GENERIC FAULT TOLERANT SOFTWARE ARCHITECTURE: MODELING, CUSTOMIZATION AND VERIFICATION

YUAN LING

(B.Sc. Wuhan University, China)

(M.En. Wuhan University, China)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2007

Acknowledgement

I would like to express my deep and sincere gratitude to my supervisor, Professor Jin Song DONG. His wide knowledge and logical way of thinking have been of great value for me. His understanding, encouraging and constructive comments have provided a good basis for the thesis and other works.

I wish to express my warm and sincere thanks to co-supervisor Dr. Jing Sun. His valuable advice and friendly help have been very helpful for my works. I also owe thanks to my lab-mates and friends for their help, discussions and friendship.

I would like to thank the numerous anonymous referees who have reviewed parts of this work prior to publication in journals and conference proceedings and whose valuable comments have contributed to the clarification of many of the ideas presented in this thesis.

This study received financial support from the National University of Singapore. The School of Computing also provided the finance for me to present paper in the conference overseas. For all this, I am very grateful.

I owe my loving thanks to my family members for their love, encouragement and financial support in my years of study. They have lost a lot due to my research abroad. Without their encouragement and understanding, it would have been impossible for me to finish this work.

Contents

7 Mechanical Verification of developed Safety Critical Distributed Systems guided by GFTSA 107

Summary

Distributed system often gives rise to complex concurrent and interacting activities. The distributed systems with high reliability requirements make the development of such systems more complicated. This thesis demonstrates a series of modeling, customization and verification of generic fault tolerant software architecture for guiding the development of distributed systems with high reliability requirements.

In this thesis, we first propose a novel heterogeneous software architecture, namely Generic Fault Tolerant Software Architecture (GFTSA), which incorporates fault tolerant techniques in the early system design phase. The proposed GFTSA combines several widely used basic software architecture styles to guide the development of distributed systems involving the cooperative & competitive concurrency. The fault tolerant techniques incorporated in GFTSA can deal with not only the exception the influence of which is limited within a single component, but also the exception which can affect the control flows of more than one component within a system.

Second, we formally model the GFTSA by using the Object-Z language, and formally reason about the fault tolerant properties of GFTSA. The formalisms of a software architecture can provide precise, explicit, common idioms & pattern s to the system designers. The formal language Object-Z based on set theory and predicate logic can capture the static and dynamic system properties in a highly structured way. Based on the reasoning rules of Object-Z, we can derive the fault tolerant properties from the GFTSA model to verify that GFTSA can preserve the

fault tolerant properties.

Third, we build a template based on the Object-Z model of GFTSA by using the XML-based Variant Configuration Language (XVCL) technique. This template can be reused in the development of distributed systems with high reliability requirements. By customizing this template, we can auto-generate the Object-Z models for the developed systems. A case study of Sales Control System (SCS), a specific mission critical distributed system, is presented to demonstrate the customization process. Following the reasoning rules of Object-Z, we can formally reason about the fault tolerant properties of SCS based on the generated Object-Z model from the template.

Fourth, we embed the formal GFTSA model in the Prototype Verification System (PVS) environment to achieve mechanical verification support for reasoning about the fault tolerant properties. In addition, we build a template based on the PVS model of GFTSA by using the XVCL technique. By customizing this template, we can auto-generate the PVS models for the developed safety critical distributed systems guided by GFTSA. Based on the generated PVS models, we can mechanically verify the fault tolerant properties of the developed systems by using the theorem prover of PVS. A case study of Line Direction Agreement System (LDAS) is presented to illustrate the customization process and mechanical verification.

Finally, we propose a template approach for the auto-generation of specifications and proof obligations at the customized system level from the GFTSA. By customizing this template, we can generate not only the formal models of safety critical distributed systems, but also the proof scripts for the fault tolerant properties of such systems. Based on the generated formal models and proof scripts, we are able to mechanically verify the fault tolerant properties in batch mode of PVS by using ProofLite technique. A case study of Electronic Power System (EPS) is presented to demonstrate the customization process and mechanical verification.

Part of the work in this thesis has been published in the journal *IEEE Transactions on Reliability* [88], and international conference *APSEC'06* [87].

List of Figures

Chapter 1

Introduction

1.1 Motivation and Goals

A distributed system can be viewed as a system composed of a set of concurrently interacting activities at different locations that cooperate with each other to perform a joint task [13]. Distributed systems are becoming increasingly widespread in business and scientific computing environments, which often give rise to complex concurrent and interacting activities. In practice, different kinds of concurrency might co-exists in a distributed system, which thus make the task of developing distributed systems complicated. Due to no small measure to their complexity, distributed systems are prone to faults and errors. For the distributed systems with high requirements for reliability [38], fault tolerant techniques are necessary, which can provide a practical way to improve the dependability of such systems [40, 83]. The concern of the fault tolerance makes the development of distributed systems more complicated [15]. Software architecture is identified as a critical design methodology which can ease the complexity of the development of distributed systems, as software architecture can provide a generic framework to guide the development of distributed systems[23, 70, 10]. How to incorporate fault tolerant techniques with functional aspects in the software architecture level is a new research area that has recently gained considerable attention. Existing work in this area mostly emphasizes the creation of fault tolerance mechanisms[60, 63]; descriptions of software architectures with respect to their reliability properties[33, 52]; and the evolution of component-based software architectures by adding or changing components to guarantee reliability properties[18, 26, 27]. In this thesis, we propose a novel heterogenous software architecture, namely Generic Fault Tolerant Software Architecture (GFTSA), which incorporates fault tolerant techniques in the early system design phase. GFTSA can provide a generic framework to guide the development of distributed systems involving not only different kinds of concurrency, but also high reliability requirements.

Good understanding and precise representation of software architecture can lead to reliable system implementations based on this architecture[9, 34]. The well-defined semantics & syntax make formal modeling techniques suitable for precisely specifying, and formally verifying architecture designs[45, 47, 69, 43, 44, 19, 42]. The formal language Z[76, 77] has been used to formalize several software architecture styles[1, 70]. Z is a formal specification language based on set theory and predicate logic, which can capture the static and dynamic properties of software architecture. Object-Z[21, 20, 74] is an extension of the Z formal specification language to accommodate object orientation. Compared to formal language Z, Object-Z can improve the clarity of large specifications through enhanced structuring, and help the system designers to reuse the GFTSA model via inheritance & instantiation mechanisms. In order to provide common idioms & patterns of GFTSA to the system designers, we investigate to formally model GFTSA by using the Object-Z language. Based on the Object-Z model of GFTSA, we propose to formally reason about the fault tolerant properties of GFTSA following the reasoning rules of Object-Z[72].

GFTSA is proposed to guide the development of distributed system with high reliability requirements. How the GFTSA model can be reused in the development of specific distributed systems is the next issue we need to tackle. The GFTSA model can be customized into the formal models of specific systems by using the inheritance & instantiation mechanisms of Object-Z. In this thesis, we propose to make such customization process more efficient and systematic. The XML-based Variant Configuration Language (XVCL) [36, 75, 35] is a meta-programming technique developed to facilitate building flexible, adaptable, and reusable software artifacts. Following the mechanisms of XVCL, we propose to build a template for the customization of GFTSA as generic, adaptable fragments based on the Object-Z model of GFTSA. By customizing this built template, we can generate the Object-Z models of specific systems automatically. Based on the reasoning rules of Object-Z, we also can formally reason about the fault tolerant properties of such systems.

Object-Z, a highly expressive formal language, can capture the properties of models in an explicit and compact way. Even though Object-Z is a good modeling techniques that can provide precise analysis and documentation, Object-Z lacks of tool support for mechanical verification, therefore, the formal reasoning about the GFTSA model and specific system models customized from GFTSA are all manual-based, which are laborious and error-prone. In this thesis, we investigate to embed the GFTSA model in Prototype Verification System (PVS)[56, 55] to make the verification more systematic, since the theorem prover of PVS can provide mechanical proof support for the verification. The Prototype Verification System (PVS) is a proof system developed at SRI. PVS has a powerful interactive theorem prover and its automation suffices to prove many results automatically, which has been applied successfully to large and difficult application in both academic and industrial settings[31, 64]. We also propose to build a template based on the PVS model of GFTSA by using XVCL technique. When developing distributed systems with high reliability requirements guided by GFTSA, we can mechanically verify the fault tolerant properties of developed systems based on the generated PVS models from this built template.

The theorem prover of PVS can help us mechanically verify the properties of models, which offers a collection of powerful primitive proof commands that are applied interactively under user guidance. The primitive proof commands input by user to verify one specific property can constitute the proof script for this property. In the batch mode of PVS, we can apply the proof script directly to the theorem prover of

PVS to verify one specific property, which does not require inputting each primitive proof command interactively. By customizing the generic proof scripts, we can get the proof scripts for the developed distributed systems, and apply them to the theorem prover of PVS to verify the fault tolerant properties of developed systems in batch mode. Since ProofLite [53] technique can provide user-friendly interface of batch mode execution and interactive proof scripting notation to the system designers, we investigate to use it in our template approach. As the proof scripting notation supported by ProofLite enables a semi-literate proving style where specification and proof scripts reside in the same context, we investigate to extend the built template based on the PVS model of GFTSA to involve not only generic PVS specification, but also generic proof scripts for the generic fault tolerant properties by using the XVCL and ProofLite techniques. By customizing this template, we can generate both PVS models, and proof scripts for the developed systems. Based on the generated PVS specification and proof scripts, we can mechanically verify the fault tolerant properties of developed systems in batch mode of PVS supported by ProofLite technique.

1.2 Thesis Outline and Overview

The thesis is structured into 8 chapters. Chapter 2 is devoted to an overview of the formal language Object-Z, the XVCL technique for customization process, the PVS and ProofLite techniques for mechanical verification.

In chapter 3, we propose a novel heterogeneous software architecture, namely Generic Fault Tolerant Software Architecture (GFTSA). We describe the software architecture style and fault tolerant techniques involved in GFTSA.

In chapter 4, we formally model GFTSA by using the Object-Z language. Based on the Object-Z model of GFTSA, we formally reason about the fault tolerant properties of GFTSA, following the reasoning rules of Object-Z.

In chapter 5, we build a template based on the Object-Z model of GFTSA by using the XVCL technique. This template can be reused in the high level model design of distributed systems with high reliability requirements via customization process. A case study of Sales Control System (SCS) is presented to illustrate the customization process.

In chapter 6, we embed the formal GFTSA model in the PVS environment to achieve mechanical verification support for reasoning about the fault tolerant properties. Several significant fault tolerant properties of GFTSA are mechanically verified by using the theorem prover of PVS. In addition, we build a template based on the PVS model of GFTSA by using the XVCL technique. This template can be reused in generating the PVS models of developed distributed systems guided by GFTSA. The fault tolerant properties of developed systems can be mechanically verified based on the generated PVS models.

In chapter 7, we present two case studies to illustrate the mechanical verification of safety critical distributed systems. A case study of Line Direction Agreement System (LDAS) is presented to demonstrate that we can generate the PVS model of LDAS from the template based on the PVS model of GFTSA. Based on this generated model, we can mechanical verify the fault tolerant properties of LDAS by using the theorem prover of PVS. By summarizing the proof scripts for the fault tolerant properties of safety critical distributed systems, we extend the template based on the PVS model of GFTSA to involve the generic proof scripts. By customizing this template, we can generate not only PVS specification, but also proof scripts for the fault tolerant properties of developed systems guided by GFTSA. Based on the generated PVS models and proof scripts, we can mechanically verify the fault tolerant properties of developed systems in batch mode of PVS. Another case study of Electronic Power System (EPS) is presented to demonstrate the customization process and mechanical verification in batch mode of PVS.

Chapter 8 gives the conclusion of the thesis and future work.

Chapter 2

Background

This chapter sets the context for the later chapters, giving notations and brief technical outlines of Object-Z, XVCL, PVS and ProofLite.

2.1 Object-Z

Z[76, 77, 29] is a formal specification language based on set theory and predicate logic. Object-Z[20, 74] is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring. The essential extension to Z given by Object-Z is the *class* construct which groups the definition of a state schema and the definitions of its associated operations. A class is a template for *objects* of that class: for each such object, its states are instances of the state schema of the class and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class. Syntactically, a class definition is a named box. In this box, the constituents of the class are defined and related. The main constituents are: a visible list, a state schema, an initial state schema and operation schemas. We consider a simple example *queue* to illustrate the basic features of Object-Z. The essential behavior of this system is to receive a new message or send a message, which needs to preserve the FIFO property.

$Queue[Item]$

$\upharpoonright(\textsc{Init}, Join, Leave)$ [visibility list]

$items : \mathrm{seq}\, Item$ [state schema]

$\textsc{Init}$

$items = \langle\ \rangle$ [initial state]

Join

$\Delta(items)$

$item? : Item$

$items' = items \frown \langle item? \rangle$

Leave

$\Delta(items)$

[operation schema]

$item! : Item$

$items \neq \langle\ \rangle$

$items = \langle item! \rangle \frown items'$

The *Queue[Item]* class schema is generic with the parameter *Item* representing the type of *items* in the *queue*. The visible list specifies the interface between objects of class schema, and their environment. The state variable *items* is declared in the state schema, which would be changed by the operations of class. The INIT schema defines the initial state of the state variable. The *Join*, and *Leave* operation schemas specify that one *item?* joins the *queue*, and one *item!* leaves the *queue*, besides the state transformations of variable *items*.

2.2 XML-based Variant Configuration Language (XVCL)

XVCL[36, 35, 75, 89] is a meta programming technique developed to facilitate building flexible, adaptable, and reusable software artifacts. When developing an XVCL solution, we partition a problem description(e.g. a software specification, or a software program) into generic, adaptable meta-components called x-frames. Each x-frame contains a fragment of problem description, called Textual Content. The Textual Content is written in a base language, which can be any language, such as Z specification language, or Java programming language.

XVCL can be seen as a meta-language whose commands direct adaption of x-frames. Textual Content in x-frames is instrumented with XVCL commands for change. The XVCL commands mark the anticipated variation points in x-frames, injecting flexibility into their Textual Contents. The x-frame adaption process includes x-frame composition and customization. The ⟨ value-of expr=“?@var?”/⟩ command marks the variant point as expression *var*, which can be customized by a ⟨*set*⟩ command in the ancestor x-frame. The XVCL command ⟨*break*⟩ command marks a place in the x-frame at which the x-frame can be customized by an ⟨*insert*⟩ command declared in the ancestor x-frames.

X-frames related by ⟨*adapt*⟩ commands form an x-framework. The specification x-frame, *SPC* for short, specifies what variant requirements you need in a specific system. The *SPC* specifies how to adapt the x-framework in order to accommodate required variants. The *SPC* becomes a root of an x-framework. During x-framework processing, the XVCL processor interprets the XVCL commands contained in the *SPC*, traverses an x-framework, performs adaption by executing XVCL commands embedded in x-frames, and emits code components for a specific system.

XVCL is an adaption domain-independent language, method and tool. XVCL performs best in immature, poorly understood and evolving domains and in domains where frequent changes occur in both large and small granularity levels.

2.3 Prototype Verification System (PVS)

PVS[57, 59, 68, 58] is an integrated environment for formal specification and formal verification. It has been developed at SRI International Computer Science Laboratory for more than 25 years and used intensively for many practical complex systems. The distinguishing feature of PVS is its integration of an expressive specification language and powerful theorem-proving capabilities. The specification language of PVS augments higher-order logic with a sophisticated type system containing predicate subtypes and dependent types. In order to support modularity and reuse, the specifications are logically organized into parameterized *theories*. The *theories* are linked by *import* and *export* lists.

A theory consists of a sequence of *declarations*, which provide names for types, constants, variables, and formulas. *Type* declarations are used to introduce new type names to the context by using one of the keywords *TYPE*, and *TYPE+*. *Variable* declarations introduce new variables and associate a type with them. *Constant* declarations introduce new constants, specify their type and optionally provide values. Since the specification language of PVS is higher order logic based, the *constant* can refer to functions and relations, as well as the usual (0-ary) constants. *Formula* declaration introduces *axioms*, *assumptions*, *lemmas*, and *obligations*. The expression that makes up the body of the *formula* is a boolean expression. The identifier associated with the declaration may be referred during proofs. The specification language offers the usual set of expression constructs, including logical and arithmetic operators, quantifiers, lambda abstractions, function application, tuples, and a polymorphic *IF-THEN-ELSE*. Expressions may appear in the body of a formula or constant declarations, or as an actual parameter of a theory instance. The type-checker tool of PVS can check the syntactic consistency of the specification, such as undeclared names and ambiguous types.

The theorem prover of PVS maintains a proof tree. Each node of the proof tree can be considered as a proof goal. Each proof goal is a *sequent* consisting of a sequence of formulas called *antecedents* and a sequence of formulas called *consequents*. The intuitive interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the consequents. The proof tree starts off with a root node of the form $\vdash A$, where A is the theorem to be proved. PVS proof steps build a proof tree by adding subtrees to leaf nodes as directed by the proof commands, which are prompted by the users. Once a sequent is recognized as *true*, that branch of the proof tree is terminated. All the branches of the proof tree have been terminated means that the theorem is proved successfully. A PVS proof command provides the means to construct proof trees when applied to a sequent. The execution of PVS proof commands can either generate further branches, or complete a branch and move the control over to the next branch in the proof tree. These commands can be used to introduce lemmas, expand definitions, apply decision procedures, eliminate quantifiers, and so on. For example, the primitive proof command *flatten* can deal with propositional by simplifying disjunctive in a formula, and the *assert* command can carry out quantifier rules, induction, simplification by using decision procedures for equality and linear arithmetic.

2.4 ProofLite Technique

ProofLite[1], a PVS tool, extends the theorem prover interface with a batch proving utility and a proof scripting notation. ProofLite enables a semi-literate proving style where specification and proof scripts reside in the same file. ProofLite can provide a user-friendly interface to a PVS batch execution by including the command line utility *proveit* that executes the theorem prover in batch mode on a *.pvs* file and rerun all its proofs. The proof scripting notation provided by ProofLite is written in specially formatted comments that resides in regular *.pvs* files. Below is a simple example, *thms.pvs*, to illustrate the command line utility *proveit* and proof scripting notation.

```
thms: THEORY
 BEGIN
  a, b: VAR real
  th1: LEMMA a*a >=0
  %|- th1: PROOF (grind) QED
  th2: LEMMA a <= b IMPLIES a*abs(a) <= b*abs(b)
  %|- th2: PROOF
  %|- (then
  %|- (skip)
  %|- (spread (case " a >= 0")
```

---

[1] The ProofLite is electronically available from `http://research.nianet.org/~munoz/ProofLite`.

```
..........
 %|-        (assert)))))
 %|- QED
END thms
```

In this *thms* theory, *th1* and *th2* are two *LEMMAS* which need to be proved. Following each *LEMMA*, there is a proof script for this *LEMMA* written by the ProofLite proof scripting notation. Each line of proof script is preceded by the special comment %| −. The ProofLite utility *proveit thms* automatically installs proof scripts into their respective formulas when processing the *thms.pvs* file, writes the output into *thms.out* to show the result of proof.

Chapter 3

Generic Fault Tolerant Software Architecture – GFTSA

In this chapter, we propose a novel heterogeneous software architecture, namely Generic Fault Tolerant Software Architecture (GFTSA).

3.1 Introduction

Different from non-distributed systems, distributed systems may involves different concurrent and interacting activities, which thus require a generic supporting framework for controlling & coordinating those concurrent activities[61]. Two kinds of concurrency are mostly discussed in this context: competitive, and cooperative. Competitive concurrency indicates that concurrent activities compete for some common resources, but without explicit cooperation. Cooperative concurrency means that concurrent activities cooperate & communicate with each other[30].

Software architecture can provide a generic framework to guide the development of distributed systems [10]. Software architecture styles, such as pipe-and-filter[2], can only guide the development of distributed systems with cooperative concurrency. Some other basic software architecture styles, such as repository style[3], can only guide the development of distributed systems with competitive concurrency. However, many distributed systems involve both cooperative, and competitive concurrency. We propose a novel heterogeneous software architecture, namely Generic Fault Tolerant Software Architecture (GFTSA), which combines several widely used basic architecture styles to guide the development of distributed systems involving both cooperative and competitive concurrency.

