

A Generic Approach to Structuring and Implementing Complex Fault-Tolerant Software

J. Xu

University of Durham, DH1 3LE, UK

B. Randell and A. Romanovsky

University of Newcastle upon Tyne, NE1 7RU, UK

Abstract

This paper addresses the practical implementation of means of tolerating residual software faults in complex software systems, especially concurrent and distributed ones. There are several inherent difficulties in implementing such fault-tolerant software systems, including the controlled use of extra redundancy and the mixture of different design concerns. In an attempt to minimise these difficulties, we present a generic implementation approach, composed of a multi-layered reference architecture, a configuration method and an architectural pattern. We evaluate our implementation approach using an industrial control application whose control software we equip with the ability to tolerate a variety of software faults. The preliminary evidence shows that our approach can simplify the implementation process, reduce repetitive development effort and provide high flexibility through a generic interface for a wide range of fault tolerance schemes.

Key Words — Architectural patterns, concurrent and distributed systems, coordinated atomic actions, fault-tolerant software, object orientation

1: Introduction

This paper investigates the issues concerned with the practical development of fault-tolerant software. Fault-tolerant software usually involves the introduction of software redundancy to normal program code. The difficulty in finding a non-intrusive way of incorporating redundancy into a complex system often hinders system development and implementation. Furthermore, in many cases the redundancy to be added is application-specific, and the developer has to address both application-dependent and redundancy-related concerns. This complicates further the task of implementing and maintaining fault-tolerant software.

In this paper, we will study how a multi-level reference architecture and a configuration method help to separate different concerns and give application programmers a simple environment for developing fault-tolerant software. We also investigate how an appropriate software pattern provides convenient support for implementing such software for concurrent and

distributed systems. We reported in [13] our experience using coordinated atomic (CA) actions as a system structuring tool to design a sophisticated control system for an industrial safety-critical application – The Fault-Tolerant Production Cell. The dependability problems addressed in [13] related to faulty sensors, actuators and mechanical devices used by the control system. Here we use the case study again to examine and evaluate our implementation approach, but this time mainly focusing on the problem of possible residual design faults in the control software system.

This paper is organised as follows. Section 2 discusses a general structuring framework for fault-tolerant software. Sections 3 and 4 describe our multi-level architecture and the associated reconfiguration method. Section 5 introduces the GSFT pattern – an architectural pattern for implementing complex fault-tolerant software, especially for concurrent and distributed systems. Section 6 uses the Fault-Tolerant Production Cell case study to illustrate the strengths and weaknesses of our approach.

2: Structuring Framework

We define a software system as a set of components which interact under the control of a design, and view the components themselves as systems at a lower level of abstraction in their own right [7]. In order to ease the task of constructing a fault-tolerant software system and control its complexity we believe it is crucial to separate different concerns properly. Our aim is that the users of fault-tolerant components (or FTC Users) should be responsible only for developing their own programs, using services provided by other programmers, without needing much knowledge of how fault tolerance is implemented. To separate different concerns further, the responsibilities involved in providing such fault tolerance can usefully be divided between:

Programmers of Fault-Tolerant Components (or FTC Programmers). They are responsible for developing components that tolerate certain sets of software faults. They may have to address both functional requirements and fault tolerance aspects. In particular, they must understand different software fault tolerance schemes and be able to develop diverse software variants and application-specific

adjudicators. They are required to select and customise an appropriate scheme according to application-specific requirements.

Programmers of Reusable Control Components (or RCC Programmers). They are responsible for providing the FTC programmers with a high-level and simple programming interface for the use of the various software fault tolerance schemes. They are also responsible for dealing with low-level implementation details.

In order to address the responsibilities of both the FTC and the RCC programmers, we will introduce a multi-level reference architecture for structuring fault-tolerant applications so that different concerns can be addressed properly at separate levels. The major idea behind our architecture is to hide the control part and low-level implementation details of a fault tolerance scheme from the FTC programmers. This enables the FTC programmers to focus *mainly* on functional requirements and leave the actual implementation of various software fault tolerance schemes to the RCC programmers. However, implementing a concrete scheme in an effective way is never an easy task. Although similar implementation issues have recurred many times in a variety of experimental studies and industrial applications [9], the application programmers, especially novices, still have a hard time understanding and reusing existing solutions. In an attempt to resolve this difficulty, we use pattern techniques [5] to document existing and well-proven experience, including our own experience in implementing a generic scheme for software fault tolerance [12].

