUPTIONS
INITIALISATION current:= No_int

OPERATIONS
  request (request_parameters) = PRE flag=stopped  THEN activate Component  END;

   interruption_1 =  PRE interruption_1 is allowed THEN
     add_interruption(interruption_name1)  || current:= interruption_name1  END;

   interruption_N =  PRE interruption_nameN is allowed THEN
     add_interruption(interruption_nameN) ||  current:= interruption_nameN  END;

   abort = BEGIN add_interruption(Abort) || current := Abort END
END
```

defines how the system interacts with its environment, i.e., it models the service requests and interruptions. The operation request models placing a request to execute a certain service. The request can be placed only when the system is in the idle state, i.e., the previous request has already been completed or cancelled. The interactions are modelled by the operations with corresponding names. Unlike requests, interruptions can arrive at any time. However, an acceptance of the new interruption depends on the previously arrived interruption as defined in Section 3 by the function *blocking*. The preconditions of interruption_1…interruption_N check whether the previously arrived interruption, current, blocks the newly arrived interruption. If the interruption is accepted then the arriving interruption is added to the list of interruptions to be handled and the value of current is updated.

The behaviour of fault tolerant dynamically adaptable component is abstractly specified by the machine FTDAComponent presented in Fig.7.

The generated and propagated exceptions are modelled by the variables exc and exc2 respectively. We abstract away from the implementation details of exceptions by choosing deferred sets EXC and EXC2 as the types for exc and exc2. The currently handled interruption is stored in the variable last_int. The local state of the component is modelled by the variable state.

Each phase of execution is specified by the corresponding operation. The value of the variable flag indicates the current phase.

```
MACHINE  FTDAComponent
EXTENDS  Interruptions
VARIABLES  flag, atomic_flag, exc, exc2, state, last_int
INVARIANT
  flag : {Executing,Handling,Recovering, Pausing,Stopping,Stopped} &
  exc : EXC  &  exc2 : EXC2  &  state : STATE  &  atomic_flag : BOOL  &  last_int : Interruptions &
  properties of exceptions, interruptions and phases
...
OPERATIONS
start(a_flag,cmd) =
  PRE component is idle
  THEN
    IF the requested command cmd is valid
    THEN  Initiate execution
        Check a_flag and function atomic and decide whether execution of cmd can be interrupted
         by assigning the corresponding value to atomic_flag
        Start execution (flag := Executing)
    ELSE Raise exception And stop component (flag := Stopping) END
  END;

execute =
  SELECT flag = Executing THEN
    IF component generated exception because execution of cmd is unsafe or it completed cmd
      THEN stop component (flag := Stopping)
      ELSE Invoke lower layer operations and catch exc2 and start handling (flag := Handling)
      END END;

i_handle =
     SELECT flag = Handling &
       Current service is not atomic and criticality of interruption is greater than criticality of
       exception (I_EVAL(first(Inter)) >  E_EVAL2(exc2)) or arrived interruption is Abort
      THEN
      IF current interruption is Abort
      THEN stop component and propagate exception
      ELSE determine the class of interruption (using I_STATUS) and handle accordingly
       END
       Remove handled interruption from the list and update last_int   END;

e_handle =
  SELECT flag = Handling &
       Current service is atomic and  the arrived interruption is not Abort, or
       no interruptions has arrived, or criticality of exception is greater than interruption
       (I_EVAL(first(Inter)) < E_EVAL2(exc2))
    THEN Classify exceptions according to E_STATUS and handle accordingly END;

pause =
  SELECT flag = Pausing AND Newly arrived interruption is Continue or Abort
    THEN Remove handled interruption And Check system state after pausing
       Start handling (i.e., flag := Handling) END;

recover =
  SELECT flag = Recovering THEN Perform error recovery and catch lower layer exception END;