Due to no small measure to the complexity of distributed systems involving competitive & cooperative concurrency, distributed systems are prone to fault and errors. For the distributed systems with high reliability requirements, fault tolerant techniques are necessary, which can provide a practical way to satisfy the reliability requirements of such systems [62, 40, 83]. When faults occur and cause exceptions in the distributed systems, their consequences may not always be limited to one system component [5]. Therefore, the fault tolerant techniques, which are used to deal with the exceptions occurred in the distributed systems, may require stepping outside the boundaries of a computer system. The fault tolerant techniques, namely *idealized fault tolerant component*[4, 41] and *coordinated error recovery mechanism*[11, 24, 84, 61], are incorporated in GFTSA to facilitate the recovery from exceptions that affect both the computer system, and its distributed environment.

How to integrate fault tolerant techniques with functional aspects in the software architecture level is a new research area that has recently gained considerable attention. Existing work in this area mostly emphasizes the creation of fault tolerant mechanisms[32, 60, 63]; descriptions of software architectures with respect to their reliability properties[66, 78, 33, 52]; and the evolution of component-based software architectures by adding or changing components to guarantee reliability properties[18, 25, 26, 27]. For our proposed software architecture, we incorporate fault tolerant techniques in GFTSA in the early system design phase.

The remainder of the chapter is organized as follows. Section 2 gives the illustration of software architecture style involved in GFTSA, and the overall literal description of GFTSA. Section 3 presents the fault tolerant techniques incorporated in GFTSA, and illustrates how these fault tolerant techniques deal with the exceptions occurred in the distributed environment. Section 4 concludes the chapter.

3.2 Software Architecture Style of GFTSA

The software architecture is the structure of the system, which comprises software components, the externally visible properties of those components, and the relationships between them. In order to provide a generic framework to guide the development of distributed systems involving cooperative & competitive concurrency, we propose a novel heterogenous software architecture, namely Generic Fault Tolerant Software Architecture (GFTSA). GFTSA can help develop the distributed system with the ability to tolerate faults, namely *FTS* (Fault Tolerant System), which is composed of a set of *Objects*, a set of *Connectors*, a set of *SharedResources*, and a *CoordinatingComponent*, as shown in Figure 3.1.

An architecture style defines a family of systems in terms of a pattern of structural organization. This provide a vocabulary of components and connector types, and a set of constraints on how they can be combined. The software architecture style involved in GFTSA demonstrates how the component & connectors in the *FTS* cooperate and compete with each together. In the following, we illustrate the significant style of *Object, connector*, and *SharedResource*, which incorporates several widely used software architecture styles.

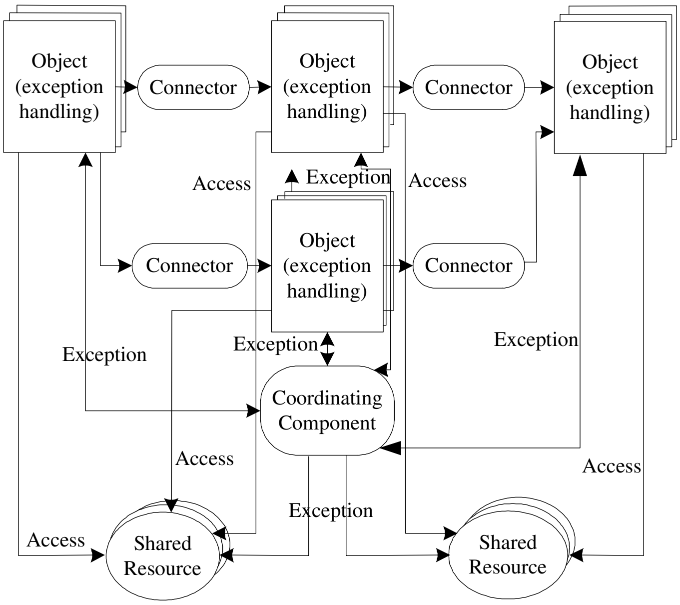

Figure 3.1: The generic fault tolerant software architecture.

3.2.1 Object

The *Object* involved in the *FTS* needs to implement independently task, and execute concurrently with other different *Objects*. In the object-oriented organization [28], data and their associated operations are encapsulated into an abstract *Object*. This object-oriented organization makes the *Object* hide the implementation details, which allows the *Objects* to be changed without affecting its others. Therefore, we design the style of *Object* similar to the object-oriented organization, which can accommodate the distributed environment. Derived from the object-oriented organization, *Object* can encapsulate data representations, and their associated primitive operations within a single component.

Accordingly, our proposed GFTSA can guide the development of distributed systems with cooperative concurrency, since the *Objects* can execute in parallel with other *Objects*. But the communication style of object-oriented organization is not so suitable for the distributed environment. For an *Object* to interact with other *Objects*, it must know the identity of other *Objects*.

3.2.2 Connector

Since the *Objects* need to execute concurrently in the distributed systems, we propose to design a communication pattern for the *Object* to accommodate the distributed environment. Referring to pipe-and-filter architecture [2], filters must be the independent entities, and they do not need to know the identity of upstream or downstream filters. They may specify input format and guarantee what appears on output, but they may not know which components appears at the ends of those pipes. Such pipe-and-filter style can support concurrent execution. Considering the cooperative concurrency occurred in the distributed systems, the *Objects* also do not need to know the identity of *Objects* which communicate with. Therefore, we design *Connectors* in our proposed architecture to help the interaction among *Objects*.

Similar to the pipe communication pattern in the pipe-and-filter architecture, the *Connectors* in GFTSA connect the *out_port* of one *Object* to the *in_port* of another *Object*. The cooperative concurrency is modelled by the *Objects* interacting with each other via the *Connectors* to cater for common goals.

3.2.3 SharedResource

In the distributed systems, the share resource, such as database recording the information, the entities occupied by several components, and etc, are widespread. Therefore, we need to consider how these *SharedResources* can be accessed by different *Objects* to preserve the consistent states. Referring to the repository style [3], there are two distinct components: a central data structure which represents the current state, and a collection of independent components which operate on the data-store. Derived from this style, we can design the *Objects* as the independent components, and the *SharedResources* as the central data structure.

Since the *SharedResource* can be accessed by different *Objects*, we need to apply a methodology to maintain the consist state of *SharedResource*, which implies that the *SharedResource* need to guarantee the transaction semantics [24, 46]. The transaction semantics indicates that at a given time, each *SharedResource* can only be accessed by one *Object*. That *Objects* compete for *SharedResource* models the competitive concurrency.

3.2.4 CoordinatingComponent

As GFTSA is proposed to guide the development of distributed systems with high reliability requirements. GFTSA must preserve the ability to deal with the exceptions occurring in the distributed environment. Different from the non-distributed systems, the exceptions occurred in the distributed can affect not only the components which raise such exceptions, but also the components which interact with these components. Therefore, we need to design an independent component, namely *CoordinatingComponent*, to help deal with these exceptions.

The *CoordinatingComponent* is designed to help resolve the multiple exceptions raised by different *Objects* in the distributed system. The *CoordinatingComponent* can communicate with *Objects* and *SharedResources* involved in the distributed system via transferring messages.

As shown in Figure 3.1, GFTSA provides a software architecture which involves three kinds of components, namely *Object, SharedResource, and CoordinatingComponent.* The *Object* component can execute primitive task independently, and interact with other *Objects* via *connectors.* The *SharedResource* component represents the resources which can be occupied by several *Objects.* The *CoordinatingComponent* in particular can help deal with the exceptions occurring in the distributed environment.

3.3 Fault Tolerant Techniques of GFTSA

If exceptions occur in the *FTS*, fault tolerant techniques need to deal with the exceptions to satisfy the reliability requirements. Our proposed GFTSA incorporates fault tolerant techniques in the early system design phase, which can be reused in the development of distributed systems with high reliability requirements. Since the exceptions in the distributed environment are different from the ones in the non-distributed environment, the consequence of which may step outside the boundaries of a computer system, the fault tolerant techniques involved in GFTSA need to concern such characteristics of the exceptions.

3.3.1 The idealized fault tolerant component

The concern of fault tolerant properties in the designing of distributed systems makes the development of such system more complicated. To ease such complexity, we adopt the concept of *idealized fault tolerant component*[5, 8] in the *Objects*. By incorporating such concept, the *Object* can include both normal and abnormal processes to the interacting components within one single component, which could minimize the impact on system complexity.

In the *Object*, the normal process is responsible for the execution of task, and the abnormal process is responsible for dealing with the exceptions. The **exception context** involved in the *Object* can be used in the abnormal process when facing exceptions. The *exception context* has a set of **exception handlers**[16, 62], one of which is called when its corresponding exception is raised. During the execution of an *Object*, a **checkpoint**[12, 39] is used to record the latest normal execution state of the *Object*. After calling the corresponding *exception handler* in the *exception context* to deal with exceptions, the *Object* can either go to a normal state, or roll back to the normal execution state recorded by the *checkpoint*. This solution is scalable as it only requires extending the behavior of existing objects rather than adding new objects to deal with exceptions.

3.3.2 The coordinated error recovery mechanism

Because of the interactive and concurrent characteristic of distributed systems, the exceptions occurring in one component of such systems can affect not only the component raises the exception, but also the other components interacting with this component. The Object using the *idealized fault tolerant component* technique cannot handle such situation. We incorporate *coordinated error recovery mechanism* in GFTSA to handle the exceptions which affect more than one component.

In order to distinguish the exceptions which affect the control flow of more than one *Object* within the distributed system, from the exceptions whose influence is limited within a single *Object*, we classify the exceptions raised in the *Object* into two types: **local exceptions**, and **global exceptions**. The influence of a local exception is limited within a single *Object*. Global exceptions, on the other hand, affect the control flows of more than one *Object* within a distributed system. Once a local exception is raised in one *Object*, the *Object* can call the corresponding exception handler in its own exception context to cope with the exception. If this exception cannot be handled successfully, a global exception is signalled, which can be transferred to the *CoordinatingComponent*. If a global exception is originally raised in an *Object*, this global exception is also passed to the *CoordinatingComponent*. The *CoordinatingComponent* broadcasts the global exception to the related *Objects* & *SharedResources* within the distributed system. These components need to replace the normal process with the abnormal process.

Different from non-distributed computing environment, we also need to consider how to deal with concurrently raised global exceptions in the distributed system. In the *coordinated error recovery mechanism*, when several global exceptions are raised in different *Objects* concurrently, these global exceptions are passed to the *CoordinatingComponent* concurrently. The *CoordinatingComponent* uses **exception graph**[85, 86] mechanism to resolve these concurrently raised exceptions into a unique global exception, namely **universal exception**, which covers all the raised exceptions. When the *CoordinatingComponent* obtains the *universal exception*, it propagates this exception to all the related *Objects* & *SharedResources* involved in the distributed system. Furthermore, the *Objects* call the corresponding *exception handlers* in their own *exception contexts* to deal with the exception. The state of each *SharedResource* needs to be restored to its prior normal state.

3.4 Summary

In this chapter, we proposed a novel heterogeneous software architecture, namely Generic Fault Tolerant Software Architecture (GFTSA), to guide the development of distributed systems with high reliability requirements. Several widely used software architecture styles are combined in GFTSA to provide a generic framework to the development of distributed systems involving cooperative & competitive concurrency. These architecture styles include object-oriented organization, pipe-and-filter architecture, and repository style. The styles of components and connectors involved in GFTSA are all derived from these architecture styles. This chapter presents the proposed GFTSA in a box-and-line fashion, accompanied with the literal illustration of basic features of components and connectors of GFTSA.

GFTSA incorporates fault tolerant techniques in the early system design phase to satisfy the reliability requirements of distributed systems. Considering the characteristics of exceptions occurred in the distributed environment, we mainly incorporate two kinds of fault tolerant techniques in GFTSA. The fault tolerant technique *idealized fault tolerant component* can make *Object* of GFTSA have the ability to deal with the *local exceptions* raised by itself or the resolved *universal exception* passed by *CoordinatingComponent.* The fault tolerant technique *coordinated error recovery mechanism* can help deal with a raised global exception or concurrently multiple raised global exceptions in the distributed system. These fault tolerant techniques can be reused in the development of distributed systems with high reliability requirements guided by GFTSA.

Chapter 4

Formal Modeling of GFTSA

This chapter presents the formal model of GFTSA in the Object-Z language.

4.1 Introduction

GFTSA is proposed to provide a generic framework to guide the development of distributed systems with reliability requirements. Good understanding and precise representation of software architecture can lead to reliable system implementation based on this architecture[22, 51]. The well-defined semantics & syntax make formal modeling techniques suitable for precisely specifying, and formally verifying architecture designs[45, 47, 69].

Z [77] is a formal language based on set theory and predicate logic, which can help describe internal state transitions, and interface communications of a system by the state and operation schema definitions. Many researchers [1, 71] have used Z to formalize the state and computations of software architectures. Object-Z [20, 74] is an extension of Z to accommodate the object-orientated style. Compared to formal language Z, Object-Z can improve the clarity of large specifications through enhanced structuring, and help the system designers to reuse the formal model of GFTSA via inheritance & instantiation mechanisms. Timed Communicated Object-Z (TCOZ) [48, 49, 50] is essentially a blending of Object-Z with Timed CSP [67]. The essence of this blending is the identification of Object-Z operation specification with terminating CSP processes. TCOZ also could be a good candidate for architecture description, which has been applied in the design and verification of a generic Computer Aided Dispatch (CAD) system [79, 80]. Compared to Object-Z, TCOZ is over expressed for our proposed architecture, but it could be useful if our architecture is further extended to involve time.

In this chapter, we formally model GFTSA by using the Object-Z language [88]. Following the semantics of Object-Z, the software architecture style and fault tolerant techniques involved in GFTSA can be specified precisely to provide explicit features to the system designers. Since GFTSA is proposed to guide the development of distributed systems with high reliability requirements, the crucial properties that GFTSA need to preserve are the fault tolerant properties. The fault tolerant properties indicate that when exceptions occur, GFTSA has the ability to deal with these exceptions and make the system recover to normal process. Based on the Object-Z model of GFTSA, we can formally reason about the fault tolerant properties of GFTSA by using the reasoning rules of Object-Z[72, 73].

The remainder of the chapter is organized as follows. Section 2 presents the formal model of GFTSA represented by the Object-Z language. Section 3 presents several significant fault tolerant properties of GFTSA, and demonstrates that GFTSA can preserve these properties by formal reasoning. Section 4 concludes the chapter.

4.2 Object-Z Model of GFTSA

The formal model of GFTSA can provide precise and explicit patterns & idioms to the system designers by formally specifying the architecture style and fault tolerant techniques involved in GFTSA. The components and connectors of GFTSA, shown in Figure 3.1, are all represented as class schemas, which group the state and operation schemas. The formal model of GFTSA is composed of *Global Types*, *Object*, *Connector*, *CoordinatingComponent*, *ShareResource*, and *FTSystem* class schemas, according to the structure of GFTSA.

4.2.1 Global Types

The global types declared below can provide notations to the *Object, Connector, CoordinatingComponent, SharedResource*, and *FTSystem* class schemas, which can be used to associate type to the constants and variables declared in these class schemas. The comments in the bracket can indicate the meaning of each type.

$[PORT]$ [port names used by *Object* to communicate]
$[MSG]$ [set of messages to be transmitted]
$[OBSTATE]$ [set of states that *Object* can be in]
$[SRSTATE]$ [set of states that *SharedResource* can be in]
$[EH]$ [set of exception handlers to deal with the exceptions]

$RESULT ::= tolerate \mid stop$ [the result]
$SIG == \{0, 1\}$ [the signal]

$NORMAL : \mathbb{P}\, OBSTATE$
[set of normal states that *Object* can be in]
$LE : \mathbb{P}\, OBSTATE$ [set of local exceptions that *Object* can raise]
$GE : \mathbb{P}\, OBSTATE$ [set of global exceptions that *Object* can raise]
$Fail : OBSTATE$ [fail state that *Object* can be in]

$NORMAL \cap LE = \varnothing \land NORMAL \cap GE = \varnothing \land LE \cap GE = \varnothing$
$Fail \notin NORMAL \cup LE \cup GE$
$OBSTATE = NORMAL \cup LE \cup GE \cup \{Fail\}$

The *NORMAL, LE, GE*, and *Fail* declared in the axiomatic definition above are four different states that the *Object* can be in. The predicate part of the axiomatic definition specifies that the state of *Object* can be in such four states, and only be in such four states.

4.2.2 Fault Tolerant Component - Object

The *Object* class schema describes the features of *Object* in GFTSA, involving not only the normal execution process but also the fault tolerant process of *Object*. Since the behavior that the *Object* receives and sends messages is similar to the pattern of *Queue* system, illustrated in Section 2.1, the *Object* class schema can

$$
\begin{array}{l}
\underline{\mathit{Object}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
\upharpoonright(\mathit{inter\_state}, \textsc{Init}, \mathit{GlobalExceptPropagate}, \\
\qquad \mathit{UniExceptReceive}, \mathit{UniExceptHandle}, \mathit{SRRequest}, \mathit{FromSR}, \mathit{ToSR}) \\
\mathit{Queue}[\downarrow \mathit{SharedResource}][\mathit{sr\_qlist}/\mathit{items}, \mathit{ans\_sr}?/\mathit{item}?, \mathit{to\_sr}!/\mathit{item}!, \\
\qquad\qquad\qquad\qquad \mathit{FromSR}/\mathit{Join}, \mathit{ToSR}/\mathit{Leave}]
\end{array}
$$

$$
\begin{array}{|l}
\mathit{n\_states} : \mathbb{P}\, \mathit{NORMAL} \\
\mathit{l\_excepts} : \mathbb{P}\, \mathit{LE} \\
\mathit{g\_excepts} : \mathbb{P}\, \mathit{GE} \\
\mathit{in\_ports}, \mathit{out\_ports} : \mathbb{P}\, \mathit{PORT} \\
\mathit{comp\_msgs} : \mathbb{P}\, \mathit{MSG} \\
\mathit{coop\_msg} : \mathit{PORT} \rightarrow \mathit{MSG} \\
\mathit{transition} : \mathit{NORMAL} \times (\mathit{PORT} \times \mathit{MSG}) \\
\qquad\qquad \rightarrow \mathit{OBSTATE} \times (\mathit{PORT} \times \mathit{MSG}) \\
\mathit{except\_context} : \mathit{LE} \cup \mathit{GE} \nrightarrow \mathit{EH} \\
\mathit{except\_handle} : \mathit{EH} \rightarrow \mathit{OBSTATE} \\
\hline
\mathit{in\_ports} \cap \mathit{out\_ports} = \varnothing \wedge \\
\mathit{n\_states} \cap \mathit{l\_excepts} \cap \mathit{g\_excepts} = \varnothing \\
\mathit{comp\_msgs} \cap \operatorname{ran} \mathit{coop\_msg} = \varnothing \\
\operatorname{dom}(\operatorname{dom} \mathit{transition}) \subseteq \mathit{n\_states} \\
\operatorname{dom}(\operatorname{ran}(\operatorname{dom} \mathit{transition})) \subseteq \mathit{in\_ports} \wedge \\
\operatorname{dom}(\operatorname{ran}(\operatorname{ran} \mathit{transition})) \subseteq \mathit{out\_ports} \\
\operatorname{ran}(\operatorname{ran}(\operatorname{dom} \mathit{transition})) \subseteq \operatorname{ran} \mathit{coop\_msg} \wedge \\
\operatorname{ran}(\operatorname{ran}(\operatorname{ran} \mathit{transition})) \subseteq \operatorname{ran} \mathit{coop\_msg} \\
\operatorname{dom} \mathit{except\_context} \subseteq \mathit{l\_excepts} \cup \mathit{g\_excepts}
\end{array}
$$

$$
\begin{array}{|l}
\mathit{inter\_state} : \mathit{OBSTATE} \\
\mathit{checkpoint} : \mathit{NORMAL} \\
\mathit{ue\_rec} : \mathit{SIG} \\
\hline
\mathit{inter\_state} \in \mathit{n\_states} \cup \mathit{l\_excepts} \\
\qquad \cup \mathit{g\_excepts} \cup \{\mathit{Fail}\} \\
\mathit{checkpoint} \in \mathit{n\_states} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\underline{\textsc{Init}\qquad\qquad\qquad\qquad} \\
\mathit{inter\_state} \in \mathit{n\_states} \\
\mathit{checkpoint} = \mathit{inter\_state} \\
\mathit{ue\_rec} = 0 \\
\hline
\end{array}
$$