2.1: Idealised Fault-Tolerant Components

An idealised fault-tolerant (FT) component [7] is a (well-defined) component which includes both normal and abnormal responses in the interface between interacting components, in a framework that minimises the impact of fault tolerance (e.g. extra redundancy) on system complexity (see Figure 1).

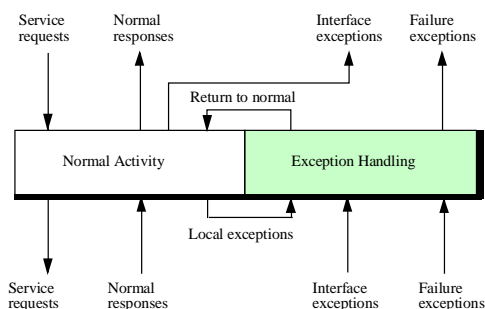


Figure 1 An idealised FT component [7]

We extend this framework to address three further concerns: i) degraded services, ii) concurrency, and iii) embedded redundancy for tolerating software faults.

Extended interface specification. In practice, when an exception is raised within a component, in many cases the corresponding exception handler cannot deliver the complete service requested by its environment, but possibly only a degraded one. Such responses are conceptually different from both normal responses and failure exceptions, and should be indicated by an appropriate specific exception. It is up to the calling environment to decide how to deal with a degraded service. Similarly, an abort exception should be distinguished from a failure exception. The former notifies the environment that something wrong happened inside the component but all possible effects that would affect the environment have nevertheless been undone, and the latter indicates that it cannot be guaranteed that all effects have been removed. Figure 2 illustrates the extended interface specification (in which each exceptional response must be indicated by the signalling of an appropriate exception).

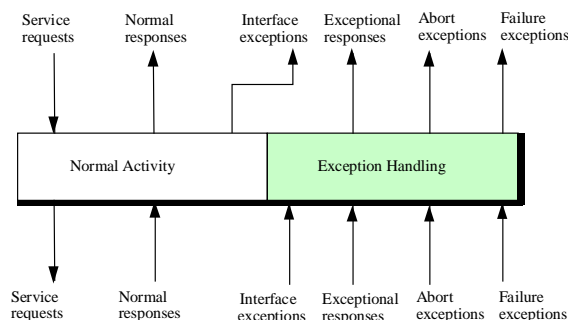


Figure 2 Extended interface of an FT component

Interface for concurrent requests. In order to support the development of concurrent and distributed fault-tolerant software, it is often useful to make concurrency explicit at the interface of a component. Linguistically, there are several concrete examples: the multi-function mechanism proposed in [2], multi-party interaction mechanisms [6], interacting processes [4] and the CA action scheme [11]. A component may be associated with m synchronous entry points. Service requests may be made concurrently through the entry points by m calling components. Within the component, there are exactly m roles that execute in parallel in response to service requests and possibly interact with each other to carry out the required computation.

Components with diverse design. A fault-tolerant component can be constructed as a containing component that encloses several diversely designed sub-components called variants and an adjudicator. Variants deliver the same service through independent designs and implementations, and the adjudicator selects a single, presumably correct, result from the results produced by variants. The containing component controls the execution of variants and determines the overall component output with the aid of the

adjudicator. This structure is also applicable to a component with m synchronous entry points. To provide the same service, each variant must have m roles to carry out the required concurrent computation, but may have a diverse internal design for the purpose of software fault tolerance.

2.2: CA Actions and Software Fault Tolerance

A CA action is a multi-threaded transactional mechanism which as well as co-ordinating multi-threaded interactions ensures consistent access to (external) objects in the presence of concurrency and potential faults.