stop =
  SELECT flag = Stopping THEN Cease component's functioning  (flag:= Stopped) END
END
```

Fig.7 Specification of FTDAComponent

The operation start models placing a request to execute a service on the service component. It sets the initial state according to the input parameters of the request. The

execution of the service is modelled by the operation execute. In the initial specification we model the effect of executing the lower layer command by receiving a propagated exception and updating the local state.

We assume that the interruption Abort should be handled regardless of the atomicity of the service though in certain systems it might be treated in the same way as other interruptions. Hence the operation i_handle becomes enabled either if the interruption Abort has arrived or the current service is non-atomic and the criticality of the caught interruption is higher than the criticality of the caught exception. The interruption handling proceeds according to the type of interruption to be handled – we discuss it in the details later.

The operation e_handle becomes enabled after the lower layer has completed its execution and the current interruption has a lower priority than the propagated exception or the current service is marked as atomic. The task of e_handle is to classify the propagated exception exc2 and perform handling accordingly.

The operation recover is similar to the operation execute in the sense that the lower layer is called: the state of the component is changed and a new propagated exception is received. The purpose of operation recover is to abstractly model the effect of error recovery. After recover, either e_handle or i_handle becomes enabled, which again evaluates the propagated exception and the current interruption and directs the control flow accordingly.

Finally, the operation stop becomes enabled in two cases: when an unrecoverable error has occurred, or the execution of the request is completed.

The behaviour of the component can be graphically represented as shown in Fig.8.

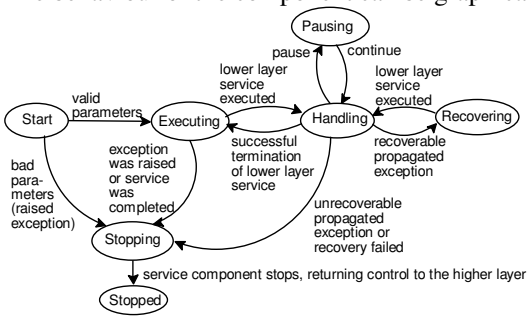

Fig.8. Behaviour of fault tolerant reactive component

The proposed specification can be used to abstractly specify components at each layer except the lowest one. Each component at the lowest layer has only one type of exceptions – the generated exceptions. Hence the specification facilities for exception handling are redundant at this layer. Moreover, the operations to be executed at the lowest layer are not decomposed any further and hence can be specified as the atomic preconditioned operations.

Let us observe that while describing the behaviour of the component we single out handling of the interruption "Pause" because its handling is merely a suspension of autonomous component functioning.

Interruption and exception handlers. Now we discuss in detail the specifications of interruption and exception handlers. The interruption handler – the operation i_handle – described below analyses the interruption to be handled – first(Inter) – and chooses the corresponding interruption handling.

```
i_handle =
 SELECT
   flag = Handling & (atomic_flag=FALSE & size(Inter)>0 &
   I_EVAL(first(Inter)) >= E_EVAL2(exc2,state)) or  first(Inter) = Abort
 THEN
   IF first(Inter)=Abort
   THEN raise(Abort_exc) || flag := Stopping
   ELSEIF first(Inter) = Pause THEN flag := Pausing
   ELSIF I_STATUS(first(Inter))=Local THEN state :: STATE
   ELSIF I_STATUS(first(Inter))=Up THEN raise(I_CONV((first(Inter)))) || flag := Stopping
   ELSE  flag := Recovering
   END ||
   last_int := first(Inter) ||  remove_interruption
 END;
```

In case the interruption is Abort the component is stopped and exception stopping the requesting component is propagated upward in the hierarchy until all activated components are stopped. If the interruption to be handled is Pause then the component suspends its functioning and enters the Pausing phase. Handling of the interruption requiring the local state update is modelled by the update of the local state. The component remains in the phase Handling and continues to analyse the caught signals. If the interruption requires handling to be performed on the higher layers of hierarchy then the component is stopped and the corresponding exception (obtained by using the function *I_CONV*) is raised. Otherwise the interruption handling is performed by requesting the operations from the lower layers, which is modelled in the operation recover.

The specification of exception handler e_handle is done in the similar style – the function *E_STATUS* is used to define the type of exception handling required. If the exception signals about normal termination, then the component continues to execute the requested service, i.e., enters the phase Executing. If the exception signals about recoverable error, then the error recovery is attempted from the operation recover by executing lower layer operations. Otherwise the component is stopped, the exception is converted using the function *E_CONV2* and propagated upward.