Transition

$\Delta(inter\_state, checkpoint)$

$inter\_state \in n\_states$
$checkpoint' = inter\_state$
$\exists\, p_1, p_2 : PORT;\ m_1, m_2 : MSG \mid$
$inter\_state, (p_1, m_1)) \in \operatorname{dom} transition$
$\qquad \bullet\ (inter\_state', (p_2, m_2)) = transition(inter\_state, (p_1, m_1))$

LocalExceptHandle

$\Delta(inter\_state)$

$inter\_state \in l\_excepts$
$except\_handle(except\_context(inter\_state)) \in n\_states \Rightarrow$
$inter\_state' = except\_handle(except\_context(inter\_state))$
$except\_handle(except\_context(inter\_state)) = Fail \Rightarrow$
$inter\_state' \in g\_excepts$

GlobalExceptPropagate

$exception! : GE$

$inter\_state \in g\_excepts$
$ue\_rec = 0$
$exception! = inter\_state$

UniExceptReceive

$\Delta(inter\_state, ue\_rec)$
$uni\_exception? : GE$

$inter\_state' = uni\_exception?$
$ue\_rec' = 1$

UniExceptHandle

$\Delta(inter\_state, ue\_rec, sr\_qlist)$

$inter\_state \in g\_excepts \wedge ue\_rec = 1$
$except\_handle(except\_context(inter\_state)) \in n\_states \Rightarrow$
$inter\_state' = except\_handle(except\_context(inter\_state))$
$except\_handle(except\_context(inter\_state)) = Fail \Rightarrow$
$inter\_state' = Fail$
$ue\_rec' = 0 \wedge sr\_qlist' = \langle\ \rangle$

SRRequest

$req\_ob! :\downarrow Object$
$req\_sr! :\downarrow SharedResource$
$sr? :\downarrow SharedResource$

$inter\_state \in n\_states$
$comp\_msg \neq \{\ \}$
$req\_ob! = self \wedge req\_sr! = sr?$

FromSR

$ans\_ob? :\downarrow Object$

$inter\_state \in n\_states$
$ans\_ob? = self$

ToSR

$msg! : MSG$

$inter\_state \in n\_states$
$msg! \in comp\_msgs$

inherit the *Queue[Item]* class schema by using instantiation and rename mechanisms of Object-Z. The generic type *Item* of *Queue[Item]* is instantiated with class schema type *SharedResource*. The *items*, *item*?, and *item*! are all be renamed according to the specific requirements.

Firstly, we give a brief illustration to the constants declared in the local axiomatic definition of *Object* class schema. The declared constants *n_states*, **l_excepts**, and **g_excepts** represent three different sets of states that an *Object* can be in: a set of normal states, a set of local exception states, and a set of global exception states. To model the idea that the IO-ports are directional, we partition ports into a set of *in_ports*, and a set of *out_ports*. The declared constant *comp_msgs* represents a set of messages that an *Object* can transmit to the *SharedResources*. We associate a message with a port in the *coop_msg*, which indicates that the message can be received or sent out from the associated port. The *transition* function specifies that when an *Object* receives a message at its *in_port*, the state of the *Object* can be changed while sending out a message from its *out_port* at the same time. The **except_context** function models that any exception occurred in the *Object* has a corresponding exception handler. The function **exception_handle** is used to check whether the exception handler deals with the exception successfully. The predicate part of the axiomatic definition imposes several constraints on the declared constants.

The state schema in the *Object* class schema declares four variables: *inter_state*, *checkpoint*, *ue_rec*, and *sr_qlist*. The state of *Object* can be changed by changing these variables. The *inter_state* denotes the current state of the *Object*, the **checkpoint** records the normal execution state of *Object*, the *ue_rec* indicates whether the *Object* has received a **universal exception** from the *CoordinatingComponent*, and the *sr_qlist* records the identity of *SharedResources*, which are available for the *Object*.

The *Transition* operation schema denotes the state transitions of the *Object* according to the *transition* function. The **LocalExceptHandle** operation specifies how the *Object* deals with local exception. The operation **GlobalExceptPropagate**, **UniExceptReceive**, and **UniExceptHandle** denote how the *Object* implements the *coordinated error recovery mechanism*.

The communication protocol among *Objects* and *SharedResource* need to guarantee the **transaction semantics**. When an *Object* wants to access a *SharedResource*, it needs to send an access request to this *SharedResouce*, which is specified in the operation schema *SRRequest*. The *FromSR* operation describes that the *Object* receives an answer from an available *SharedResource*. The *ToSR* operation denotes that the *Object* sends out the message, called *msg!*, to the assured available *SharedResource*.

4.2.3 Connector

The *Connector* class schema describes that a *connector* of GFTSA is responsible for connecting the *send_port* of an *Object* to the *receive_port* of another *Object* to transfer the message represented by *msg*. No operation occurs on the *connectors*. In the predicate part of axiomatic definition, the declaration $send\_ob, receive\_ob :\downarrow Object$ means that *send_ob*, and *receive_ob* refer to the *Object* class schema, or any of its subclass schemas which are in the inheritance hierarchy rooted at the *Object* class.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\; Connector \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\begin{array}{|l}
\hline
send\_port, receive\_port : PORT \\
\hline
\exists\, send\_ob, receive\_ob :\downarrow \\
\quad FTComponent;\ msg : MSG \mid \\
\quad send\_ob \neq receive\_ob \land \\
\qquad send\_port \in send\_ob.out\_ports \land \\
\qquad\quad receive\_port \in receive\_ob.in\_ports \\
\qquad\qquad \bullet\ msg = send\_ob.coop\_msg(send\_port) \land \\
\qquad\qquad\quad receive\_ob.coop\_msg(receive\_port) = msg \\
\hline
\end{array} \\
\hline
\end{array}
$$

4.2.4 CoordinatingComponent

The *CoordinatingComponent* class schema describes how the *CoordinatingComponent* of GFTSA implements the *coordinated error recovery mechanism* when a global exception is raised, or multiple global exceptions are raised concurrently. Since the behavior of the *CoordinatingComponent* that it receives and sends *exceptions* is similar to the pattern of *Queue* system, the *CoordinatingComponent* class schema inherits the *Queue[Item]* class schema by instantiating *Item* with the type *GE*, and renaming *items, item?*, and *item!*.

CoordinatingComponent

$\upharpoonright(\textsc{Init}, \mathit{ExceptRec}, \mathit{ExceptGraph})$

$\mathit{Queue}[\mathit{GE}][\mathit{exceptions}/\mathit{items}, \mathit{exception}?/\mathit{item}?, \mathit{ExceptRec}/\mathit{Join}]$

$\mathit{except\_graph} : \mathrm{seq}_1\ \mathit{GE} \to \mathit{GE}$

ExceptGraph

$\Delta(\mathit{exceptions})$

$\mathit{uni\_exception}! : \mathit{GE}$

$\mathit{exceptions} \neq \langle\ \rangle$

$\mathit{uni\_exception}! = \mathit{except\_graph}(\mathit{exceptions})$

$\mathit{exceptions}' = \langle\ \rangle$

The constant declared in the axiomatic definition, **except_graph**, is a function to resolve multiple concurrently raised exceptions into a **universal exception**, namely *uni_exception*, which can cover all the raised exceptions. The variable *exceptions* represents the sequence of received exceptions from *Objects*. The operations *ExceptRec*, and *ExceptGraph* are responsible for receiving exception? from *Objects*, resolving these received exceptions by the *except_graph*, and sending out the resolved exception *uni_exception!* to the *Object* and *SharedResource*.

4.2.5 SharedResource

The *SharedResource* class schema models how the *SharedResource* of GFTSA can guarantee the *transaction semantics* when receiving messages from *Objects*, and preserve consistent state when facing exceptions. Since the behavior that the *SharedResource* receives and sends messages is similar to the pattern of *Queue* system, the *SharedResource* class schema also inherits the *Queue[Item]* schema by using the instantiation & rename mechanisms of Object-Z.

In the axiomatic definition, the declared constant *states* represents a set of states that the *SharedResource* can be in, and function *trans* is used to model the state transition of the *SharedResource* when it receives a message from an *Object*. The state variables **semaphore**, *ob_qlist*, *sr_state*, and **checkpoint** represent the signal to show whether the *SharedResource* is accessed by an *Object*, the request list of *Objects*, the current state of *SharedResource*, and the recorded prior nor-

SharedResource

$\upharpoonright(\textsc{Init}, ObList, Available, Trans, Except)$

$Queue[\downarrow Object][ob\_qlist/items, req\_ob?/item?,$

$ans\_ob!/item!, ObList/Join, Available/Leave]$

$states : \mathbb{P}\, SRSTATE$

$trans : SRSTATE \times MSG \rightarrow SRSTATE$

$\mathrm{dom}(\mathrm{dom}\, trans) = \mathrm{ran}\, trans \subseteq states$

$semaphore : SIG$

$sr\_state : SRSTATE$

$checkpoint : SRSTATE$

$sr\_state \in states \land checkpoint \in states$

$\textsc{Init}$

$semaphore = 0$

$checkpoint = sr\_state$

ObList

$req\_sr? : \downarrow SharedResource$

$req\_sr? = self$

Available

$\Delta(semaphore)$

$ans\_sr! : \downarrow SharedResource$

$semaphore = 0 \land ans\_sr! = self$

$semaphore' = 1$

Trans

$\Delta(semaphore, sr\_state, checkpoint)$

$to\_sr? : \downarrow SharedResource$

$msg? : MSG$

$semaphore = 1 \land to\_sr? = self$

$sr\_state' = trans(sr\_state, msg?)$

$checkpoint' = sr\_state'$

$semaphore' = 0$

Except

$\Delta(semaphore, ob\_qlist, sr\_state)$

$uni\_exception? : GE$

$semaphore' = 0$

$ob\_qlist' = \langle\ \rangle$

$sr\_state' = checkpoint$

mal state of *SharedResource* respectively. The *ObList* operation specifies that the *SharedResource* receives an access request from an *Object*. The *Available* operation models that the *SharedResource* sends out a signal to the *Object*, when the *SharedResource* is available. The *Trans* operation specifies the state transitions of *SharedResource* according to the *trans* function when receiving *msg?* from an *Object*. The *Except* operation describes that the state of *SharedResource* needs to roll back to the normal state recorded in the *checkpoint* when facing a *uni_exception?*.

4.2.6 Fault Tolerant System - FTSystem

The *FTSystem* class schema describes how the components & connectors in GFTSA, which constitute a *FTS* (Fault Tolerant System), are synchronized. In the local axiomatic definition, the declared constant *critical* represents the set of *Objects* whose *Fail* state can cause the whole *FTS* to stop, and **Result_Control** is a function to check the execution result of *FTS*. If the state of any critical *Object* is not in the *Fail* state, the execution result of *FTS* is *tolerate*, which means that the *FTS* can recover from the exceptions; otherwise the whole *FTS* has to stop execution.

The instances of components & connectors in the *FTS* are all declared in the state schema. The secondary variable *ob_fail* records a set of *Objects* in the *Fail* state. The *SystemRecover* operation models that the states of all *Objects* in the *FTS* should be initialized, when the execution result of *FTS* is *tolerate*. The *Transition* operation expression is the conjunction of *Transition* operations of all *Objects* in

FTSystem

$$\begin{array}{l}
critical : \mathbb{P} \downarrow Object \\
Result\_Control : \mathbb{P} \downarrow Object \rightarrow RESULT \\
\hline
\forall fobs : \mathbb{P} \downarrow Object \bullet \exists fob : fobs \bullet \\
\quad fob \in critical \Rightarrow Result\_Control(fobs) = stop \\
\forall fobs : \mathbb{P} \downarrow Object \bullet \forall fob : fobs \bullet \\
\quad fob \notin critical \Rightarrow Result\_Control(fobs) = tolerate
\end{array}$$

$$\begin{array}{l}
obs : \mathbb{P} \downarrow Object \\
cs : \mathbb{P} \downarrow connector \\
coco : \downarrow CoordinatingComponent \\
srs : \mathbb{P} \downarrow SharedResource \\
\Delta \\
ob\_fail : \mathbb{P} \downarrow Object \\
\hline
\forall ob_1, ob_2 : obs \bullet ob_1 \neq ob_2 \\
\forall ob : obs;\ pt : PORT \mid pt \in ob.out\_ports \bullet \\
\quad \exists_1\, c : cs \bullet pt = c.send\_port \\
\forall ob : obs;\ pt : PORT \mid pt \in ob.in\_ports \bullet \\
\quad \exists_1\, c : cs \bullet pt = c.receive\_port \\
\forall ob : obs \mid ob.inter\_state = Fail \bullet \\
\quad ob \in ob\_fail
\end{array}$$

INIT

$$\begin{array}{l}
\forall ob : obs \bullet ob.\mathrm{INIT} \\
coco.\mathrm{INIT} \\
\forall sr : srs \bullet sr.\mathrm{INIT}
\end{array}$$

SystemRecover

$$\begin{array}{l}
Result\_Control(ob\_fail) = tolerate \\
\forall ob : obs \bullet ob.\mathrm{INIT}
\end{array}$$

$$\begin{array}{l}
Transition \mathrel{\widehat=} \bigwedge ob : obs \bullet ob.Transition \\
ExceptPropagate \mathrel{\widehat=} \bigwedge ob : obs \bullet \\
ob.GlobalExceptPropagate \parallel coco.ExceptRec \\
ExceptGraph \mathrel{\widehat=} coco.ExceptGraph \parallel \\
(\bigwedge ob : obs \bullet ob.UEReceive \wedge \bigwedge sr : srs \bullet sr.Except) \\
ObReqSR \mathrel{\widehat=} \bigwedge ob : obs \bullet ob.SRReq \parallel \bigwedge (sr : srs \bullet sr.ObList) \\
SRAnsOb \mathrel{\widehat=} \bigwedge sr : srs \bullet sr.Available \parallel (\bigwedge ob : obs \bullet ob.FromSR) \\
ObAccSR \mathrel{\widehat=} \bigwedge ob : obs \bullet ob.ToSR \parallel (\bigwedge sr : srs \bullet sr.Trans)
\end{array}$$

the *FTS*. The operation *ExceptPropagate* specifies that, when global exceptions are raised in the *Objects*, these *Objects* need to pass the exceptions to the *CoordinatingComponent* by the parallel operator ∥ to compose two operations together. The *ExceptGraph* operation specifies that the *CoordinatingComponent* sends the resolved exception *uni_exception!* to all the *Objects* & *SharedResources*. The operations *ObReqSR*, *SRAnsOb*, and *ObAccSR* are used to model how the *Objects* compete for the *SharedResources*.

4.3 Reasoning about GFTSA

The formal model of GFTSA specifies the software architecture style and fault tolerant techniques involved in GFTSA in a precise and compact way by using the Object-Z language. Since GFTSA is used to help design distributed systems with high reliability requirements, GFTSA needs to preserve fault tolerant properties to satisfy such requirements. In this section, we reason about [65] GFTSA to demonstrate that GFTSA can preserve fault tolerant properties. The process of reasoning needs to use reasoning rules of Object-Z[72, 73] to prove that fault tolerant properties can be derived from the Object-Z model of GFTSA. The following items show that GFTSA can preserve significant fault tolerant properties, which are expressed as theorems.

1. When a *global exception* is raised by an *Object* in the *FTS*, all of the *Objects*, and *SharedResources* in the *FTS* can deal with this exception. This property can be formally expressed as follows.

Theorem

$$\begin{aligned} &FTS :: \exists\, ob : obs \mid ob.inter\_state \in ob.g\_excepts \vdash \\ &\quad \forall\, ob : obs;\ sr : srs \bullet ob.ue\_rec' = 1 \land \\ &\qquad sr.sr\_state' = sr.checkpoint \end{aligned}$$

As an intermediate step, it is useful to think of the proof strategy informally. When a global exception is raised in an *Object*, the *Object* can use *GlobalExceptPropagate* operation to send this global exception out to the *CoordinatingComponent*, and the *CoordinatingComponent* can use the *ExceptRec* operation to receive this global exception. Two operations are combined in the *ExceptPropagate* operation expression declared in the *FTSystem* class schema. Because the sequence *exceptions* is not empty, the *ExceptGraph* operation in the *CoordinatingComponent* class schema sends out the *uni_exception!*. The *UEReceive* operation in the *Object*, and the operation *Except* in the *SharedResource* can receive the *uni_exception?*, which is expressed in the *ExceptGraph* operation declared in the *FTSystem* class schema. When an *Object* receives the *uni_exception?*, the value of *ue_rec* is changed to 1. When a *SharedResource* receives the *uni_exception?*, its *sr_state* rolls back to the normal state recorded in the *checkpoint*. These transformations assure that the *Objects*, and *SharedResources* are informed about the exception. In the following, a formal proof based on this strategy is constructed.

Proof

$$
\begin{array}{l}
FTSystem :: \exists\, ob : obs \mid ob.inter\_state \\
\in ob.g\_excepts \vdash ob.GlobalExceptPropagate \\
FTSystem :: ob : obs \mid obs \in \mathbb{P} \downarrow Object \vdash \\
\qquad ob \in\downarrow Object \\
\hline
Object :: GlobalExceptionPropagate \vdash \\
\qquad exception! = inter\_state \\
FTSystem \vdash ExceptPropagate \\
\hline
FTSystem \vdash coco.ExceptRec \\
FTSystem \vdash coco \in\downarrow CoordinatingComponent \\
\hline
CoordinatingComponent \vdash exceptions' = \\
\qquad exceptions \frown \langle exception? \rangle \\
CoordinatingComponent :: ExceptGrahp \mid \\
exceptions \neq \langle\ \rangle \vdash \\
\qquad uni\_exception! = except\_graph(exceptions) \\
FTSystem \vdash ExceptGraph \\
\hline
FTSystem :: ob : obs \vdash ob.UEReceive \\
FTSystem :: sr : srs \vdash sr.Except \\
Object :: UEReceive \vdash ue\_rec' = 1 \\
SharedResource :: Except \vdash \\
\qquad sr\_state' = checkpoint \\
\hline
FTSystem :: ob : obs;\ sr : srs \vdash ob.ue\_rec' = 1 \\
\qquad \wedge\ sr.sr\_state' = sr.checkpoint
\end{array}
$$

2. If an *Object* in the *FTS* raises a *local exception*, the other *Objects* except this one in the *FTS* are not influenced, and still in their normal states. This property can be formally expressed as follows.

Theorem

$$
\begin{array}{l}
FTS :: \exists\, ob : obs \mid ob.inter\_state \in ob.l\_excepts \vdash \\
\qquad \forall\, other\_ob : obs \mid other\_ob \neq ob \bullet \\
\qquad\qquad other\_ob.inter\_state \in other\_ob.n\_state
\end{array}
$$

The proof strategy is described first. When a *local exception* is raised by an *Object*, the *LocalExceptHandle* operation in the *Object* class is executed to handle this exception. If this local exception is handled successfully, the state of *Object* recovers to a normal state. Otherwise, the state of *Object* is changed to a global exception state. There is no communication between this *Object* and other *Objects*. The other *Objects* except this one are still in their normal states, which cannot be influenced by the raised *local exception*. In the following, a formal proof based on this strategy is constructed.