The various approaches to software fault tolerance can be in general divided into two categories: dynamic redundancy [10] and masking redundancy [1]. A system with dynamic redundancy consists of several redundant components with just one active at a time. If a software error is detected, the active component is replaced by a spare one. Masking redundancy uses extra software components (called versions or variants) within a system such that the effects of one or more software errors are masked from the environment of that system. Here we discuss briefly two schemes for concurrent and distributed object systems by combining the CA action scheme with dynamic redundancy and masking redundancy respectively. For a given CA action A and its specification, we develop several variants independently from the same specification. The action A then serves as the *container* action, and within the container a set of nested CA actions serve as software variants. They each provide the actual functionality of A and together supply redundancy for coping with software faults.

Dynamic Redundancy (DR): In this scheme, the container action A controls the adaptive execution of action variants, which in effect involves the participating threads of A performing a sequence of one or more nested CA actions, depending on the errors encountered. Ideally, external objects of container action A will be checkpointed for the purpose of action recovery. (Whenever the state restoration of external objects is not feasible, appropriate compensatory operations will be performed instead.)

Masking Redundancy (MR): This scheme is essentially based on the concurrent execution of action variants nested in a container action and on determining the majority of the results produced by the variants in order to provide some form of fault masking. The container action delegates its roles to the nested action variants and afterwards adjudicates the results returned from the variants. Every participating thread of the container action is in effect forked into n sub-threads which will participate in n action variants respectively.

For any external object, n clones of it must be made so that n action variants can operate on these clones, not on the original object. When all the action variants are complete, for each external object the container action must determine a correct clone by executing an adjudication algorithm. The selected clone will replace the original object and the other clones and the original object will be discarded. This adjudication process in effect merges the sub-threads into the original threads of the container. If the container action aborts for whatever reason, all the clones must be discarded and only the unchanged, original objects retained.

2.3: Architectural Styles

Development tools and support mechanisms for implementing fault-tolerant software can be provided by a variety of techniques such as high-level programming interfaces, libraries of reusable components and development environments, singly or in combination. We propose here a general solution at system-level by developing an architectural style for constructing fault-tolerant applications, employing a library of reusable components. Our solution comprises:

- 1) A *reference architecture* that consists of multiple levels corresponding to different responsibilities and concerns.
- 2) A simple *configuration method* for selecting and configuring components within the reference architecture to meet particular application requirements.
- 3) A library of *reusable components*, which contains reusable experience and expertise in the domain of software fault tolerance.
- 4) An *architectural pattern* is used to capture well-proven solutions and to specify precisely relationships between the components and the ways in which they collaborate.

3: Multi-Level Reference Architecture

Figure 3 illustrates a static view of our reference architecture.

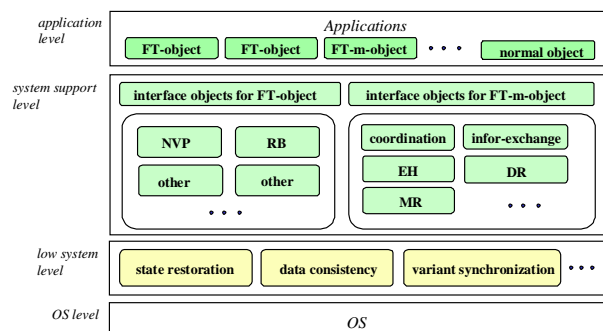


Figure 3 Reference architecture for FT applications

This system architecture is composed of several levels: i) the *application level* for the implementation of various applications which may include a set of fault-tolerant objects, ii) the *system support level* composed of interface objects and reusable control mechanisms, iii) the *low system level* that provides low-level mechanisms such as state restoration and variant synchronisation, and iv) the *OS level* such as UNIX or Windows NT.

Application Level. This level consists of application-specific objects. Prompted by dependability concerns, some critical objects may be implemented as fault-tolerant objects, and some objects may use or invoke fault-tolerant objects to perform their intended computation. Fault-tolerant objects must adhere to the standard interface characteristics of an idealised fault-tolerant component. Two kinds of fault-tolerant objects are supported: standard ones for sequential invocation and extended ones for concurrent invocation via m entry points, i.e. the FT- m -objects in the figure.

System Support Level. This level provides high-level support for the construction of fault-tolerant objects. There are two special categories of objects that specify interfaces between interacting objects: i) external interfaces and ii) generic FT interfaces. The external interfaces capture application-independent, external characteristics of a fault-tolerant object and specify an abstraction interface between the object and its users. The FTC programmers must follow this structuring framework (e.g. using the inheritance mechanism). The generic FT interface objects provide the FTC programmers with a high-level programming interface that facilitates the selection and customisation of various fault tolerance schemes.