```
e_handle =
 SELECT flag = Handling & (atomic_flag=TRUE or
   size(Inter)=0 or I_Crit(first(Inter)) < E_Crit2(exc2,state))
 THEN
   IF E_STATUS(exc2) = Ok THEN flag := Executing
   ELSIF E_STATUS(exc2) = Recov THEN flag := Recovering
   ELSE raise(E_CONV2(exc2)) || flag := Stopping END
 END;
```

The refinement process follows the graphical representation given in Fig.5. We refine the abstract specification FTDAComponent by introducing the activation of the lower layer components into the operation execute. Since the refinement step introduces an abstract specification of the lower layer component the refined specification FTDAComponentR also contains more detailed representation of the error recovery and interruption handling procedures. The specification of the lower layer components has the form of FTDAComponent. By recursively applying the same specification and refinement pattern, we eventually arrive at the complete specification of the layered system. At the final refinement step all remaining unimplementable mathematical structures are replaced by the corresponding constructs of the targeted programming language and code is generated. Therefore, the result of our refinement process is implementation of a fault tolerant reactive control system.

Due to a lack of space we have omitted a description of the case study which has validated the proposed approach. Its detailed description can be found in the accompanying technical report [9].

5. Conclusions

In this paper we proposed a formal approach to the development of reactive fault tolerant control systems. We created a generic pattern for modelling such systems. The proposed approach is based on recursive application and instantiation of the pattern via refinement. It can be summarized as follows:

1. Specify the external interface of the system which includes requests to execute services and interruptions. Instantiate a general model to reflect interruptions, particular to the services under construction.
2. Define the functions for evaluating criticality of exceptions and interruptions, conversion rules and atomicity indicators of the operations. Specify functionality of the component server together with fault tolerance mechanisms modelled as exception handling. Incorporate the interruption handler. Use the defined functions to design interruption and exception handlers.
3. Instantiate the specification pattern to model components on the lower layer of hierarchy. Refine the component server by introducing invocation of operations of these components. Verify correctness of the transformation by proofs.
4. Repeat step 3 until the lowest layer of hierarchy is reached.

The use of automatic tool supporting the B Method significantly simplified the required verification process. The proofs were generated automatically and majority of then was also proved automatically by the tool.

In [8] the formal semantics is given to services and layered architectures. However, the proposed formalization leaves the problem of interruption and exception handling aside.

Representation of exception handling in the development process has been addressed by Ferreira et al. [6]. They consider UML-supported development of component-based system and demonstrate how to integrate reasoning about exception handling in the development process. In our approach we integrate exception handling in the formal development process and also formalize the relationships between interruptions and exceptions.

The idea of reasoning about fault tolerance in the refinement process has also been explored by Joseph and Liu [12]. They specified a fault intolerant system in a temporal logic framework and demonstrated how to transform it into a fault tolerant system by refinement. However, they analyse a “flat” system structure. The advantage of our approach is possibility to introduce hierarchy (layers) and describe different exceptions and recovery actions for different layers.

Reasoning about fault tolerance in B has also been explored by Lano et. al [10]. However, they focused on structuring B specifications to model a certain mechanism for damage confinement rather than exception handling mechanisms.

Arora and Kulkarni [7] have done the extensive research on establishing correctness of adding the fault tolerance mechanisms to fault intolerant systems. Correctness proof of such an extended system is based on soundness of their algorithm working with the next-state (transition) relation. In our approach we start with an abstract specification of a system and develop a fault tolerant system by refinement, incorporating the fault

tolerance mechanisms on the way. Correctness of our transformation is guaranteed by soundness of the B method. Moreover, an automatic tool support available for our approach facilitates verification of correctness.

We argue that generality of the proposed approach and availability of the automatic tool support makes our approach scalable to the complex real-life applications. In the future we are planning to overcome the current limitations of the approach such as a simplistic representation of exceptions and state-insensitive handling of exceptions and interruptions. To achieve this, we are going to strengthen the proposed approach by introducing more sophisticated models of exceptions and interruptions. Moreover, it would be interesting to extend the proposed approach to incorporate reasoning about parallelism.

Acknowledgements The work reported in this paper is financially supported by IST-511599 EU Project RODIN.

References

1. 1. J.-R. Abrial. *The B-Book*. Cambridge University Press,1996.
2. J.-R. Abrial. *Event Driven Sequential Program Construction.* 2000, via http://www.matisse.qinetiq.com.
3. T.Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice.* Dependable Computing and Fault Tolerant Systems, Vol 3**.** Springer Verlag; 1990.
4. A.Avizienis. Towards Systematic Design of Fault-Tolerant Systems. *Computer* 30 (4), pp. 51-58. 1997.
5. F.Cristian. Exception Handling. In T.Anderson (ed.): *Dependability of Resilient Computers.* BSP Professional Books, 1989.
6. L.Ferreira, C.Rubira and R. de Lemos. Explicit Representation of Exception Handling in the Development of Dependable Component-Based Systems. Proceedings of HASE'2001. USA, October 2001.
7. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-time and Fault-tolerant Systems (FTRTFTS'2000),* Pune, India. 2000.
8. M.Broy. *Service-Oriented Systems Engineering: Modeling Services and Layered Architectures*. In Proc. Of FORTE 2003, pp. 48-61, Berlin, Germany, September 2003.
9. L.Laibinis and E.Troubitsyna. *Formal Service-oriented Development of Fault Tolerant.* TUCS Technical Report 648, 2004. http://www.tucs.fi/publications/insight.php?id=tLaTr04b&table=techreport
10. K.Lano, D. Clark, K. Androutsopoulos, P. Kan. Invariant-Based Synthesis of Fault-tolerant Systems. In Proc. of *Formal Techniques in Real-Time and Fault-Tolerant Systems. FTRTFT 2000,* LNCS vol. 1926, p. 46 -57. India, 2000.
11. J.-C. Laprie. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Vienna, 1991.
12. Z. Liu and M. Joseph. Transformations of programs for fault-tolerance, *Formal Aspects of Computing,* Vol 4, No. 5, pp. 442-469, 1992.
13. *MATISSE Handbook for Correct Systems Construction.* 2003. *http://www.esil.univ-mrs.fr/~spc/matisse/Handbook/*
14. B.Rubel. Patterns for Generating a Layered Architecture. In J.O. Coplien, D.C. Schmidt (Eds.). *Pattern Languages of Program Design*. Addison-Wesley. 1995.
15. S.Schneider. *The B Method. An introduction.* Palgrave2001.
16. Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 2001. Available at http://www.atelierb.societe.com/index uk.html.
17. Storey N. *Safety-critical computer systems*. Addison-Wesley, 1996.