Proof

$$\begin{array}{l}
FTSystem :: \exists\, ob : obs \mid ob.inter\_state \in \\
\quad ob.l\_excepts \vdash ob.\mathit{LocalExceptionHandle} \\
FTSystem :: \exists\, ob : obs \mid obs \in \mathbb{P} \\
\quad \downarrow \mathit{Object} \vdash ob \in\downarrow \mathit{Object} \\
\mathit{Object} :: \mathit{ExceptionHandle} \vdash \\
\quad \mathit{inter\_state}' \in \mathit{n\_states} \\
\qquad \vee\ \mathit{inter\_state}' \in \mathit{g\_excepts} \\
\hline
FTSystem \vdash ob.inter\_state' \in ob.n\_states\ \vee \\
\quad ob.inter\_state' \in ob.g\_excepts \\
FTSystem :: other\_ob : obs \mid other\_ob \neq ob \vdash \\
\quad other\_ob.inter\_state \neq ob.inter\_state' \\
\hline
FTSystem :: \forall\, other\_ob : obs \mid other\_ob \neq ob \\
\quad \vdash other\_ob.inter\_state \in other\_ob.n\_state
\end{array}$$

3. When two *global exceptions* are raised concurrently by two different *Objects* in the *FTS*, all the *Objects* in the *FTS* need to be informed about the *universal global exception*. This property can be formally expressed as follows.

Theorem

$$\begin{array}{l}
FTSystem :: ob_1, ob_2 : obs \mid \\
\quad ob_1.inter\_state = ob_1.g\_excepts \\
\qquad \wedge\ ob_2.inter\_state = ob_2.g\_excepts \wedge ob_1 \neq ob_2 \vdash \\
\qquad\quad \forall\, ob : obs \bullet ob.ue\_rec' = 1
\end{array}$$

When two *global exceptions* are raised concurrently in the *FTS*, each *Object* can use *GlobalExceptPropagate* operation to send its *global exception* out to the *CoordinatingComponent*, and the *CoordinatingComponent* can use the *ExceptRec* operation to receive these two *global exceptions*. The *GlobalExceptPropagate*, and *ExceptRec* operations are combined in the *ExceptPropagate* operation expression declared in the *FTSystem* class schema. Because the sequence *exceptions* is not empty, the *ExceptGraph* operation can use the function *except_graph* to get the *uni_exception*, which covers the two *global exceptions*. After that, the *uni_exception!* is sent out to all the *Objects* in the *FTS*. The *UEReceive* operation in the *Object* class can receive this *uni_exception?*. The operations *ExceptGraph*, and *UEReceive* are combined in the *ExceptGraph* operation expression declared in the *FTSystem* class schema. When an *Object* receives the *uni_exception?*, the value of *ue_rec* is changed to 1, which means that the *Object* is informed about the exceptions. In the following, a formal proof based on this strategy is constructed.

Proof

$$\begin{array}{l}
FTSystem :: ob_1 : obs \mid ob_1.inter\_state = \\
\quad ob_1.g\_excepts \vdash ob_1.GlobalExceptPropagate \\
FTSystem :: ob_1 : obs \mid obs \in \mathbb{P} \downarrow Object \\
\quad \vdash ob_1 \in\downarrow Object \\
FTSystem :: ob_2 : obs \mid ob_2.inter\_state = \\
ob_2.g\_excepts \vdash ob_2.GlobalExceptPropagate
\end{array}$$

$$
\begin{array}{l}
FTSystem :: ob_2 : obs \mid obs \in \mathbb{P} \downarrow Object \vdash \\
\quad ob_2 \in \downarrow Object \\
Object :: GlobalExceptionPropagate \vdash \\
\quad exception! = inter\_state \\
\hline
FTSystem :: ob_1 : obs \mid ob_1.inter\_state = \\
\quad ob_1.g\_excepts \vdash exception! = ob_1.inter\_state \\
FTSystem :: ob_2 : obs \mid ob_2.inter\_state = \\
\quad ob_2.g\_excepts \vdash exception! = ob_2.inter\_state \\
FTSystem \vdash ExceptPropagate \\
\hline
FTSystem :: ob_1 : obs \mid ob_1.inter\_state = \\
\quad ob_1.g\_excepts \vdash exception! = ob_1.inter\_state \\
FTSystem :: ob_2 : obs \mid ob_2.inter\_state = \\
\quad ob_2.g\_excepts \vdash exception! = ob_2.inter\_state \\
CoordinatingComponent \vdash ExceptRec \\
\hline
CoordinatingComponent :: ExceptRec \vdash \\
\quad exceptions' = exceptions \frown \langle \\
\qquad ob_1.inter\_state, ob_2.inter\_state \rangle \\
CoordinatingComponent :: exceptions' \neq \langle \, \rangle \vdash \\
\qquad ExceptGraph \\
\hline
CoordinatingComponent :: ExceptGraph \\
\quad \vdash uni\_exception! = except\_graph(exceptions') \\
FTSystem \vdash ExceptGraph \\
\hline
Object \vdash UEReceive \\
FTSystem :: ob : obs \mid obs \in \mathbb{P} \downarrow Object \vdash \\
\quad ob \in \downarrow Object \\
Object :: UEReceive \vdash ue\_rec' = 1 \\
\hline
FTSystem :: ob : obs \vdash ob.ue\_rec' = 1
\end{array}
$$

4. When a non-critical *Object* fails, the *FTS* can tolerate this fault, which means that the states of all *Objects* in the *FTS* can recover to their normal states. This property can be formally expressed as follows.

Theorem

$$
\begin{array}{l}
FTSystem :: \exists \, ob : obs \mid ob.inter\_state = Fail \land \\
\quad ob \notin critical \vdash \\
\qquad \forall \, ob : obs \bullet ob.inter\_state' \in ob.n\_states
\end{array}
$$

First, we give the informal proof description. When the state of an *Object* is

Fail, if this *Object* is not in the state *critical* declared in the *FTSystem* class schema, we can gain the execution result, namely *tolerate*, by using function *Result_Control*. Because the execution result is *tolerate*, the *SystemRecover* operation is used to reset the states of all *Objects* to the initial states, which means that all the *Objects* recover to normal states. A formal proof based on the previous description is shown in the following.

Proof

$$
\begin{array}{l}
\mathit{FTSystem} :: \exists\, ob : obs \mid ob.inter\_state = \mathit{Fail} \land \\
\quad ob \notin critical \vdash ob\_fail = \{ob\} \land \\
\qquad ob \notin critical \\
\mathit{FTSystem} :: ob\_fail = \{ob\} \land \\
\quad ob \notin critical \vdash \\
\qquad \mathit{Result\_Control}(ob\_fail) = tolerate \\
\hline
\mathit{FTSystem} \vdash \mathit{Result\_Control}(ob\_fail) = tolerate \\
\mathit{FTSystem} :: \mathit{Result\_Control}(ob\_fail) = tolerate \\
\quad \vdash \mathit{SystemRecover} \\
\hline
\mathit{FTSystem} \vdash \mathit{SystemRecover} \\
\mathit{FTSystem} :: \mathit{SystemRecover} \vdash \forall\, ob : obs \bullet ob.\textsc{Init} \\
\hline
\mathit{FTSystem} :: ob : obs \vdash ob.\textsc{Init} \\
\mathit{FTSystem} :: ob : obs;\ obs : \mathbb{P} \downarrow \mathit{Object} \vdash ob \in\downarrow \mathit{Object} \\
\mathit{Object} :: \textsc{Init} \vdash inter\_state \in n\_states \\
\hline
\mathit{FTSystem} :: ob : obs \vdash ob.inter\_state' \in ob.n\_states
\end{array}
$$

4.4 Conclusion

In this chapter, we formally model the proposed GFTSA, a novel heterogenous software architecture, by using the Object-Z language. The Object-Z model of GFTSA can provide explicit & common idioms to the system designers. Since Object-Z is the *class* construct which groups the definition of a state schema and the definitions of its associated operation schemas, we represent the components and connectors in GFTSA as corresponding *class* schemas. In each *class* schema, we formally model the static and dynamic features of component or connector.

As GFTSA is proposed to guide the development of distributed systems with high reliability requirements, we incorporate the *idealized fault tolerant component* and *coordinated error recovery mechanism* in the architecture. To model the *idealized fault tolerant component*, the *Object* class schema declares abnormal states & exception handler functions in the axiomatic definition, and exception handler operations, which are all used to indicate how *Object* deals with exceptions. The *CoordinatingComponent* class schema models the *coordinated error recovery mechanism*. The *SharedResource* class schema also specifies the exception handler operation and how to guarantee the transaction semantics.

Based on the Object-Z model of GFTSA, we formally reason about several significant fault tolerant properties to verify that GFTSA can satisfy the reliability requirements. The formal reasoning process involves that we can derive the fault tolerant properties, expressed as theorems, from the Object-Z model of GFTSA.

Chapter 5

Customization of GFTSA

This chapter investigates to build a template based on the Object-Z model of GFTSA. By customizing this built template, we can auto-generate the Object-Z models of developed distributed systems guided by GFTSA.

5.1 Introduction

The Object-Z model of GFTSA can provide precise and common idioms & patterns to the system designers. How the Object-Z model of GFTSA can be customized in the development of distributed systems is the next issue we need to tackle. Since the inheritance & instantiation mechanisms of Object-Z can help the customization, each class schema in the Object-Z model of GFTSA can be inherited by the corresponding class schema of developed systems.

Besides inheriting corresponding class schema in the Object-Z model of GFTSA, each class schema of developed systems also needs to be specified according to system requirements. The specifications could involve defining visible list, declaring new constants and predicates in the axiomatic definition, defining the initial state schema or new operation schemas, and etc, which could not be supported by the inheritance & instantiation mechanisms of Object-Z. In order to make the customization process more efficient, we investigate to build a template based on the class schemas in the Object-Z model of GFTSA. The class schemas in the template involve not only inheriting and instantiating corresponding class schemas in the Object-Z model of GFTSA, but also other specifications which cannot be supported by the inheritance & instantiation mechanisms of Object-Z. By customizing the class schemas in the built template, we can generate the Object-Z models of developed systems. The customization process could be small or large change to the class schemas in the template. Since the granularity of customization is flexible, in order to make the customization process automatic, we investigate to apply XML-based Variant Configuration Language (XVCL) technique to the customization process.

XML-based Variant Configuration Language (XVCL)[36, 75, 35, 89] is a meta-programming technique developed to facilitate building flexible, adaptable, and reusable software artifacts. When developing an XVCL solution, we partition a problem description(e.g. a software specification, or a software program) into generic, adaptable meta-components called x-frames. Following the mechanisms of XVCL, we build our proposed template as primitive x-frames. The Textual Content of x-frames is written as the combination of Object-Z specification and XVCL commands. XVCL commands mark the anticipated variation points in x-frames, injecting flexibility into their Textual Contents. When developing a distributed system with high reliability requirements, we firstly compose x-frames for the developed system by adapting the x-frames in the template. Based on the composed x-frames, we can auto-generate the Object-Z model of the developed system by running the XVCL processor. A case study of Sales Control System (SCS) [87] is presented to illustrate the customization process. Following the reasoning rules of Object-Z, we can formally verify that the generated Object-Z model of SCS can preserve the fault tolerant properties.

The remainder of the chapter is organized as follows. Section 2 presents a template for customization, which is built based on the Object-Z model of GFTSA by using the XVCL technique. Section 3 presents a case study of SCS to illustrate how to generate the Object-Z model of SCS automatically from the built template, and formally reason about the fault tolerant properties of SCS based on the generated model by following the reasoning rules of Object-Z. Section 4 concludes the chapter.

5.2 Template based on Object-Z model of GFTSA

GFTSA is proposed to guide the development of distributed systems with high reliability requirements. By using the inheritance & instantiation mechanisms of Object-Z, the Object-Z model of GFTSA can be customized into the models of developed distributed systems. During the customization process, besides inheriting and instantiating corresponding class schemas in the Object-Z model of GFTSA, the class schemas in the model of developed systems also need to be specified according to system requirements. These specifications cannot be supported by the reuse mechanisms of Object-Z. The following class schemas inherit and instantiate corresponding class schema in the Object-Z model of GFTSA. The items in quotation marks of these class schemas can be instantiated according to system requirements. The comments in the brackets clarify the meaning of these items.

“*objectname*”

[customized *Object* class name]

$\upharpoonright(inter\_state, \textsc{Init}, GEPropagate, UEReceive,$
$\quad UEHandle, SRReq, FromSR, ToSR)$

Object″*rename*”

[operation schema rename option]

“*newfunction*”

[inserted new function declaration]

$“newvar”$
[inserted new constant variable declaration]

$n\_states = \{“nstates”\}$ [set of normal state name]
$l\_excepts = \{″lexcepts”\}$ [set of local exception name]
$g\_excepts = \{“gexcepts”\}$ [set of global exception name]
$in\_ports = \{“inports”\}$ [set of input port name]
$out\_ports = \{“outports”\}$ [set of output port name]
$comp\_msgs = \{“compmsgs”\}$ [messages to *SharedResource*]
$coop\_msg = \{“coopmsg”\}$ [messages to other *Objects*]
$“newvarpredicate”$ [predicate of new constant variable]
$“transition”$ [concrete description of *transition* function]
$except\_context = \{“exceptcontext”\}$ [*except_context* predicate]
$except\_handle = \{“excepthandle”\}$ [*except_handle predicate*]

INIT
$inter\_state = “inistate”$ [initial state of *Object*]

$“newop”$ [inserted new operation schema]

$“connectorname”$
[customized *Connector* class name]
$Connector$
$send\_port = “s\_port”$ [sending port name]
$receive\_port = “r\_port”$ [receiving port name]

CC
$\upharpoonright(\textsc{Init}, ExceptRec, ExceptGraph)$
$CoordinatingComponent$
$“egraph”$ [description of *except_graph* function]

$“srname”$
[customized *SharedResource* class name]
$\upharpoonright(\textsc{Init}, ObList, Available, Trans, Except)$
$SharedResource$
$states = \{“states”\}$ [set of normal state name]
$trans = \{“trans”\}$ [concrete description of *trans* function]

“*systemname*”
[customized FTSystem class name]

FTSystem

“*in_component*” : “*component*” [component instance]

obs = {“*obsnames*”} [customized *Object* instance set]
cs = {“*csnames*”} [customized *Connector* instance set]
coco = “*coconame*” [customized *CC* instance]
srs = {“*srnames*”} [customized *SharedResource* instance set]

By instantiating the items in quotation marks of the class schemas shown above, we can generate the Object-Z models of develop distributed systems. In order to make this process more automatic, we investigate to build these class schemas as a template. Since the granularity of those items in quotation marks is flexible, XML-based Variant Configuration Language (XVCL) [36, 35, 75, 89] can be applied to help us build the template. XVCL is a meta-programming technique developed to facilitate building flexible, adaptable, and reusable software artifacts. All of small or large variation points can be represented as meta-expressions, which can be instantiated during the customization process according to the specific requirements.

Following the mechanism of XVCL, we build the template as generic, adaptable fragments, called x-frames. Each x-frame is an XML file combining the Object-Z specification and XVCL commands. The x-frames for the template is composed according to the five class schemas above, namely *object*, *connector*, *coco*, *sr*, and *ftsystem*. In these x-frames, each item in quotation marks is represented as a variable. The XVCL commands, such as ⟨ the value of expr=“?@var?”/⟩, ⟨ break⟩, and ⟨ ifdef ⟩, are used to mark this variable to help the adaption in the customization process. As shown in Figure 5.1, we finish the first step for customization that is building the template based on the Object-Z model of GFTSA by applying the XVCL technique. In the following, the x-frames of the built template are briefly presented to illustrate the features of the template to the designers.

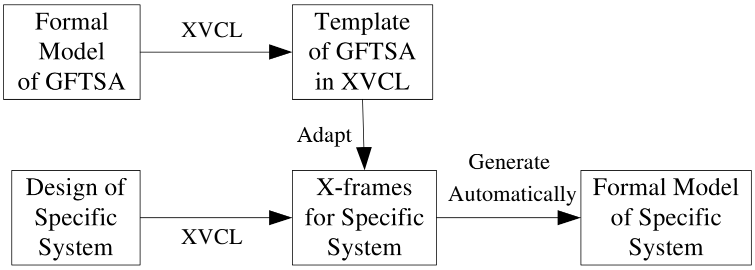

Figure 5.1: The customization process.

5.2.1 The x-frame for the fault-tolerant component-Object

The *object* x-frame is built for the fault-tolerant component *Objects* of developed distributed systems. This x-frame is composed according to the *"objectname"* class schema.

```
<x-frame name="objectname" language="latex">

\begin{class}{<value-of expr="?@objectname?"/>}

 ....

\also Object <ifdef var="rename">

[<value-of expr="?@rename?"/>]\\

</ifdef> <ifndef var="rename">\\
```

```
</ifndef> <break name="newfunction"/>
\begin{axdef}
<ifdef var="newvar">
<value-of expr="?@newvar?"/>\\
</ifdef> \ST
n\_states= \{<value-of expr="?@nstates?"/>\}\\
.....
\end{axdef}\\
\begin{init}
inter\_state =<value-of expr="?@inistate?"/> \\
\end{init}\\
<break name="newop"/>
\end{class}
</x-frame>
```

In this x-frame, the items in quotation marks of *"objectname"* class schema are marked by the XVCL commands, which are represented as variables in the $\langle\ \rangle$. These variables can be instantiated according to specific system requirements.

5.2.2 The x-frame for Connector

The *connector* x-frame is built for the *Connectors* of developed distributed systems. We compose this x-frame according to the *"connectorname"* class schema.

```
 <x-frame name="connector" language="latex">
```

```
\begin{class}{<value-of expr="?@connector_name?"/>}

\zproject (\Init, ExceptRec, ExceptGraph)\\

\also Connector\\

\begin{axdef}

send\_port=<value-of expr="?@s_port?"/>\\

receive\_port=<value-of expr="?@r_port?"/>\\

\end{axdef}

\end{class}

</x-frame>
```

The items in quotation marks of *"connectorname"* class schema are expressed as variables in the XVCL command *value-of*. The *s_port* and *r_port* represent the sending port name and receiving port name correspondingly, which are used by *Objects* to communicate with each other.

5.2.3 The x-frame for CoordinatingComponent

The *coordatingcomponent* x-frame is built for the *CoordinatingComponents* of developed distributed systems. This x-frame is composed according to the *"CC"* class schema.

```
 <x-frame name="coco" language="latex">

\begin{class}{CC}

\also CoordinatingComponent\\

\begin{axdef}
```

```
<break name="egraph"/>

\end{axdef}

\end{class}

</x-frame>
```

The item *e_graph* is expressed as variable in the XVCL commands *break*, which can be instantiated via adaption by inserting the declaration of *except_graph* function.

5.2.4 The x-frame for SharedResource

The *sharedresource* x-frame is built for the *SharedResources* of developed distributed systems. We compose this x-frame according to the *"srname"* class schema.

```
 <x-frame name="sr" anguage="latex">

\begin{class}{<value-of expr="?@srname?"/>}

\zproject (\Init, ObList, Available, Trans, Except)\\

\also SharedResource\\

\begin{axdef}

states=\{<value-of expr="?@states?"/>\}\\

trans=\{<value-of expr="?@trans?"/>\}\\

\end{axdef}

\end{class}

</x-frame>
```

The items in quotation marks of *"srname"* class schema are expressed as variables in the XVCL command *value-of*. The *states* represents a set of normal state that *SharedResource* can be in. The *trans* represents the *trans* function, which is used to declare the state transition of *SharedResource*.