Low System Level & OS Level. This level offers low-level services which are necessary for certain software fault tolerance schemes, including state saving and restoration, data consistency and variant synchronisation. Other services may be also implemented at this level. The OS level provides conventional OS capabilities.

4: Configuring Fault-Tolerant Objects

To implement a fault-tolerant object, the FTC programmers are responsible for developing required software variants (and an application-specific adjudicator if needed). With the aid of the reference architecture and reusable objects, the FTC programmers can define the fault-tolerant object simply by:

- 1) reusing the standard structuring framework specified by an external interface object to implement an application-specific interface to the FTC users, and
- 2) specifying a software fault tolerance scheme by selecting the corresponding control mechanism and

plugging in the (possibly application-specific) adjudicator, through a generic FT interface.

To request services from a generic FT interface object, the FTC programmers need to pass the references of the variants and the adjudicator to the interface. They may also have to specify the maximum number of processors required for the chosen scheme, and the objects that keep input and output data. The generic FT interface object is also an idealised fault-tolerant component, and so its execution will in general produce either: a normal outcome, an exceptional outcome (signalling a specific exception to the fault-tolerant object), an interface exception, an abort exception, or a failure exception.

5: Architectural Pattern

In this section we will introduce a pattern to detail further our solution to the problems that arise in designing and implementing fault-tolerant objects. It extends, to the case of concurrent and distributed systems, the sort of pattern devised by [3] based on our original scheme for structuring fault-tolerant software [12]. Our solution scheme describes a pre-defined set of reusable classes or objects, and details their responsibilities and relationships, as well as their cooperation.

5.1: Generic Software Fault Tolerance (GSFT)

This GSFT pattern combines the structured characteristics of an idealised fault-tolerant component with the extensibility and flexibility offered by the object-oriented approach.

The notation used in Figure 4 for inheritance, delegation, aggregation and classes has the usual meaning and is the same as the notation used in [5]. In Figure 4, the client object invokes services of the fault-tolerant object. The fault-tolerant object is derived from the external interface to conform to the interface characteristics of an idealised component. It requests services from the generic FT interface to execute software variants and the adjudication function.

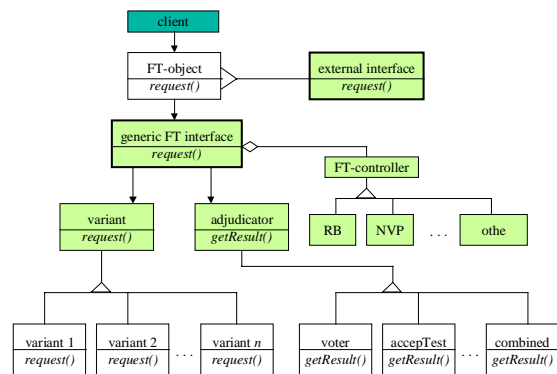


Figure 4 Structure of the GSFT pattern

The generic FT interface object i) requests services from software variants, ii) sends the results of the variant executions to the adjudicator, iii) receives results back from the adjudicator and iv) reports the results back to the fault-tolerant object (which returns the results back to the client in turn). This interface contains an FT controller from which the subclass for a special fault tolerance scheme can be derived. These subclasses are responsible for actually controlling the execution of software variants and the result adjudication. The variant is an abstract class that declares the common interface for software variants, and the adjudicator is also an abstract class that declares the common interface for adjudication functions. The voter, accepTest (i.e. the acceptance test) and combined (i.e. the combined use of voters and acceptance tests, etc.) can be derived from the adjudicator class.

The structure of Figure 4 is also applicable to concurrent programs. Figure 5 shows a slightly extended structure which provides software fault tolerance based on the CA action scheme. Clients 1, 2, ... and m are the participants of a CA action. They enter an FT-m-object synchronously by requesting services of the object. The generic interface contains an FT-m-controller from which the EH (concurrent exception handling), DR and MR subclasses etc. can be derived to implement a specified fault tolerance scheme.

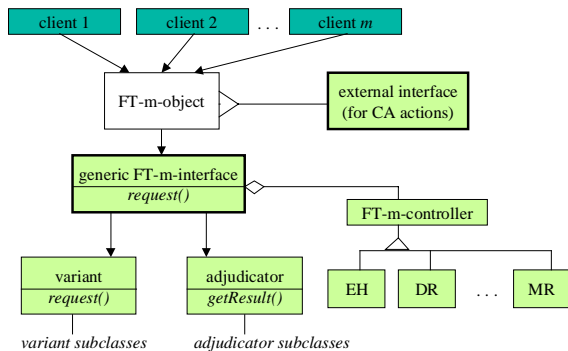


Figure 5 Extended structure for concurrent FT

It is difficult to describe the dynamic behaviour of fault-tolerant software systems in general. We present only a typical scenario here. Figure 6 illustrates the collaboration between objects in the pattern that represents the CA action scheme that uses masking MR to achieve fault tolerance. Multiple clients 1, 2, ..., and m invoke the services of the FT-m-object through m synchronous entry points and pass the references of external shared objects to it. The FT-m-object handles the required service by requesting a service from the generic FT-m-interface. It specifies the chosen scheme as MR and passes the references of external objects, variants and the voter to the interface. Note that both the FT-m-object and the interface component can refuse an invalid service request by signalling an interface

exception to their clients. The generic FT-m-interface then delegates the requested service to the MR controller. The controller invokes the execution of n functionally equivalent software variants. The voter attempts to adjudicate the results produced by the variants and select one of them as the correct or signal an exception if no majority is found. The MR controller then decides the final state of all external objects. Afterwards the generic FT-m-interface determines its own response to the FT-m-object based on the final state of the external objects. The FT-m-object finally returns an agreed result (i.e. the current state of the external objects) back to the client or signals an appropriate exception.

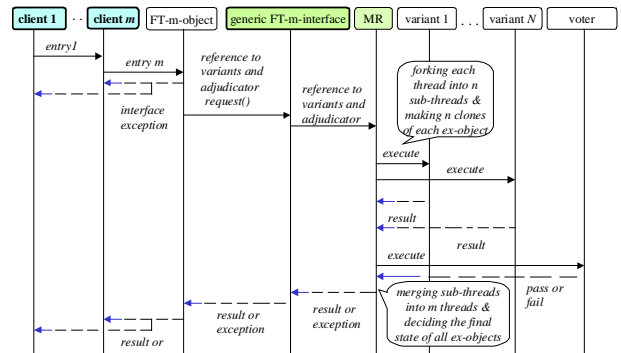


Figure 6 Interaction diagram for CA actions using MR

Controlling the execution of software variants: There are several key technical issues we have to deal with carefully when implementing a software fault tolerance scheme. One of them is how to control the execution of software variants. We provide an object-oriented solution here based on the Composite pattern introduced in [5]. Figure 7 shows the control structure based on the Composite pattern and the DR scheme. The DR controls the execution of variants by sending requests to class variant. The variant class is an abstract class that provides a common interface for a set of concrete variants. Apart from n subclasses implementing software variants, an additional subclass, called Controller, is implemented as an aggregate of those variants and performs the actual control operations. To control the execution of variants, a Controller object has to create and store a list of concrete variant objects of its sibling classes.

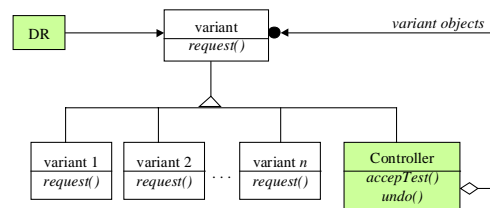


Figure 7 Control structure for CA actions using DR

According to the requests received, a DR object decides how many and which variants are needed and then adds the chosen variants to the `Controller` object.

6: Case Study

Our reference architecture and the GSFT pattern help determine the basic structure of the solution to the problem of building fault-tolerant software, but they do not specify a fully-detailed solution. They provide a scheme for a generic solution to a family of problems, rather than a prefabricated module that can be used “as is”. This scheme must be implemented according to the specific needs of a particular application in hand. In this section, we use a realistic case study to investigate various details specific to a concrete implementation.