5.2.5 The x-frame for Fault Tolerant System-ftsystem

The *ftsystem* x-frame is built for the synchronization of components and connectors of developed distributed system. This x-frame is composed according to the *"systemname"* class schema.

```
 <x-frame name="ftsystem" language="latex">
\begin{class}{<value-of expr="?@systemname?"/>}
\also System\\
\begin{anonschema}
<while using-items-in="in_component,component">
    <value-of expr="?@in_component?"/>:
    <value-of expr="?@component?"/>\\
</while> \ST
obs=\{<value-of expr="?@obsnames?"/>\}\\
cs=\{<value-of expr="?@csnames?"/>\}\\
coco=<value-of expr="?@coconame?"/>\\
srs=\{<value-of expr="?@srname?"/>\}\\
\end{anonschema}
\end{class}
```

```
</x-frame>
```

The items in quotation marks of *"systemname"* class schema are expressed as variables, which are marked by the XVCL command *value-of*. These variables can be instantiated according to the specific system requirements.

5.3 A Case Study-Sales Control System (SCS)

GFTSA is proposed to provide a generic framework to guide the development of distributed systems with high safety requirements. The Object-Z model of GFTSA can provide explicit idioms & patterns to the system designers. A template is built to help the customization from the Object-Z model of GFTSA to the models of developed systems. In this section, a case study of Sales Control System (SCS) is presented to demonstrate how GFTSA can guide the high level system design of distributed systems with high safety requirements.

5.3.1 Sales Control System (SCS)

The distributed system Sales Control System (SCS) [7, 87] is designed to maintain a database describing all the products to be sold so that many distributed sales points can obtain the correct prices of the items selected by the customers, which needs to satisfy a high reliable requirements. The SCS consists of a database, a set of control points and a set of sales points. Figure 5.2 shows an example of SCS, which is composed of two control points, a database and three sales points.

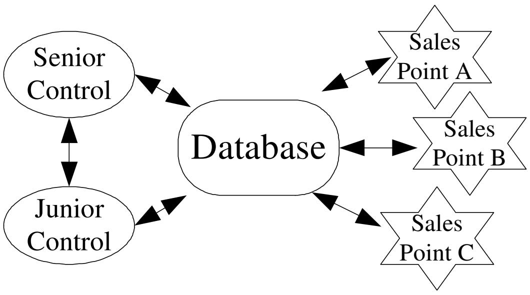

Figure 5.2: The Sales Control System.

A control point provides the interface that allows the human manager of the system to update the product information in the database at run time. We assume that such updating is regarded as a very critical activity and consequently, to guard against fraud, the policy is that the senior control point needs to monitor and, if necessary, to correct the updates made by the junior control point. Therefore, the senior and junior control points cooperate with each other to update the database. The database stores product information which can be accessed by control and sales points. This competitive concurrency needs to guarantee the transaction semantics of the database.

According to the box-and-line patterns of GFTSA shown in Figure 3.1, the SCS is composed of five *Objects*, called *SeniorControl*, *JuniorControl*, *SalesPointA*, *SalesPointB*, *SalesPointC* and a *SharedResource*, called *Database*. Two *Connectors*, called *SJC* and *JSC*, are used to assist the communication between *SeniorControl* and *JuniorControl*. A *CoordinatingComponent*, called *CC*, is also involved in the SCS to implement *coordinated error recovery mechanism*.

5.3.2 Generation of Formal Model of SCS

Based on the description of SCS, we investigate to develop the formal model of SCS by customizing the built template. The five primitive x-frames in the template can be customized via adaption. The adaption implies that a new x-frame for one component in the developed system can be built based on the corresponding primitive x-frame in the template by using XVCL command ⟨ adapt ⟩, and instantiating the variation points.

Building x-frames for Formal Model of SCS

We build x-frames for the formal model of SCS based on the primitive x-frames of the built template, as a step shown in the customization process of Figure 5.1. Figure 5.3 describes x-frame adaption relationship between the SCS and the built template.

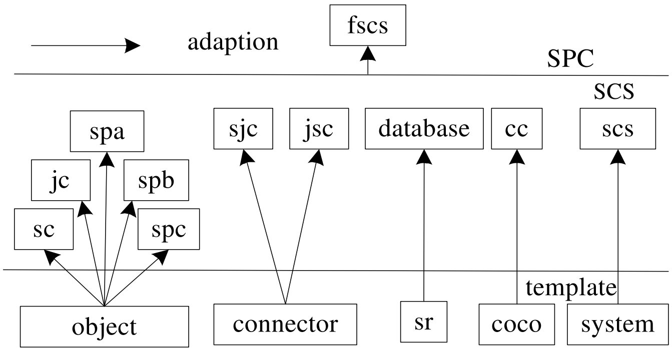

Figure 5.3: The x-frame Adaption Relationship of SCS.

The *sc*, *jc*, *spa*, *spb*, and *spc* x-frames are built for *SeniorControl*, *JuniorControl*, *SalesPointA*, *SalesPointB*, and *SalesPointC* correspondingly. The *sjc* and *jsc* x-frames are built for the connectors *SJC* and *JSC* correspondingly. The *database* x-frame is built for the *Database* and the *cc* x-frame is built for the *CC* component. The *scs* x-frame is built to describe how these components & connectors synchronize. We use *jsc* as an example to illustrate how we can compose the x-frame for the connector *JSC*.

```
<x-frame name="jsc" language="latex">
<set var="connector_name" value="JSC"/>
<set var="s_port" value="JSC\_Out"/>
<set var="r_port" value="JSC\_In"/>
<adapt x-frame="connector.xvcl"/>
</x-frame>
```

In this *jsc* x-frame, we adapt *connector* x-frame in the template of GFTSA, and set values to the variables defined in the *connector* x-frame. The *JSC* is the name of the connector, and also the name of the class schema for this connector, which is set to the variable *connector_name*. The *JSC_Out* and *JSC_In* are the names for sending port and receiving port of the *JSC* connector, which are set to the variables *s_port* and *r_port*. The composition of all other x-frames follow such mechanisms.

The *fscs* SPC is the root of x-framework for SCS, which adapts all of the ten x-frames built for SCS. During x-framework processing, the XVCL processor interprets the XVCL commands contained in the *fscs* SPC, traverses an x-framework, performs adaption by executing XVCL commands embedded in x-frames, and emits the formal model of *SCS* to the *fscs.tex* file.

Formal Model of SCS

By running the XVCL processor, we can auto-generate the Object-Z model of SCS. Figure 5.4 shows the model design of SCS in the box-and-line fashion guided by the pattern of GFTSA.

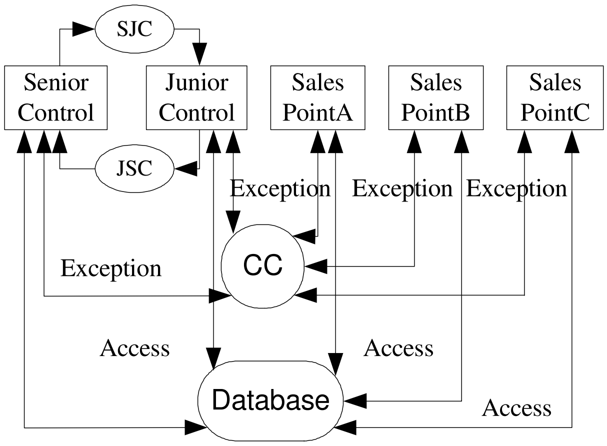

Figure 5.4: GFTSA Architecture View of SCS.

We use a representative class schema *JuniorControl* to illustrate the features of the Object-Z model of *SCS*. The *JuniorControl* class represents the *Objects* in the *SCS* which describes how the *JuniorControl* point interacts with *SeniorControl* point to update the product information stored in the *Database*, and how to deal with local and global exceptions. The *JuniorControl* class schema inherits the *Object* class schema. The local exception *NetworkDisconnected* defined in *L_excepts* represents that the network cannot work when the *JuniorControl* point is waiting for the authorization from the *SeniorControl* point. A *Local_ExceptHandle* function is defined to handle this exception. After the network can work, the *JuniorControl* point needs to send the *RequestUpdate* to the *SeniorControl* point again.

The global exception *InformationLost* represents that the *Database* has lost some product information. A *Global_ExceptHanle* function is defined to handle this exception. The *JuniorControl* point needs to recover the product information in the *Database*. The *Trans* function defines not only the normal state transitions, but also exceptional state transitions of the *JuniorControl* point.

$$
\begin{array}{l}
\textit{JuniorControl} \\
\hline
\upharpoonright(\textit{inter\_state}, \textsc{Init}, \textit{GlobalExceptPropagate}, \textit{UniExceptReceive}, \\
\quad \textit{UniExceptHandle}, \textit{SRRequest}, \textit{FromSR}, \textit{ToSR}) \\
\textit{Object} \\
\quad \textit{Local\_ExcepthHandle} : \textit{OBSTATE} \rightarrow (\textit{PORT} \times \textit{MSG}) \\
\quad \textit{Global\_ExceptHandle} : \textit{OBSTATE} \rightarrow \textit{Handler} \\
\hline
\quad \textit{Local\_ExceptHandle}(\textit{NetworkDisconnected}) = \\
\qquad (\textit{JSC\_Out}, \textit{RequestUpdate}) \\
\quad \textit{Local\_ExceptHandle}(\textit{InformationLost}) = \\
\qquad \textit{DatabaseRecover} \\
\quad \textit{n\_states} = \{\textit{NormalProcess}, \textit{AuthorizeRequest}\} \\
\quad \textit{l\_excepts} = \{\textit{NetworkDisconnected}\} \\
\quad \textit{g\_excepts} = \{\textit{InformationLost}\} \\
\quad \textit{in\_ports} = \{\textit{SJC\_In}\} \\
\quad \textit{out\_ports} = \{\textit{JSC\_Out}\} \\
\quad \textit{comp\_msgs} = \{\textit{ProductUpdate}\} \\
\quad \textit{coop\_msg} = \{(\textit{SJC\_In}, \textit{UpdateApproved}), \\
\qquad (\textit{JSC\_Out}, \textit{RequestUpdate})\}
\end{array}
$$

$$\begin{array}{l}
Trans = \{((NormalProcess, (NULL, NULL)), \\
\quad (AuthorizeRequest, (JSC\_Out, RequestUpdate))), \\
\quad ((AuthorizeRequest, (SJC\_In, UpdateApproved)), \\
\quad (NormalProcess, (NULL, NULL))), \\
\quad ((AuthorizeRequest, (NULL, NetworkDisconnected)), \\
\quad (NetworkDisconnected, (NULL, NULL))), \\
\quad ((NormalProcess, (NULL, InformationLost)), \\
\quad (InformationLost, (NULL, NULL))), \\
except\_context = \{(NetworkDisconnected, Local\_ExceptHandle), \\
\qquad (InformationLost, Global\_ExceptHandle)\} \\
except\_handle = \{(Local\_ExceptHandle, AuthorizeRequest), \\
\qquad (Global\_ExceptHandle, NormalProcess)\}
\end{array}$$

INIT

$$inter\_state = NormalProcess$$

5.3.3 Reasoning about SCS

By customizing our built template based on the Object-Z model of GFTSA, we can auto-generate the formal model of *SCS*. Since the *SCS* has high safety requirements that when exceptions raised, the *SCS* can deal with these exceptions, the formal model of *SCS* needs to preserve fault tolerant properties. We can formally reason about the fault tolerant properties based on the Object-Z model of *SCS*. The process of reasoning needs to derive fault tolerant properties from the generated model of SCS by using the reasoning rules of Object-Z [72]. The following items show that *SCS* can preserve significant fault tolerant properties, which are expressed as theorems.

1. When the *InformationLostA* is raised in the *SalesPointA*, which represents that the *SalesPointA* cannot get the product information from the *Database*, the *SCS* can tolerate this exception. This property can be formally expressed as follows.

   Theorem

   $$\begin{array}{l} SCS :: spa.inter\_state = InformationLostA \vdash \\ \quad \forall\, scs : SCS;\ ob : scs.obs \bullet \\ \qquad ob.inter\_state' \in ob.n\_state \end{array}$$

   As an intermediate step, it is useful to informally think of proof strategy. When a global exception, namely *InformationLostA*, is raised in the *SalesPointA*, the *SalesPointA* can use *GlobalExceptPropagate* operation to send this global exception out to *CC* and *CC* can use *ExceptRec* operation to receive this global exception. Two operations are combined in the *ExceptPropagate* operation expression declared in *SCS* class schema. Since the sequence *exceptions* is not empty, the *ExceptGraph* operation in the *CC* class sends out the *uni_exception!*. The *UEReceive* operation in each *FTComponent* of SCS can receive this *uni_exception?*, which is expressed in the *ExceptGraph* operation declared in the *SCS* class schema. When each *FTComponent* of *SCS* receives the *uni_exception?*, the state is changed to normal state. These transformations assure that *FTComponents* in the *SCS* can handle the global exception. Formal proof based on this strategy is constructed in the following.

Proof

$$
\begin{array}{l}
\mathit{SCS} \vdash \mathit{spa.inter\_state} = \mathit{InformationLostA} \\
\mathit{SCS} \vdash \mathit{spa} \in \mathit{SalesPointA} \\
\mathit{SalesPointA} \vdash \mathit{InformationLostA} \in \mathit{g\_excepts} \\
\mathit{SalesPointA} :: \mathit{inter\_state} \in \mathit{g\_excepts} \vdash \\
\qquad\qquad \mathit{GlobalExceptPropagate} \\
\hline
\mathit{SCS} \vdash \mathit{spa.GlobalExceptionPropagate} \\
\mathit{SalesPointA} :: \mathit{GlobalExceptionPropagate} \vdash \\
\qquad\qquad \mathit{exception!} = \mathit{inter\_state} \\
\hline
\mathit{SalesPointA} \vdash \mathit{exception!} = \mathit{InformationLostA} \\
\mathit{SCS} \vdash \mathit{ExceptPropagate} \\
\hline
\mathit{SCS} \vdash \mathit{coco.ExceptRec} \\
\mathit{SCS} \vdash \mathit{coco} \in \mathit{CC} \\
\hline
\mathit{CC} :: \mathit{ExceptRec} \vdash \mathit{exceptions}' = \\
\qquad \mathit{exceptions} \frown \langle \mathit{InformationLostA} \rangle \\
\mathit{CC} :: \mathit{exceptions}' \neq \langle\ \rangle \vdash \mathit{ExceptGraph} \\
\hline
\mathit{CC} :: \mathit{ExceptGraph} \vdash \\
\qquad \mathit{uni\_exception!} = \mathit{except\_graph}(\mathit{exceptions}') \\
\qquad\qquad = \mathit{InformationLost} \\
\mathit{SCS} \vdash \mathit{ExceptGraph} \\
\hline
\mathit{SCS} :: \mathit{ob} : \mathit{obs} \vdash \mathit{ob.UEReceive} \\
\mathit{SCS} :: \mathit{obs} : \mathbb{P} \downarrow \mathit{FTComponent} \vdash \mathit{ob} \in\downarrow \mathit{FTComponent} \\
\mathit{FTComponent} :: \mathit{UEReceive} \vdash \mathit{ue\_rec}' = 1 \wedge \\
\qquad\qquad \mathit{inter\_state}' = \mathit{uni\_exception?} \\
\qquad\qquad\qquad = \mathit{InformationLost} \\
\mathit{FTComponent} :: \mathit{ue\_rec} = 1 \wedge \mathit{inter\_state} = \\
\mathit{uni\_exception?} \vdash \mathit{UniExceptHandle} \\
\mathit{FTComponent} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\mathit{except\_handle}(\mathit{except\_context}(\mathit{InformationLost})) \\
\mathit{SCS} \vdash \{\mathit{jc}, \mathit{sc}, \mathit{spa}, \mathit{spb}, \mathit{spc}\} \in \mathbb{P} \downarrow \mathit{FTComponent} \\
\hline
\mathit{SalesPointA} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\qquad \mathit{except\_handle}(\mathit{except\_context} \\
\qquad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\mathit{JuniorControl} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\qquad \mathit{except\_handle}(\mathit{except\_context}
\end{array}
$$

$$\begin{array}{l}
\quad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\mathit{SeniorControl} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\quad \mathit{except\_handle}(\mathit{except\_context} \\
\quad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\mathit{SalesPointB} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\quad \mathit{except\_handle}(\mathit{except\_context} \\
\quad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\mathit{SalesPointC} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\quad \mathit{except\_handle}(\mathit{except\_context} \\
\quad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\hline
\mathit{SCS} :: \mathit{scs} : \mathit{SCS};\ \mathit{ob} : \mathit{scs.obs} \vdash \\
\quad \mathit{ob.inter\_state}' \in \mathit{ob.n\_state}
\end{array}$$

2. When the *InformationLostA* is raised in the *SalesPointA*, and concurrently the *InformationLostB* is raised in the *SalesPointB*, the *SCS* can also handle these two concurrent global exceptions and recover system to normal state.

Theorem

$$\begin{array}{l}
\mathit{SCS} :: \mathit{spa.inter\_state} = \mathit{InformationLostA} \land \\
\quad \mathit{spb.inter\_state} = \mathit{InformationLostB} \\
\qquad \vdash \forall\, \mathit{scs} : \mathit{SCS};\ \mathit{ob} : \mathit{scs.obs} \bullet \\
\qquad\qquad \mathit{ob.inter\_state}' \in \mathit{ob.n\_state}
\end{array}$$

When the *InformationLostA* raised in *SalesPointA* and the *InformationLostB* raised in *SalesPointB* concurrently, each of them can use *GlobalExceptPropagate* operation to send the exception out to the *CC* and the *CC* can execute the *ExceptRec* operation to receive these two global exceptions. The *GlobalExceptPropagate* and *ExceptRec* operations are combined in the *ExceptPropagate* operation expression declared in the *SCS* class schema. Because the sequence *exceptions* is not empty, the *ExceptGraph* operation in the *CC* class schema can send out the *uni_exception!* which covers *InformationLostA* and *InformationLostB*. The *UEReceive* operation in each *FTCompo-*

nent of SCS can receive this *uni_exception?*. Two operations *ExceptGraph* and *UEReceive* are combined in the *ExceptGraph* operation expression declared in the *SCS* class schema. When each *FTComponent* in the SCS receives the *uni_exception?*, the state is changed to normal state. Following is the formal proof based on this strategy.

Proof

$$
\begin{array}{l}
SCS \vdash spa.inter\_state = InformationLostA \wedge \\
\qquad spb.inter\_state = InformationLostB \\
SCS \vdash spa \in SalesPointA \wedge spb \in SalesPointB \\
\hline
SalesPointA \vdash InformationLostA \in g\_excepts \\
SalesPointA :: inter\_state \in g\_excepts \\
\qquad \vdash GlobalExceptPropagate \\
SalesPointB \vdash InformationLostB \in g\_excepts \\
SalesPointB :: inter\_state \in g\_excepts \\
\qquad \vdash GlobalExceptPropagate \\
\hline
SCS \vdash spa.GlobalExceptionPropagate \wedge \\
\qquad spb.GlobalExceptionPropagate \\
SalesPointA :: GlobalExceptionPropagate \\
\qquad \vdash exception! = inter\_state \\
SalesPointB :: GlobalExceptionPropagate \\
\qquad \vdash exception! = inter\_state \\
\hline
SalesPointA \vdash exception! = InformationLostA \\
SalesPointB \vdash exception! = InformationLostB \\
SCS \vdash ExceptPropagate \\
\hline
SCS \vdash coco.ExceptRec \\
SCS \vdash coco \in CC \\
\hline
CC :: ExceptRec \vdash exceptions' = exceptions \\
\qquad {}^\frown\langle InformationLostA, \\
\qquad\qquad InformationLostB\rangle \\
CC :: exceptions' \neq \langle\ \rangle \vdash ExceptGraph \\
\hline
CC :: ExceptGraph \vdash uni\_exception! \\
\qquad = except\_graph(exceptions') \\
\qquad\qquad = InformationLost \\
SCS \vdash ExceptGraph \\
\hline
SCS :: ob : obs \vdash ob.UEReceive \\
SCS :: obs : \mathbb{P} \downarrow FTComponent \vdash ob \in \downarrow FTComponent
\end{array}
$$

$$
\begin{array}{l}
\mathit{FTComponent} :: \mathit{UEReceive} \vdash \mathit{ue\_rec}' = 1 \land \\
\quad \mathit{inter\_state}' = \mathit{uni\_exception}? \\
\qquad = \mathit{InformationLost} \\
\mathit{FTComponent} :: \mathit{ue\_rec} = 1 \land \\
\mathit{inter\_state} = \mathit{uni\_exception}? \vdash \mathit{UniExceptHandle} \\
\mathit{FTComponent} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\quad \mathit{except\_handle} \\
\qquad (\mathit{except\_context}(\mathit{InformationLost})) \\
\mathit{SCS} \vdash \{\mathit{jc}, \mathit{sc}, \mathit{spa}, \mathit{spb}, \mathit{spc}\} \in \mathbb{P} \downarrow \mathit{FTComponent} \\
\hline
\mathit{SalesPointA} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\quad \mathit{except\_handle}(\mathit{except\_context} \\
\quad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\mathit{SalesPointB} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\quad \mathit{except\_handle}(\mathit{except\_context} \\
\quad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\mathit{SalesPointC} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\quad \mathit{except\_handle}(\mathit{except\_context} \\
\quad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\mathit{JuniorControl} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\quad \mathit{except\_handle}(\mathit{except\_context} \\
\quad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\mathit{SeniorControl} :: \mathit{UniExceptHandle} \vdash \mathit{inter\_state}' = \\
\quad \mathit{except\_handle}(\mathit{except\_context} \\
\quad (\mathit{InformationLost})) = \mathit{NormalProcess} \\
\hline
\mathit{SCS} :: \mathit{scs} : \mathit{SCS};\ \mathit{ob} : \mathit{scs}.\mathit{obs} \vdash \\
\quad \mathit{ob}.\mathit{inter\_state}' \in \mathit{ob}.\mathit{n\_state}
\end{array}
$$

5.4 Conclusion

In this chapter, we investigate to build a template based on the Object-Z model of GFTSA by using the XVCL technique, which can help auto-generate the formal models of developed distributed systems guided by GFTSA via customization. Following the XVCL mechanism, the template is built as generic, adaptable x-frames, which are written as the combination of Object-Z specification and XVCL commands. When developing a distributed systems with high reliability requirements, we can just compose x-frames for this system by adapting the x-frames in the built template. By running the XVCL processor, we can generate the Object-Z model of developed system from these composed x-frames automatically.