An industrial production cell model, based on a metal-processing plant in Karlsruhe, Germany, was first created by the FZI (Forschungszentrum Informatik) in 1993. An extended model, called the “Fault-Tolerant Production Cell”, was introduced in [8]. This Cell consists of six devices: two conveyor belts (a feed belt and a deposit belt), an elevating rotary table, two presses and a rotary robot that has two orthogonal extendible arms (see Figure 8). These devices are associated with a set of sensors and actuators via which the control system can exercise control over the whole cell. The task of the cell is to get a metal blank from its “environment” via the feed belt, transform it into a forged plate by using a press, and return it afterwards to the environment via the deposit belt. Figure 8 illustrates a group of CA actions that constitutes a control system controlling the cell.

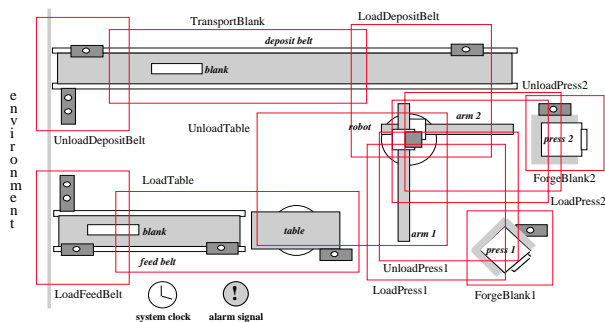


Figure 8 CA actions that control the Cell

6.1: Software Faults in the Control Program

A control program for the Fault-Tolerant Production Cell is a complex program with multiple concurrent control threads that must coordinate the interaction of multiple concurrent devices in a highly safe manner. It cannot be simply assumed that the control software itself will be free of error. Software faults might manifest themselves during the system operation time.

Apart from possible incomplete or inconsistent specifications of computation requirements, there are

other sources which may introduce software bugs into the control program. Since our control program involves concurrency and/or distribution, software faults can be introduced into the part that coordinates multiple concurrent objects. And though individual objects may meet their specifications, the specifications for cooperation between objects are often incomplete and inaccurate.

6.2: Dealing with Software Faults

For simplicity's sake, we investigate here situations involving software faults only, i.e. we assume that *only faults that can occur are software faults in the control program, and all the hardware components of the Production Cell are fault-free.*

The major software faults that might remain in the control program include i) interaction faults that occur when the interaction relationship between roles of a CA action, between interacting CA actions, or between the control program and the Production Cell has not been specified properly or analysed sufficiently, and ii) timing faults that occur when an operation, a role or a CA action is not completed in a pre-specified amount of time. Software faults may be also involved in error detection and recovery mechanisms for CA actions. We regard these mechanisms as part of a CA action support mechanism, and assume that the support mechanism itself has been tested extensively and is fault-free. To provide support for this assumption, these mechanisms are implemented as reusable and well-tested components in our GSFT pattern. The CA action support mechanism also contains a concurrency-control mechanism for controlling access to external objects. In the Fault-Tolerant Production Cell, the external objects (i.e. metal blanks) cannot be shared by two or more concurrent CA actions due to safety-related concerns. We choose to use a simple concurrency-control mechanism to minimise the probability of residual software bugs. This mechanism allows a monitoring object to get the state information of a blank concurrently with a running CA action.

Software faults such as interaction faults and timing faults have not been addressed adequately in the initial requirements for the Fault-Tolerant Production Cell [8], while the high-level specification for implementing the control program (such as the COALA specification used in [13]) focused mainly on hardware device, sensor and actuator failures. For these reasons, and the inadequacy of available fault removal techniques, we had to admit that despite our best efforts to the contrary, software faults might exist in our control program. This indeed turned out to be the case since we have observed subsequently that some software design faults occur while our control program is in use. For example, a transient software fault occurs in the FZI simulator when

arm 1 of the robot is required to place the blank into an unoccupied press. The arm performs most specified operations but it fails to drop the blank into the press. This fault manifests itself only occasionally and makes its removal extremely difficult. Another software bug appears in our control program in the form of an interaction fault. This interaction fault manifests itself only when more than two blanks are placed into the system. Under certain conditions at run-time, two interacting CA actions can be involved in a deadlock situation.

It is therefore evident that it is worth trying to deal with residual software faults that may manifest themselves while the Production Cell is in operation. There are a variety of software fault tolerance schemes we may consider. For transient software faults, a simple “re-try” strategy is often effective. However, for most software design faults (e.g. the interaction fault in our control program) a rollback and re-try approach is insufficient. Instead, tolerance of such software faults must rely on the application of design diversity. We found that deliberately-chosen diverse data structures and algorithms are less likely to fail simultaneously. Such an approach is often called “enforced diversity”. For example, based on our CA action-based design of the `LoadPress1` CA action were developed using enforced diversity. Variant One involves several concurrent activities and two concurrent nested actions `RotateRobot` and `MovePress1toMiddle`. Variant Two is designed to provide the identical functionality but following a simpler algorithm without any concurrency. Because the interaction relationship between these nested CA actions is essentially diverse in two variants, the probability that the variants fail identically should be reasonably low.

Figure 9 illustrates the control structure of the `LoadPress1` CA action using the DR scheme to tolerate software faults. CA action `LoadPress1` is designed as a container action which contains two diversely designed variants. Normally, the container action just executes the first variant. If no error is detected, then the container action ends with a normal outcome. However, if an error is detected by either an assertion statement or the acceptance test (at the end of the first variant), this error must be reported to the container action.

When an error occurs in Variant One, an appropriate exception must be signalled from the variant to the container. The corresponding handler will then be called, which has to restore the related device objects and the blank object to their original state — the state before the execution of action `LoadPress1`. After finishing the restoration, the handler will invoke the second variant in the hope that the same software error will not occur again. In the worst case that an error (not

necessarily the same one) takes place again, it is always possible for action `LoadPress1` to signal an `abort` exception, if the state restoration is successful, or a `failure` exception to its containing action. By way of example, we consider the arm 1 position error again. A handler must handle the exceptional situation where the nested action `RetractArm1` fails to place the arm in a correct position. This handler can use both a re-try strategy and a diversely designed variant for the action. We outline the functionality of the handler as follows.

Handler for Arm 1 Position Error: Re-execute the `RetractArm1` action and check the arm position. If the same error persists even after a pre-determined number of re-executions, then restore both the device objects (i.e. `Robot` and `Press1`) and the blank object to their original state. Invoke the second variant. If the variant signals an exception, instead of a normal outcome, then restore the state again. If the state restoration is successful, signal an `abort` exception to the container action, otherwise signal a `failure` exception.

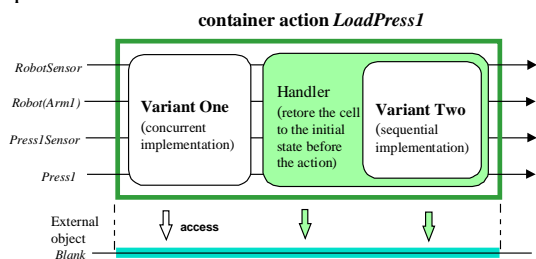


Figure 9 `LoadPress1` as container action

6.3: Implementation of the Control Program

Our implementation of the control program uses 12 main CA actions to address the safety requirements. (Each action concerns a situation in which two physical devices, e.g. the robot arm and the press, have to coordinate their activities.) The application of software fault tolerance techniques at this level enhances the ability of the control program to handle software errors so as to improve both system reliability and safety.

We chose Java as the implementation language and used an object-oriented solution based on inheritance and delegation. It is evident that the GSFT pattern enables reuse of reusable support mechanisms and simplifies to a great extent the development of a fault-tolerant object by separating different concerns. However, the GSFT pattern provides no support for addressing functional aspects of a CA action. While a pattern captures key properties of an architectural solution, many implementation details are suppressed. We feel that patterns should be treated as just one of many important tools in a toolkit of supporting software development. During the testing phase and the demonstration of our implementation, the control system proved highly robust

when dealing with hardware-related faults. It is however difficult to determine precisely the ability of the control program to tolerate software faults although we know that the ability depends mainly upon the error-detection coverage provided by a combination of the acceptance test, executable assertion statements and run-time checks by the hardware system. We found that introducing a set of meaningful software faults into the control system that could pass tests and checks is not an easy task at all. Nevertheless, some events that occurred at run-time provided quite encouraging feedback. For example, a previously unknown software fault remaining in the FZI simulator itself was detected successfully by the acceptance test of a CA action and recovered by the re-try operation associated with that action. With the aid of the GSFT patterns and reusable mechanisms, the additional cost of achieving software fault tolerance was contributed mainly by the cost of developing action variants. We are now in the stage of collecting experimental data for further dependability and performance-related evaluation.