A case study of *SCS* is presented to illustrate the customization process. Guided by the pattern of GFTSA, we can design the structure of *SCS*. According to the built template, we compose specific x-frames for *SCS* by adapting corresponding x-frame and instantiating the variables in the corresponding x-frame. A *fscs* SPC file is built to compose all the specific x-frames together. By running XVCL processor with *fscs* SPC, we can generate formal model of *SCS* automatically. In order to demonstrate that the developed system guided by GFTSA preserves the fault tolerant properties, we formally reason about the fault tolerant properties of SCS by using the reasoning rules of Object-Z. The formal reasoning demonstrates that the developed system guided by GFTSA can satisfy the high reliability requirements.

Chapter 6

Mechanical Verification of GFTSA

This chapter presents the mechanical verification of fault tolerant properties of GFTSA by using the theorem prover of PVS, and a template which is built based on the PVS model of GFTSA to help the mechanical verification of safety critical distributed systems.

6.1 Introduction

In order to provide a generic framework for the development of distributes systems with high reliability requirements, we have proposed a Generic Fault Tolerant Software Architecture (GFTSA). Since GFTSA incorporates fault tolerant techniques to deal with the exceptions, GFTSA can preserve fault tolerant properties. Based on the Object-Z model of GFTSA, we can formally reason about the fault tolerant properties of GFTSA. These formal reasoning involves showing that the fault tolerant properties, expressed as theorems, can be derived from the Object-Z model of GFTSA by using the reasoning rules of Object-Z. Though Object-Z model of GFTSA can provide precise analysis and documentation to the users, since Object-Z lacks of tool support for mechanical verification, the formal reasonings about GFTSA we have done are all manual-based, which are laborious and error-prone. Thus, we investigate to use prover to mechanically verify the fault tolerant properties of GFTSA. Prototype Verification System (PVS)[57, 59, 68, 58] is a good candidate for us, because the theorem prover of PVS can provide mechanical proof support for the verification.

The Prototype Verification System (PVS) is a proof system developed at SRI. PVS has a powerful interactive theorem prover and its automation suffices to prove many results automatically. PVS differs from most other interactive theorem provers in the power of its basic steps which can be decision procedure for automatic rewriting, induction, and other relatively large units of deduction. PVS differs from other highly automated theorem provers in being directly controlled by the user. PVS has been applied successfully to large and difficult application in both academic and industrial settings[31, 64].

As the theorem prover of PVS only supports the model in the PVS specification language, we need to embed GFTSA model in PVS. One way we can do is that we encode the whole set of Object-Z notations into PVS. However, it is tedious and arduous to take all the Object-Z semantics into account and construct Object-Z semantics in PVS properly. As our focus is the mechanical verification of GFTSA, instead, we suppose to just embed the Object-Z model of GFTSA in PVS.

We can develop safety critical distributed systems guided by GFTSA. The developed systems guided by GFTSA also can preserve the fault tolerant properties. Since the theorem prover of PVS only supports the mechanical verification of model in PVS specification language, we need to get the PVS models of developed systems. Based on the PVS model of GFTSA, we can generate the PVS models of safety critical distributed systems via customization. Following the customization methodology which is used in the generation of Object-Z models of developed systems guided by GFTSA, we can build a template for the customization by using the XVCL technique. The template is composed of primitive x-frames importing the theories in the PVS model of GFTSA. By customizing this template, we can generate the PVS models of developed systems automatically. Based on the generated PVS models, the theorem prover of PVS can mechanically verify the fault tolerant properties of such models.

The remainder of the chapter is organized as follows. In section 2, we present the formal model of GFTSA in PVS specification language. Section 3 presents the mechanical verification of fault tolerant properties of GFTSA. In section 4, a template is built based on the PVS model of GFTSA. Section 5 concludes the chapter.

6.2 PVS Model of GFTSA

The formal model of GFTSA in Object-Z can provide an explicit features of GFTSA to the system designers in a compact and understandable way [88]. Since Object-Z lacks of tool support for mechanical verification, in order to provide a mechanical proof support for the verification of fault tolerant properties, we embed the GFTSA model in the PVS environment to use the theorem prover of PVS.

The Object-Z model of GFTSA is composed of generic types and several class schemas for components & connectors of GFTSA. The PVS specification language supports modularity and reuse by means of parameterized *theories*. Therefore, the class schemas for components & connectors of GFTSA can be built as *theories*. A *theory* consists of a series of *declarations*, which provide names for types, constants, variables, axioms, and formulas. We still use the *Queue* system as an example, which has been used to illustrate the basic features of Object-Z in Section 2.1, to explain the basic features of PVS specification language.

```
queue  [ Item: TYPE+ ] : THEORY

  BEGIN
```

```
i: VAR nat

items: TYPE=[#size: nat, elements: ARRAY[{i|i<size} -> Item]#]

itms: VAR items

item: VAR Item

e: Item

nonemptyqueue?(itms): bool=(size(itms)>0)

nitms: VAR (nonemptyqueue?)

empty: items=(#size:=0, elements:=(LAMBDA (j:{i|i<0}):e)#)

join(item, itms): items=(#size:=size(itms)+1, elements:=elements(itms)

     WITH [(size(itms)):=item]#)

leave(item,nitms): items=(#size:=size(nitms)-1, elements:=(LAMBDA

      (j:{i|i<size(nitms)-1}): elements(nitms)(j+1))#)

END queue
```

The *queue* theory is generic with the parameter *Item*. The *Item* is declared as an uninterpreted and nonempty type. The *items* is declared as a record type. In the type of *items*, the *size* and *elements* are two fields of this type. The *i, itms, item* are all declared as variables associated with specific types. The *e* is declared as a constant associated with *Item* type. The *nonemptyqueue?* is declared as a predicate of *itms*, which also could be used as a type. Two *join* and *leave* operations are declared to specify that one *item* joins and leaves the queue, associated with the change of *size* and *elements* fields. The *queue* theory can be reused later to specify *object*, *coordinatingcomponent*, and *sharedresource* theories.

Similar to the structure of Object-Z model of GFTSA, the PVS model of GFTSA also includes *global type*, *object*, *connector*, *coordinatingcomponent*, *shareresource*, and *ftsystem* theories, which will be presented in the following sections. We can use typecheck tool of PVS to check for semantic errors of such PVS model.

6.2.1 Generic Type

These declared types can be used to declare constants and variables in the following *object*, *connector*, *coordinatingcomponent*, *shareresource*, and *ftsystem* theories.

```
 generictype: THEORY
  BEGIN
  PORT: TYPE+
  MSG:  TYPE+
  OBSTATE: TYPE+
  SRSTATE: TYPE+
  EH: TYPE+
  RESULT:  TYPE={tolerate, stop}
  SIG:  TYPE={0,1}
  Fail: OBSTATE
  OBID: TYPE+
  SRID: TYPE+
  CONID: TYPE+
  CCID: TYPE+
END generictype
```

The *PORT* is an uninterpreted and nonempty type to represent the ports used by *Objects* to communicate with each other. The *MSG* is an uninterpreted and nonempty type to represent the communicated messages. The *OBSTATE* and *SRSTATE* represent the states of *Objects* and *SharedResources* correspondingly. The *EH* represents the exception handler. The *RESULT* and *SIG* are two enumeration type declarations. The *OBID, SRID, CONID, and CCID* are declared to represent the identifications of *Object*, *SharedResource*, *Connector*, and *CoordinatingComponent* correspondingly. After giving these types, we do not need to define these types in the following *theories.*

6.2.2 CoordinatingComponent

The *coordinatingcomponent* theory describes how the *CoordinatingComponent* in GFTSA implements the *coordinated error recovery mechanism* when a global exception is raised in an *Object* or multiple global exceptions are raised concurrently in different *Objects.* The *coordinatingcomponent* theory imports the *generictype* theory, and the parameterized theory *queue* instantiated with type *[OBSTATE].*

```
coordinatingcomponent: THEORY
  BEGIN
   IMPORTING generictype, queue[OBSTATE]
  except_graph: [items[OBSTATE] -> OBSTATE]
  exception: OBSTATE
  CC: TYPE=[#exceptions: items[OBSTATE], uni_exception:OBSTATE#]
```

```
 cc: VAR CC

 emptycc:CC=(#exceptions:=empty, uni_exception:=e#)

 ExceptRec(cc): CC=(#exceptions:=join(exception,exceptions(cc)),

                  uni_exception:=exception#)

 ExceptGraph(cc):CC=

 IF exceptions(ExceptRec(cc))/=empty THEN

 (#exceptions:=empty, uni_exception:=

          except_graph(exceptions(ExceptRec(cc)))#)

 ELSE emptycc

 ENDIF

END coordinatingcomponent
```

In the *coordinatingcomponent* theory, the *except_graph* is declared as a function to resolve several concurrently raised global exceptions into an *universal exception*, namely *uni_exception*. The *ExceptRec* function is declared to represent how the *coordinatingcomponent* receives *exception* from *Objects*. The *ExceptGraph* function is responsible for resolving these received exceptions by the *except_graph* to the *uni_exception*.

6.2.3 Fault-Tolerant Component-Object

The *object* theory describes the stable activities, and error recovery activities of *Object* component, which represents the fault-tolerant components in GFTSA. The *object* imports the *generictype* theory, the parameterized *queue* theory instantiated with type *[SRID]*, and *coordinatingcomponent* theory.

```
object: THEORY
  BEGIN
    IMPORTING generictype, queue[SRID], coordinatingcomponent
   n_states: setof[OBSTATE]
   l_excepts: setof[OBSTATE]
   g_excepts: setof[OBSTATE]
   ......
   transition: [[OBSTATE, [PORT -> MSG]] -> [OBSTATE, [PORT -> MSG]]]
   except_context: [OBSTATE -> EH]
   except_handle: [EH -> OBSTATE]
   ......
   Transition(ob): Object=
      IF member(inter_state(ob),n_states) THEN
   (#inter_state:=PROJ_1(transition(inter_state(ob),
  ......
  LocalExceptHandle(ob):Object=
   IF member(inter_state(ob),l_excepts) THEN
   .......
  GlobalExceptPropagate(ob): OBSTATE=
   IF member(inter_state(ob),g_excepts) AND ue_rec(ob)=0 THEN
   inter_state(ob)
  ......
```

```
 UniExceptReceive(ob,ccp): Object=

  IF uni_exception=except_graph(exceptions(ExceptRec(ccp))) THEN

 ......

 UniExceptHandle(ob): Object=

  IF member(inter_state(ob), g_excepts) AND ue_rec(ob)=1 THEN

......

 END object
```

In the *object* theory, the declared constants *n_states, l_excepts, and g_excepts* represent three different sets of states that an *Object* can be in: a set of normal states, a set of local exception states, and a set of global exception states. The *transition* function specifies the state change of an *Object* when receiving or sending the messages to other *Objects*. The *except_context* function is declared to model that any exception occurring in the *Object* has a corresponding exception handler. The function *exception_handle* is used to check whether the exception handler deals with the exception successfully. The *Transition* function denotes the state transitions of an *Object* according to the *transition* function. The *LocalExceptHandle* function specifies how the *Objects* deals with local exception. The *GlobalExceptPropagate, UniExceptReceive, and UniExceptHandle* denote how the *Object* implements the *coordinated error recovery mechanism.*

6.2.4 Connector

The *connector* imports the *generictype* and *object* theories. The *connector* theory describes that the *Connector* in GFTSA is responsible for connecting *send_port* of a *send_ob* and *receive_port* of another *receive_ob* to transfer the message. These properties are specified in two axioms.

```
connector : THEORY
  BEGIN
    IMPORTING generictype, object
  send_port, receive_port: PORT
  send_ob, receive_ob: VAR Object
  connectorprd1:AXIOM
    member(send_port, out_ports(Constant(send_ob))) AND
      member(receive_port, in_ports(Constant(receive_ob)))
  connectorprd2:AXIOM
     EXISTS(msg: MSG): msg=coop_msg(Constant(send_ob))(send_port) AND
   coop_msg(Constant(receive_ob))(receive_port)=msg
  END connector
```

6.2.5 SharedResource

The *sharedresource* theory models how the *SharedResource* can guarantee the transaction semantics when receiving messages from *Objects* and preserve consistent state when facing exceptions. The *shareresource* imports the *generictype* theory, the parameterized theory *queue* instantiated with type *[SRID]*, *object* theory, and *coordinatingcomponent* theory.

```
sharedresource: THEORY
  BEGIN
   IMPORTING generictype, queue[OBID], object, coordinatingcomponent
  states: setof[SRSTATE]
  trans:  [[SRSTATE, MSG] -> SRSTATE]
  SR: TYPE=[#semaphore: SIG, ob_qlist:(nonemptyqueue?[OBID]),
    sr_state: SRSTATE, checkpoint: SRSTATE#]
  ......
  ObList(sr): SR=
  IF req_sr=srid THEN
  (#semaphore:=0, ob_qlist:=join(req_ob,ob_qlist(sr)),
  ......
  Available(sr): answer=
  IF semaphore(sr)=0 AND ob_qlist(sr)/=empty  THEN
  (#semaphore:=1, ob_qlist:=leave(ans_ob, ob_qlist(sr)),ans_sr:=srid#)
  ......
  Trans(sr): SR=
   IF semaphore(sr)=1 AND to_sr=srid THEN
  ......
  Except(sr,ccp): SR=
   IF uni_exception=except_graph(exceptions(ExceptRec(ccp))) THEN
    (#semaphore:=0, ob_qlist:=empty, sr_state:=checkpoint(sr),
```

```
      checkpoint:=checkpoint(sr)#)
   ELSE sr
   ENDIF
 END sharedresource
```

Referring to the *SharedResource* class schema in the Object-Z model of GFTSA, the constants *states, trans* declared in the axiom definition are also declared as constants associated with specific types. These constants all have the same meaning as illustrated in the *SharedResource* class schema. The variables declared in the state schema of Object-Z model of GFTSA are declared as the fields in the *SR* record type. The *ObList* function specifies that the *SharedResource* receives access request from an *Object*. The *Available* function models that when the *SharedResource* is available, how it sends out a signal to the *Object*. The *Trans* function specifies the state transitions of *SharedResource* according to the *trans* function. The *Except* function describes that the state of *SharedResource* need to roll back to the normal state recorded in the *checkpoint* when facing exceptions.

6.2.6 Fault-Tolerant System-ftsystem

The *ftsystem* theory imports all the theories for the components & connector of GFTSA. Several formulas, such as *Propagate* which can be used in the mechanical verification of properties, all can be declared in this theory. The properties which need to be verified are all declared as *LEMMA* in this theory, such as *pred1_ft*.

```
ftsystem: THEORY

  BEGIN

   IMPORTING object, connector, coordinatingcomponent, sharedresource

   ......

   Propagate: AXIOM

    member(inter_state(obj),g_excepts) AND ue_rec(obj)=0 IMPLIES

     GlobalExceptPropagate(obj)=inter_state(obj)

   ExceptPropagate: AXIOM

    exception=GlobalExceptPropagate(obj)

   NonEmpty: AXIOM

    exception=inter_state(obj) IMPLIES

        exceptions(ExceptRec(ccp)) /=empty

   ......

   pred1_ft: LEMMA

    (EXISTS (obj:Object):

       member(inter_state(obj),g_excepts) AND ue_rec(obj)=0)IMPLIES

         (FORALL (obj:Object),(ccp: CC):

   ......

   pred4_ft: LEMMA

     (FORALL (fobs: setof[OBID]): disjoint?(fobs, critical)) IMPLIES

        (FORALL (obj: Object),

          (fobs: setof[OBID]):systemrecover(obj, fobs)=Init)

  end ftsystem
```

The *Propagete AXIOM* means that when a global exception raised in an *Object*, it will send out this exception. The *ExceptPropagate and NonEmpty AXIOMS* specify that, when *CoordinatingComponent* receives the global exception, the exception list will be nonempty. The *Lemmas pred1_ft to pred4_ft* are four significant fault tolerant properties that GFTSA can preserve, which can be mechanically verified by the theorem prover of PVS. The mechanical verification of these properties will be illustrated in the next section.

6.3 Mechanical Verification of GFTSA using PVS

Since GFTSA is used to help develop safety critical distributed systems, the verification of GFTSA mainly involves showing that GFTSA can preserve fault tolerant properties, which are expressed as *LEMMA*. For each *LEMMA*, we build a proof tree by inputting proof commands until each branch of the tree is proved to be true. In the following, several significant fault tolerant properties and their proof scripts are presented to illustrate the mechanical verification of GFTSA by using the theorem prover of PVS.

6.3.1 A Global Exception raised in a Fault-tolerant Component

When a global exception is raised by an *Object* in the *FTS*, all of the *Objects* in the *FTS* should be informed about the exception. During the proof of *pred1_ft* property, which is firstly shown in the consequent {1}, we can use primitive proof commands in response to the *Rule?* prompt from PVS theorem prover to prove this property interactively. These primitive proof commands could be *flatten*, *prop*, and *assert* etc, which can help the proofs more automatically and systematically. The proof script displayed below can show the basic features of the theorem prover of PVS.

```
pred1_ft :
  |-------
{1}   (EXISTS (obj: Object):
         member(inter_state(obj), g_excepts) AND ue_rec(obj) = 0)
       IMPLIES
       (FORALL (obj: Object), (ccp: CC):
          inter_state(UniExceptReceive(obj, ccp)) =
           except_graph(exceptions(ExceptRec(ccp))))
Rule?: (flatten) Applying disjunctive simplification to flatten
                 sequent, this simplifies to:
pred1_ft :
{-1}  (EXISTS(obj: Object):
         member(inter_state(obj), g_excepts) AND ue_rec(obj) = 0)
  |-------
{1}   (FORALL (obj: Object), (ccp: CC):
         inter_state(UniExceptReceive(obj, ccp)) =
          except_graph(exceptions(ExceptRec(ccp))))
```

```
......

Rule?: (lemma "Propagate") Applying Propagate this

simplifies to:

pred1_ft :

{-1}  FORALL (obj: Object):

     member(inter_state(obj), g_excepts) AND ue_rec(obj) = 0 IMPLIES

         GlobalExceptPropagate(obj) = inter_state(obj)

[-2]  member(inter_state(obj!1), g_excepts) AND ue_rec(obj!1) = 0

  |-------

[1]   (FORALL (obj: Object), (ccp: CC):

         inter_state(UniExceptReceive(obj, ccp)) =

          except_graph(exceptions(ExceptRec(ccp))))

 .......

Rule? : (prop) Applying propositional simplification, which is

trivially true. Q.E.D.
```

The *(flatten)* command eliminates the disjunctive connectives in the consequent {1} of *pred1_ft* so as to flatten it out into the sequent. The next proof command *(skolem!)* is used to replace the existentially quantified variable *obj* in the antecedent {-1} with constant *obj!1.* The following proof step *(lemma "Propagate")* is used to bring in an instance of the *"Propagate"* as an antecedent sequent formula. By prompting these PVS primitive proof commands, we can move on to complete the verification of this property successfully.