7: Conclusions

The design and implementation of fault-tolerant software for critical computer applications are complex and error-prone tasks – they need an architectural solution that separates different concerns and makes certain aspects transparent to a given type of programmers. We have developed a multi-level reference architecture for implementing fault-tolerant software, which separates the application-specific functionality, interfaces to fault tolerance schemes and application-independent control mechanisms. Such separation has helped us to promote better understanding of both functional and non-functional aspects of a fault-tolerant application and might have resulted in increased dependability of the real application.

Our approach provides support for implementing fault-tolerant software using pre-defined classes and run-time libraries. In principle, it does not require special pre-processors or builders such as the architecture introduced in or a new programming language with particular syntax for specifying fault tolerance schemes. Since different groups of objects are located at different levels and low-level services and implementation-details are hidden from higher-level objects, our reference architecture may readily be ported to different platforms, without requiring any direct support from a special underlying operating system.

Although there are some application-specific solutions to the implementation of fault-tolerant software, it remains difficult to reuse fault-tolerant software components directly for complex applications. We have found that pattern techniques are very promising with regard to promoting widespread reuse of our architectural solutions for implementing complex fault-

tolerant software. We have used the GSFT pattern to equip a control program for the Fault-Tolerant Production Cell with the ability to cope with software faults. The GSFT pattern, together with the object-oriented structure and high-level control abstractions like CA actions, helps greatly in reducing the development effort, simplifying the implementation process and permitting a high level of flexibility and extensibility.

Acknowledgements

This work was supported by the ESPRIT DeVa project, the EPSRC IBHIS project and the *e*-Demand project.

References

- [1] A. Avizienis, "The *N*-version approach to fault-tolerant software," *IEEE Trans. Soft. Eng.*, vol.11, no.12, pp.1491-1501, 1985.
- [2] J.P. Banâtre, M. Banâtre and F. Ployette, "The concept of multi-functions: a general structuring tool for distributed operating systems," in *IEEE ICDS-6*, pp.478-485, 1986.
- [3] F. Daniels, K. Kim and M.A. Vouk, "The reliable hybrid pattern: a generalized software fault tolerant design pattern," in *Int. Conf. PloP'97*, pp.1-9, 1997.
- [4] N. Francez and I.R. Forman. *Interacting Processes: a multiparty approach to coordinated distributed programming*, Addison-Wesley, 1996.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
- [6] Y-J. Joung and S. A. Smolka, "A comprehensive study of the complexity of multiparty interaction," *Journal of the ACM*, vol. 43, no. 1, pp.75-115, 1996.
- [7] P.A. Lee and T. Anderson. *Fault Tolerance: principles and practice*, Second Edition, Springer-Verlag, 1990.
- [8] A. Lötzbeyer, "Task description of a Fault-Tolerant Production Cell," Version 1.6, available from <http://www.fzi.de/prost/projects/korsys/korsys.html>, 1996.
- [9] M.R. Lyu (ed.). *Software Fault Tolerance*, Wiley, 1995.
- [10] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Soft. Eng.*, vol. SE-1, no. 2, pp.220-232, 1975.
- [11] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery," in *25th Int. Symp. Fault-Tolerant Comput.*, pp.499-508, Pasadena, 1995.
- [12] J. Xu, B. Randell, C.M.F. Rubira-Casavara and R.J. Stroud, "Toward an object-oriented approach to software fault tolerance," in *Recent Advances in Fault-Tolerant Parallel and Distributed Systems* (eds. D.K. Pradhan and D.R. Avresky), IEEE CS Press, pp.226-233, 1995.
- [13] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A. Zorzo, E. Canver, and F. von Henke, "Rigorous development of a fault-tolerant embedded system based on coordinated atomic actions," *IEEE Trans. Computer*, Feb. 2002.