6.3.2 Two Global Exceptions raised Concurrently in Fault-tolerant Components

When two global exceptions are raised concurrently by two different *Objects* in the *FTS*, all the *Objects* in the *FTS* need to be informed about a universal global exception. This *pred2_ft* property is firstly shown as the consequent {1} in the theorem prover of PVS.

The proof script for this property starts with the application of *(flatten)* to the given conjecture followed by *(skolem!)* command to replace the existentially

```
pred2_ft :
  |-------
{1}   (EXISTS (obj1, obj2: Object):
         member(inter_state(obj1), g_excepts) AND
          ue_rec(obj1) = 0 AND
           member(inter_state(obj2), g_excepts) AND ue_rec(obj2) = 0)
       IMPLIES
       (FORALL (obj: Object), (ccp: CC):
          ue_rec(UniExceptReceive(obj, ccp)) = 1)
Rule?: (flatten) Applying disjunctive simplification to flatten
 .......
Rule?: (skolem!) Skolemizing, this simplifies to:
pred2_ft :
{-1}member(inter_state(obj1!1), g_excepts) AND
```

```
      ue_rec(obj1!1) = 0 AND

       member(inter_state(obj2!1), g_excepts) AND ue_rec(obj2!1) = 0

  |-------

[1]   (FORALL (obj: Object), (ccp: CC):

        ue_rec(UniExceptReceive(obj, ccp)) = 1)

 .......

Rule?: (lemma "ExceptPropagate") Applying ExceptPropagate this

simplifies to:

pred2_ft :

{-1}  FORALL (obj: Object): exception =GlobalExceptPropagate(obj)

[-2]  GlobalExceptPropagate(obj2!1) =inter_state(obj2!1) [-3]

GlobalExceptPropagate(obj1!1) =inter_state(obj1!1)

  |-------

[1]   (FORALL (obj: Object), (ccp: CC):

        ue_rec(UniExceptReceive(obj, ccp)) = 1)

Rule?: (instantiate -1 ("obj1!1"))

 .......

Rule? : (prop) Applying propositional simplification, which is

trivially true. Q.E.D.
```

quantified variable. The following strategy in the proof script is to bring in the declared *LEMMAS* to make the consequent in the sequent to be true. These *LEMMAS* involve *“Propagate”, ”ExceptPropagate”, “NonEmpty”, “CCReceive”, “ExceptGraph”, and “UniExcept”*, which have already been proved to be true. The strategy of these *LEMMAS* can be described as follows. When two global exceptions are raised concurrently in the *FTS*, each *Object* can use *GlobalExceptPropagate* function to *Propagate* its global exception to the *CoordinatingComponent*, and the *CoordinatingComponent* can use the *ExceptRec* function to receive these two global exceptions. Because the record *exceptions* is not empty, the *ExceptGraph* function can use the function *except_graph* to get the *uni_exception*. After that, the *uni_exception* is sent out to all the *Objects* in the *FTS*. The *UEReceive* function in the *Object* can receive this *uni_exception*. When an *Object* receives the *uni_exception*, the value of *ue_rec* is changed to 1, which means that the *Object* is informed about the exceptions.

6.3.3 A Local Exception raised in a Fault-tolerant Component

If an *Object* in the *FTS* raises a local exception, the other *Objects* are not influenced and convince to be in their stable states. This *pred3_ft* property is firstly shown as the consequent {1} in the theorem prover of PVS.

```
pred3_ft :
  |-------
{1}   (EXISTS (obj: Object):
          member(inter_state(obj), l_excepts) AND
           Variable(other_obj!1, obj))
        IMPLIES member(inter_state(other_obj!1), n_states)
```

```
Rule?: (flatten) Applying disjunctive simplification to flatten

 ........

Rule?: (lemma "NonEquVar1") Applying NonEquVar1 this simplifies

 .......

Rule?: (lemma "NonEquVar2") Applying NonEquVar2 this simplifies

to:

pred3_ft :

{-1}  FORALL (obj1, obj2: Object):

        obj1 /= obj2 IMPLIES inter_state(obj1) /= inter_state(obj2)

[-2]  other_obj!1 /= obj!1 [-3]

member(inter_state(obj!1),l_excepts)

  |-------

[1]   member(inter_state(other_obj!1), n_states)

 ........

Rule?: (assert) Simplifying, rewriting, and recording with

decision procedures, Q.E.D.
```

The proof script for this property starts with the application of *(flatten)* to the given conjecture followed by *(skolem!)* command to replace the existentially quantified variable. After that, the *LEMMAS "NonEquVar1", "NonEquVar2", "LocalExcept1", and "LocalExcept2"* are brought in to the antecedent of the sequent. The proof strategy of these *LEMMAS* can be described as follows. When a local exception is raised in an *Object*, the other *Objects* will have different state from this *Object*. Because the raised exception is a local exception, no exception will be sent out to the *CoordinatingComponent*. Therefore, the states of other *Objects* will still be normal.

6.3.4 Fault-tolerant System recover From non-critical Fault-tolerant component Failure

When a non-critical *Object* fails, the *FTS* can tolerate this fault, which means that the states of all *Objects* in the *FTS* can recover to their normal states. This *pred4_ft* property is firstly shown as the consequent {1} in the theorem prover of PVS.

The proof script for this property starts with the application of *(flatten)* to the given conjecture followed by *(skolem!)* command to replace the existentially quantified variable. The following proof steps mainly bring in *"syspred2"*, and

```
pred4_ft :
  |-------
{1}(FORALL (fobs: setof[OBID]): disjoint?(fobs, critical)) IMPLIES
       (FORALL (obj: Object), (fobs: setof[OBID]):
          systemrecover(obj, fobs) = Init)
Rule?: (flatten) Applying disjunctive simplification to flatten
                  sequent, this simplifies to:
pred4_ft :
{-1}  (FORALL (fobs:setof[OBID]):disjoint?(fobs,critical))
  |-------
```

```
{1}   (FORALL (obj: Object), (fobs: setof[OBID]):
         systemrecover(obj, fobs) = Init)
Rule?: (skolem!) Skolemizing, this simplifies to:
 .........
Rule?: (lemma "syspred2") Applying syspred2 this simplifies to:
pred4_ft : {-1}  FORALL (fobs: setof[OBID]):
   disjoint?(fobs, critical) IMPLIES Result_Control(fobs) = tolerate
[-2]  disjoint?(fobs!1, critical)
  |-------
[1]   systemrecover(obj!1, fobs!1) = Init
Rule?: (instantiate -1("fobs!1"))
 ........
Rule?: (lemma "Recover") Applying Recover this simplifies to:
pred4_ft :
 {-1}  FORALL (fobs: setof[OBID], obj: Object):
        Result_Control(fobs) = tolerate IMPLIES
         systemrecover(obj, fobs) = Init
 .......
Rule?: (prop) Applying propositional simplification, which is
trivially true.Q.E.D.
```

"Recover" to make the consequent in the sequent to be true. The proof strategy of these *LEMMAS* can be described as follows. When the state of an *Object* is *Fail*, if this *Object* is not in the state *critical*, we can gain the execution result, namely

tolerate, by using function *Result_Control*. Because the execution result is *tolerate*, the *SystemRecover* function is used to reset the states of all *Objects* to the initial states, which means that all the *Objects* recover to normal states.

6.4 Template based on PVS Model of GFTSA

GFTSA is proposed to guide the development of safety critical distributed systems. The developed systems guided by GFTSA can preserve fault tolerant properties. By using the theorem prover of PVS, we can mechanically verify the fault tolerant properties. As the theorem prover of PVS only supports the model in PVS specification language, we need to generate the PVS models of developed systems. Based on the Object-Z model of GFTSA, we have built a template to make the generation of Object-Z models of distributed systems more efficient by using the XVCL technique. Following the similar XVCL technique, we investigate to build a template to help the generation of PVS models of developed systems based on the PVS model of GFTSA.

This template is built as generic and adaptable x-frames, which are all written as the combination of PVS specification and XVCL commands. In each primitive x-frame, besides importing corresponding theories in the PVS model of GFTSA, all of small or large variation points are represented as meta-expressions, which can be instantiated during the customization process according to the specific requirements. In the following, we will present these primitive x-frames involved in the template, which are *constants*, *connector*, *cc*, *sr*, *object*, and *system*.

6.4.1 The x-frame for global constants

The *constants* x-frame is built for the global constants which can be used in the declared formulas of other theories. This theory imports the *generictype* theory in the PVS model of GFTSA.

```
<x-frame name="constants" language="PVS">
constants: THEORY
  BEGIN
       IMPORTING generictype
   <value-of expr="?@ports?"/> <value-of expr="?@gobstates?"/>
   <value-of expr="?@msgs"/>
 END constants
</x-frame>
```

In this x-frame, the *ports* represents the *in_port* and *out_port* used to transmit the messages among *Objects*. The *gobstates* represents the states of *Object*. The *msgs* represents the messages transmitted among *Objects* and *SharedResource*.

6.4.2 The x-frame for connector

The *connector* x-frame is built for the *connector* theories of specific safety critical distributed systems. This theory imports the *connector* theory in the PVS model of GFTSA. The *constants* theory also is imported.

```
<x-frame name="connector" language="PVS">
<value-of expr="?@connectorname?"/>: THEORY
 BEGIN
   IMPORTING connector,constants
  send_port: PORT=<value-of expr="?@s_port?"/>
  receive_port: PORT=<value-of expr="?@r_port?"/> END
 <value-of expr="?@connectorname?"/>
</x-frame>
```

In this x-frame, the *s_port* represents the sending port of the *Connector*, and the *r_port* represents the receiving port of the *Connector*. The *connectorname* represents the *Connector* name of the specific system.

6.4.3 The x-frame for coordinatingcomponent

The *cc* x-frame is built for the *coordinatingcomponent* theory of specific safety critical distributed system. This theory imports the *coordinatingcomponent* theory in the PVS model of GFTSA. The *constants* theory also is imported.

```
<x-frame name="cc" language="PVS">
<value-of expr="?@ccname?"/>: THEORY
 BEGIN
  IMPORTING coordinatingcomponent,constants
```

```
  ge: VAR items[OBSTATE]

  except: OBSTATE

  except_graph(ge): OBSTATE= <break name="egraph"/>

  END

 <value-of expr="?@ccname?"/>

</x-frame>
```

In this x-frame, the *egraph* variable in the ⟨ break⟩ represents a function, which is used to resolve the concurrently raised exceptions into a universal exception. The *ccname* represents the *CoordinatingComponent* name of the developed system.

6.4.4 The x-frame for sharedresource

The *sr* x-frame can be adapted to build the *sharedresource* theory of specific safety critical distributed system. This theory imports the *sharedresource* theory in the PVS model of GFTSA. The *constants* theory is also imported.

```
 <x-frame name="sr" language="PVS">

 <value-of expr="?@srname?"/>: THEORY

  BEGIN

    IMPORTING sharedresource,constants

   <value-of expr="?@srstates?"/>: SRSTATE

   states: setof[SRSTATE]=

     {x:SRSTATE|<value-of expr="?@srstatedeclare?"/>

   <break name="srtransaxiom"/>
```

```
   END
 <value-of expr="?@srname?"/>
</x-frame>
```

In this x-frame, the *srstates* represents the states that the *SharedResource* can be in. The *srstatedeclare* represents the declaration of the states in the *srstates*. The *srtransaxiom* represents the *AXIOM* declaration for the state transitions. The *srname* represents the *SharedResource* name of specific system.

6.4.5 The x-frame for object

The *object* x-frame can be adapted to build the *Objects* theories of specific safety critical distributed systems. This theory imports the *object* theory in the PVS model of GFTSA. The *constants* theory is also imported.

```
 <x-frame name="object" language="PVS">
 <value-of expr="?@objectname?"/>:THEORY
   BEGIN
     IMPORTING object, states
    Object: TYPE=[#inter_state:OBSTATE,checkpoint:
    OBSTATE,ue_rec:SIG,sr_qlist:(nonemptyqueue?[SRID])#]
   <value-of expr="?@excepthandlenames?"/>: TYPE=EH
   state: VAR OBSTATE
   <value-of expr="?@sobstates?"/>: OBSTATE
   <break name="excepthandledeclare"/>
```

```
 n_states: setof[OBSTATE]=

     {x:OBSTATE|<value-of expr="?@nstatesdeclare?"/>}

 <ifdef var="lexceptsdeclare">

 l_excepts: setof[OBSTATE]=

     {x:OBSTATE|<value-of expr="?@lexceptsdeclare?"/>}

 </ifdef>

 g_excepts: setof[OBSTATE]=

     {x:OBSTATE|<value-of expr="?@gexceptsdeclare?"/>}

 <ifdef var="inportsdeclare">

 in_ports: setof[PORT]=

     {x:PORT|<value-of expr="?@inportsdeclare?"/>}

 </ifdef>

 <ifdef var="outportsdeclare">

 out_ports: setof[PORT]=

     {x:PORT|<value-of expr="?@outportsdeclare?"/>}

 </ifdef>

 <ifdef var="compmsgsdeclare">

 comp_msgs: setof[MSG]=

     {x:MSG|<value-of expr="?@compmsgsdeclare?"/>}

 </ifdef>

<break name="coopmsgaxiom"/>

<break name="transitionaxiom"/>

<break name="exceptcontextaxiom"/>

<break name="excepthandleaxiom"/>
```

```
  obinistate: OBSTATE=<value-of expr="?@inistate?"/> \\
 END <value-of expr="?@objectname?"/>
</x-frame>
```

In this primitive x-frame, the *exceptionhandlenames* represents the exception handler function used in the *Object*. The *sobstates* represents the normal states that the *Object* can be in. The *nstatesdeclare* represents the declaration of normal states of *Object*. The *lexceptdeclare* and *gexceptdeclare* represent the local exceptional and global exceptional states that the *Object* can be in. The *inportsdeclare, outportsdeclare, compmsgsdeclare, and coopmsgsdeclare* represent the declaration of *in_port, out_port*, competitive messages and cooperative messages. The *transitionaxiom, exceptioncontextaxiom, excepthandleaxiom* represent the *AXIOM* declarations about the state transition, exception context function, and except handle function correspondingly.

6.4.6 The x-frame for ftsystem

The *system* can be adapted to build the *ftsystem* theory of specific safety critical distributed system. This theory imports the built theories for the components & connectors involved in the specific system.

```
<x-frame name="ftsystem" language="PVS">
 <value-of expr="?@systemname?"/>:THEORY
  BEGIN
    IMPORTING: <value-of expr="?@componentnames?"/>
```

```
   ccp: VAR CC
   <break name="lemmas"/>
   <break name="predicates"/> END
 <value-of expr="?@systemname?"/>
</x-frame>
```

In this x-frame, the *componentnames* represents the theories which need to be imported in the *ftsystem* theory. The *lemmas* and *predicates* represents the transition and fault tolerant properties of specific system, which are declared as *LEMMAS*.

By instantiating the variables defined in these x-frames according to specific requirements, we can generate the PVS model of developed system guided by GFTSA. In the next Chapter of Mechanical Verification of developed Safe Critical Distributed Systems guided by GFTSA, we will present case studies to illustrate the customization from such built template.

6.5 Conclusion

In this chapter, we embed the GFTSA model in PVS to achieve mechanical verification support for reasoning about fault tolerant properties. The component & connectors of GFTSA all are represented as *theories*, which constitute a *theory* chain by importing. Based on the PVS model of GFTSA, we can mechanically verify the fault tolerant properties of GFTSA by virtue of the theorem prover of PVS. When we verify a property in PVS, firstly we formalize this property as a *LEMMA*, then we can input primitive proof commands to the theorem prover interactively to verify this *LEMMA* until the proof result is *true*. The mechanical verification of GFTSA can guarantee that GFTSA can preserve fault tolerant properties.

Since GFTSA is proposed to guide the development of safety critical distributed systems, we investigate to build a template based on the PVS model of GFTSA. This template can be used to help the generation of PVS models of developed safety critical distributed systems guided by GFTSA.

Chapter 7

Mechanical Verification of developed Safety Critical Distributed Systems guided by GFTSA

This chapter presents a template for the specification and proof scripts of developed safety critical distributed systems guided by GFTSA, and two case studies to illustrate the mechanical verification of safety critical distributed systems.

7.1 Introduction

The Generic Fault Tolerant Software Architecture (GFTSA) is proposed to guide the development of safety critical distributed systems. In order to achieve mechanical verification support for reasoning about the fault tolerant properties of GFTSA, we have embedded the GFTSA model in the PVS theorem prover. Based on the PVS model of GFTSA, we can mechanically verify the fault tolerant properties of GFTSA by using the theorem prover of PVS. Since the developed safety critical distributed systems guided by GFTSA need to preserve the fault tolerant properties, the theorem prover of PVS also can help such verification. As the theorem prover of PVS only supports the models in the PVS specification language, we need to get the PVS models of developed systems. A template has been built based on the PVS model of GFTSA to generate the PVS models of developed systems. In this chapter, we present a case study of Line Direction Agreement System (LDAS) to illustrate how we can generate the PVS model of LDAS from the built template via customization, and the mechanical verification of fault tolerant properties of LDAS by using the theorem prover of PVS. During the mechanical verification of LDAS, the theorem prover of PVS needs to be applied primitive proof commands interactively under user guidance. In order to make such verification more efficient, we investigate to use the batch mode of PVS in the verification.

The primitive proof commands input by user to verify one specific property can constitute the proof script for this property. In the batch mode of PVS, we can apply the proof script directly to the theorem prover of PVS to verify one specific property, which does not require inputting each primitive proof command interactively. By customizing the generic proof scripts, we can generate the proof scripts for the developed safety critical distributed systems, and apply them to the theorem prover to verify the fault tolerant properties of developed systems in batch mode. The ProofLite [53] technique can provide user-friendly interface of batch mode execution and interactive proof scripting notation to the system designers. As the proof scripting notation supported by ProofLite enables a semi-literate proving style where specification and proof scripts reside in the same context, we investigate to extend our built template to involve generic fault tolerant properties accompanying with generic proof scripts by using XVCL and Prooflite techniques. By customizing this template, we can generate both PVS models, and proof scripts for the developed systems. A case study of an Electric Power System (EPS) [14] is presented to illustrate the customization process and mechanical verification.

The remainder of the chapter is organized as follows. In section 2, we present a case study of LDAS to illustrate how we can generate the PVS model of LDAS from the template based on the PVS model of GFTSA, and mechanically verify the fault tolerant properties of LDAS by using the theorem prover of PVS. Section 3 presents the extension of the template based on PVS model of GFTSA, which involves not only PVS specification, but also generic proof scripts for the fault tolerant properties. Section 4 presents a case study of Electronic Power System (EPS) to demonstrate that we can generate the specification and proof scripts for fault tolerant properties of specific system by customizing the built template.

Section 5 concludes the paper.

7.2 Case Study-LDAS (Line Direction Agreement System)

In the Section 6.4, we have built a template based on the PVS model of GFTSA. In this section, we use a case study of LDAS to demonstrate how the PVS model of LDAS can be generated from this template. Based on the generated PVS model of LDAS, we can mechanically verify the fault tolerant properties of LDAS.

7.2.1 Line Direction Agreement System(LDAS)

The Line Direction Agreement System (LDAS) [17], a safety critical distributed system, is designed to control the line direction to prevent the head-on train crashes on the line. Each station communicate with *LDACS* (Line Direction Agreement Control System) to guarantee that, at a time, only one train runs on the line connecting two stations. The operator in each station can command the station. Figure 7.1 shows a part of LDAS, composed of three stations. The overall LDAS can be much more complex comprising of several stations, and communication paths.

In the system under consideration, *StatonA* and *StationB* communicate with *LDACS* to control the direction of *LineAB*. Similarly, *StationB* and *StationC* communicate

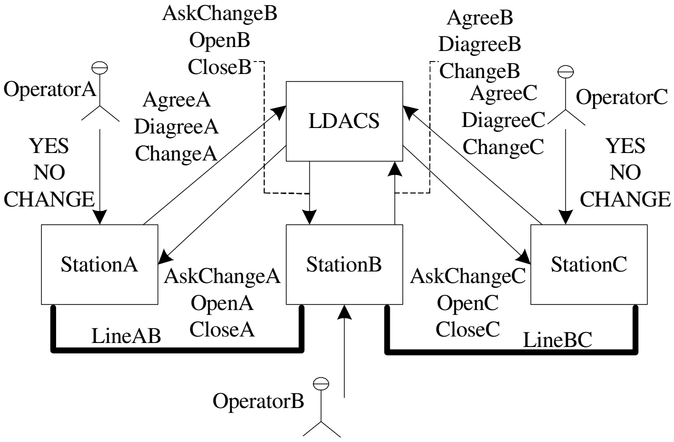

Figure 7.1: The LDAS System.

with *LDACS* to control *LineBC*. We can consider *StationA*, *OperatorA, LDACS, StationB, and OperatorB* to be a relatively independent sub-system that can be analyzed in isolation, as it has the requirement for both cooperative & competitive concurrency, even in isolation. The interaction pattern in the other sub-systems, e.g., the system comprising of *StationB* and *StationC* follows the same regulation as the above mentioned case.

According to the box-and-line patterns of GFTSA shown in Figure 3.1, the LDAS sub-system is composed of five *Objects*, called *StationA*, *OperatorA*, *LDACS*, *StationB*, *OperatorB*, and a *SharedResource*, called *LineAB*. Six connectors are used to assist the communication among the *Objects*. A *CoordinatingComponent* called *CC* is also involved in the LDAS to implement the *coordinated error recovery mechanism*.

7.2.2 The Generation of LDAS Formal Model

In order to mechanically verify that LDAS can satisfy the high safety requirements by using the theorem prover of PVS, we need to generate the PVS model of LDAS from the built template, presented in the Section 6.4. The six primitive x-frames in the template of GFTSA can be reused during the customization via adaptation. Following the mechanisms of XVCL, the adaptation means that a new x-frame for one component in the specific system is built based on the corresponding primitive x-frame in the template by using XVCL command ⟨ adapt ⟩ and instantiating the variation points.

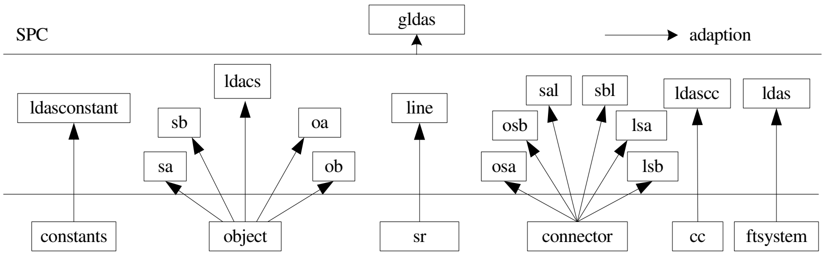

Figure 7.2: The x-frame Adaption Relationship of LDAS.

We can build x-frames for the PVS model of LDAS based on the primitive x-frames of template. Figure 7.2 describes x-frame adaptation relationship between the LDAS, and the template. The *ldasconstant* is built to declare the global constants which will be used by other theories involved in the PVS model of LDAS. The *sa*, *sb*, *ldacs*, *oa* and *ob* x-frames are built for *StationA*, *StationB*, *LDACS*, *OperatorA* and *OperatorB* correspondingly. The *osa*, *sal* , *lsa*, *lsb*, *sbl*, and *osb* x-frames are built for six connectors in the LDAS. The *ldascc* x-frame is built for *CC* component and the *line* x-frame is built for *LineAB* component. The *ldas* x-frame is built to describe how these components and connectors synchronize. By running XVCL processor with gsa SPC file which adapts all of the 15 x-frames of LDAS, we can generate PVS model of LDAS automatically. Figure 7.3 shows the model design of LDAS in the box-and-line fashion guided by the pattern of GFTSA.

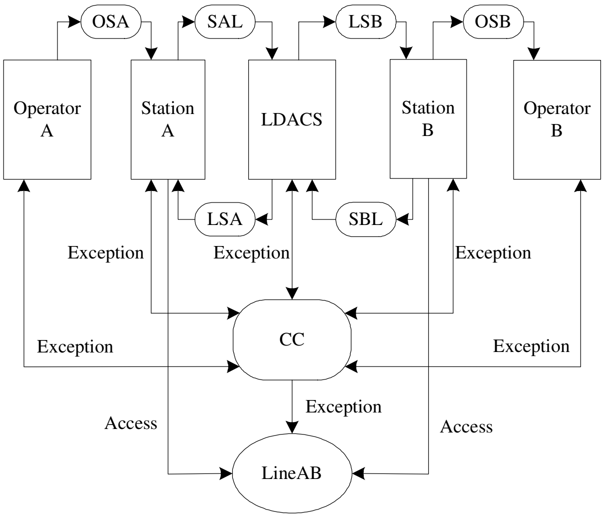

Figure 7.3: GFTSA architecture view of LDAS sub-system.

Several representative theories is presented to illustrate the features of PVS model of LDAS.

The sa theory

The *sa* theory represents the *Object* component in the LDAS which describes how the *StationA* interacts with other *Object* & *SharedResource*, and how to deal with local & global exceptions.

```
sa:THEORY

   BEGIN

    IMPORTING  object, ldasconstant

     Object: TYPE=[#inter_state:OBSTATE,checkpoint: OBSTATE,

       ue_rec:SIG,sr_qlist:(nonemptyqueue?[SRID])#]

     forward_recovery: TYPE=EH

     state: VAR OBSTATE

     APointClosed,AskedOpen,AskedClose,APointOpened: OBSTATE

     forward_recovery(state): OBSTATE=APointClosed

     n_states: setof[OBSTATE]={x:OBSTATE|x=APointClosed OR

       x=AskedOpen OR x=AskedClose OR x=APointOpened}

     l_excepts:setof[OBSTATE]={x:OBSTATE|x=input_exception}

     g_excepts: setof[OBSTATE]={x:OBSTATE|x=bothopen_exception

      OR x=bothclose_exception}

     in_ports: setof[PORT]={x:PORT|x=LSA_In OR x=OSA_In}

     out_ports: setof[PORT]={x:PORT|x=SAL_Out}

     comp_msgs: setof[MSG]={x:MSG|x=APointOpen OR x=APointClose}

     coop_msg1: AXIOM coop_msg(LSA_In)=AskChangeA

     ......

     coop_msg9: AXIOM coop_msg(OSA_In)=CHANGE

     transition1: AXIOM

      transition(APointClosed,OSA_In,CHANGE)=(AskedOpen,SAL_Out,ChangeA)

     .......

     transition12:AXIOM
```

```
     transition(APointOpened,LSA_In,CloseA)=(input_exception,none,NONE)
    except_context1: AXIOM
     except_context(bothopen_exception)=forward_recovery
    except_context2: AXIOM
     except_context(bothclose_exception)=forward_recovery
    except_handle: AXIOM
     except_handle(forward_recovery)=APointOpened
    obinistate: OBSTATE=APointClosed
  END stationA
```

The *sa* theory imports *object* and *ldasconstant* theory. A *forward_recovery* function is defined to handle exceptions. The *APointClose*, *AskedOpen*, *AskedClose* and *AskedOpen* are four normal states. The *input_exception*, *bothopen_exception* and *output_exception* are three exceptional states. The local exception *input_exception* declared in *l_excepts* represents that the *StationA* cannot handle the received messages. The global exception *bothopen_exception* represents that both *StationA* and *StationB* can open the gates. The global exception *bothclose_exception* represents that both *StationA* and *StationB* can close the gates. The messages associated with the port of *Object* are declared in the *AXIOM* of *coop_msg*. When receiving messages from other *Objects*, the state of *StationA* could be transformed from one normal state to either another normal state or an exceptional state, which are declared in the *AXIOM* declaration of *transition*. We use the *forward_recovery* to handle *bothopen_exception* and *bothclose_exception*, which are declared in the *AXIOM* of except_context. When we use the *forwar_recovery* to handle the exceptions, the state of *StationA* will be recovered to the *APointOpened* state, which declared in the *AXIOM* of *except_handle*.

The ldas Theory

How the component & connectors in the LDAS cooperate with each other is described in the *ldas* theory.

```
ldas:THEORY
  BEGIN
    IMPORTING operatorA,operatorB,stationA,stationB,ldacs,
              osa,sal,lsa,lsb,sbl,osb,coco,line
   ccp: VAR CC
  stationA_prop: LEMMA
    FORALL (obj: stationA.Object):
     member(inter_state(obj),stationA.g_excepts) AND ue_rec(obj)=0
      IMPLIES
       GlobalExceptPropagate(obj)=inter_state(obj)
  .......
  ldacs_UniExceptRec2:LEMMA
    FORALL (obj: ldacs.Object),(ccp:CC):
     ldacs.uni_exception=coco.except_graph(exceptions(ExceptRec(ccp)))
       IMPLIES
     inter_state(UniExceptReceive(obj,ccp))=ldacs.uni_exception AND
          ue_rec(UniExceptReceive(obj,ccp))=1
```

```
.......
ldas_ft1: LEMMA
  (EXISTS (obj: stationA.Object):
   inter_state(obj)=bothopen_exception AND ue_rec(obj)=0)
    IMPLIES
    (FORALL (obj: ldacs.Object):
      member(inter_state(UniExceptHandle(obj)),ldacs.n_states))
 ldas_ft2: LEMMA
 (EXISTS (obj1: stationA.Object),(obj2:stationB.Object):
   inter_state(obj1)=bothopen_exception AND ue_rec(obj1)=0 AND
    inter_state(obj2)=bothclose_exception AND ue_rec(obj2)=0) IMPLIES
      (FORALL (obj: ldacs.Object):
        member(inter_state(UniExceptHandle(obj)),ldacs.n_states))
END ldas
```

The *ldas* theory imports all the *theories* for the components & connectors of LDAS. Several significant properties of LDAS are declared as *LEMMA*. Several *LEMMAS* are shown in the *ldas* theory as example. The *StationA_prop LEMMA* represents that when raising a global exception, the *StationA* needs to use *GlobalExceptPropagate* operation to send this exception to the *CoordinatingComponent*. The *ldacs_UniExceptRec2 LEMMA* represents that when the *LDACS* receives the resolved *uni_exception* from the *CoordinatingComponent*, the *inter_state* and *ue_rec* of *LDACS* will be changed to *uni_exception* and *1*. The *ldas_ft1 and ldas_ft2* are two signification fault tolerant properties that LDAS can preserve. By using the theorem prover of PVS, we can mechanically verify these properties successfully.

7.2.3 Mechanical Verification of LDAS

Based on the generated PVS model of LDAS, we can mechanically verify the fault tolerant properties of LDAS by using the theorem prover of PVS. Two significant fault tolerant properties *ldas_ft1* and *ldas_ft2* are presented as *LEMMA* in the *ldas* theory. The proof scripts for these two properties are presented in the following.

Facing bothopen Exception

When the *bothopen_exception* is raised in the *StationA*, the LDAS can tolerate this exception which means that any other *Object*, such as *LDACS*, can handle this exception and recover to normal execution process. This *ldas_ft1* property is firstly shown as the consequent {1} in the theorem prover.

```
ldas_ft1 :
  |-------
{1}   (EXISTS (obj: stationA.Object):
          inter_state(obj) = bothopen_exception AND ue_rec(obj) = 0)
        IMPLIES
        (FORALL (obj: ldacs.Object):
           member(inter_state(UniExceptHandle(obj)), ldacs.n_states))
Rule?: (flatten) Applying disjunctive simplification to flatten
 sequent, this simplifies to:
```

```
ldas_ft1 :

{-1} (EXISTS(obj: stationA.Object):

         inter_state(obj) = bothopen_exception AND ue_rec(obj) = 0)

  |-------

{1}   (FORALL (obj: ldacs.Object):

         member(inter_state(UniExceptHandle(obj)), ldacs.n_states))

Rule?: (skolem!) Skolemizing, this simplifies to:

ldas_ft1 :

{-1}  inter_state(obj!1) = bothopen_exception AND ue_rec(obj!1) =0

  |-------

[1]   (FORALL (obj: ldacs.Object):

         member(inter_state(UniExceptHandle(obj)), ldacs.n_states))

Rule?: (lemma "stationA_member") Applying member this simplifies

  .......

Rule? (instantiate -1 ("obj!1" "ccp!1"))

 Instantiating the top quantifier in -1 with the terms:

 (obj!1 ccp!1), this simplifies to:

ldas_ft1 :

{-1} inter_state(UniExceptReceive(obj!1, ccp!1))

       =ldacs.uni_exception AND

       ue_rec(UniExceptReceive(obj!1, ccp!1)) = 1

       IMPLIES

       inter_state(obj!1) = ldacs.uni_exception AND ue_rec(obj!1) = 1

[-2]  inter_state(UniExceptReceive(obj!1, ccp!1)) =
```

```
 ldacs.uni_exception [-3]  ue_rec(UniExceptReceive(obj!1, ccp!1)) =1
[-4]  ldacs.uni_exception = bothopen_exception
  |-------
[1]   (FORALL (obj: ldacs.Object):
         member(inter_state(UniExceptHandle(obj)), ldacs.n_states))
Rule? (assert) Simplifying, rewriting, and recording with decision
procedures, Q.E.D. Run time  = 4.00 secs. Real time = 178.88 secs.
```

The proof script for this property starts with the application of *(flatten)* to the given conjecture followed by *(skolem)* command to replace the existentially quantified variable. After that, the LEMMAS *"stationA_member", "stationA_except", "open_exceptRec", "open_exceptGraph1", "open_exceptGraph2", "ldacs_UniExcept Rec1", "ldacs_UniExceptRec2", "ldacs_stateChange"* are brought into the antecedent of the sequent. The proof strategy of these *LEMMAS* can be described as follows. When a global exception called *bothopen_exception* is raised in the *StationA*, the *StationA* can use *GlobalExceptPropagate* operation to send this global exception out to *CC* and *CC* can use *ExceptRec* operation to receive this global exception. Since the sequence *exceptions* is not empty, the *ExceptGraph* operation in the *CC* class sends out the *uni_exception!*. The *UEReceive* operation in each *Object* of LDAS can receive this *uni_exception?*. When each *Object* of LDAS receives the *uni_exception?*, the state is changed to normal state. These transformations assure that *Objects* in the LDAS can handle the global exception.

Facing bothopen and bothclose Exceptions Raised Concurrently

When the *bothopen_exception* has been raised in the *StationA* and the *bothclose_exception* has been raised in the *StationB* concurrently, the LDAS can handle this situation which means that each *Object* such as *LDACS* can be recovered to normal execution process. This *ldas_ft2* property is firstly shown as the consequent {1} in the theorem prover of PVS.

```
ldas_ft2 :
  |-------
{1}   (EXISTS (obj1: stationA.Object), (obj2: stationB.Object):
         inter_state(obj1) = bothopen_exception AND
          ue_rec(obj1) = 0 AND
           inter_state(obj2) = bothclose_exception AND ue_rec(obj2) = 0)
       IMPLIES
       (FORALL (obj: ldacs.Object):
          member(inter_state(UniExceptHandle(obj)), ldacs.n_states))
Rule?: (flatten) Applying disjunctive simplification to flatten
sequent, this simplifies to:
 ......
Rule?: (lemma "stationA_except") Applying stationA_except this
 simplifies to:
ldas_ft2 :
{-1}  FORALL (obj: stationA.Object): exception =
    GlobalExceptPropagate(obj)
```

```
[-2] GlobalExceptPropagate(obj1!1) = bothopen_exception [-3]

inter_state(obj2!1) = bothclose_exception [-4]  ue_rec(obj2!1)= 0

.....

ldas_ft2 :

{-1}  member(inter_state(obj!1), ldacs.g_excepts)AND

       member(except_handle(except_context(inter_state(obj!1))),

              ldacs.n_states)

       IMPLIES

       inter_state(UniExceptHandle(obj!1)) =

        except_handle(except_context(inter_state(obj!1)))

[-2]  member(except_handle(except_context(inter_state(obj!1))),

             ldacs.n_states)

[-3]  except_handle(except_context(inter_state(obj!1))) = LockedAB

[-4]  member(inter_state(obj!1), ldacs.g_excepts)

  |-------

[1]   member(inter_state(UniExceptHandle(obj!1)), ldacs.n_states)

Rerunning step: (assert) Simplifying, rewriting, and recording

with decision procedures, Q.E.D.

 Run time  = 0.38 secs. Real time = 0.98 secs.
```

The proof script for this property starts with the application of *(flatten)* to the given conjecture followed by *(skolem!)* command to replace the existentially quantified variable. After that, the *LEMMAS "stationA_member", "stationA_prop", "stationA_except", "open_exceptRec", "open_exceptGraph1", "open_exceptGraph2",* *"ldacs_UniExceptRec1", "ldacs_UniExceptRec2", "ldacs_stateChange", "ldacs_member1", "ldacs_UniExceptHandle", "ldacs_member2", "ldacs_UniExceptHandle2"* are brought in to the antecedent of the sequent. The proof strategy of these *LEMMAS* can be described as follows. When the *bothopen_exception* raised in *StationA* and the *bothclose_exception* raised in *StationB* concurrently, each of them can use *GlobalExceptPropagate* operation to send the exception out to the *CC* and the *CC* can execute the *ExceptRec* operation to receive these two global exceptions. Because the sequence *exceptions* is not empty, the *ExceptGraph* operation in the *CC* class schema can send out the *uni_exception!* which covers *bothopen_exception* and *bothclose_exception*. The *UEReceive* operation in the *LDACS* can receive this *uni_exception?*. When the *LDACS* in the LDAS receives the *uni_exception?*, its state is changed to normal state.

7.3 Template based on PVS model of GFTSA and Proof Scripts

The case study of LDAS can demonstrate that the developed LDAS guided by GFTSA can preserve the fault tolerant properties by using the theorem prover of PVS. When considering the fault tolerant properties of distributed systems with high reliability requirements, for example, SCS and LDAS, we can summarize these properties as the generic ones that the systems can deal with a global exception or multiple raised global exceptions. These generic properties can be customized according to specific system. When mechanically verifying these properties, we interactively apply primitive proof commands to the theorem prover of PVS. These primitive commands mainly involve *"lemma name"* preceded or followed by *"skolem!"*, *"instantiate"*, *"replace"*, and *"assert"* commands. The *"lemma name"* introduces an instance of the lemma named *name* as a new formula in the sequent. The proof commands for one property can constitute the proof scripts for such property. We investigate to summarize the generic proof script for the generic fault tolerant properties. When mechanically verifying the fault tolerant properties of developed system, we can customize the generic proof scripts and apply them to the theorem prover of PVS directly. Therefore, we can verify the fault tolerant properties of such system in the batch mode of PVS, and do not need to input the proof scripts to the theorem prover one by one. ProofLite [53] technique can provide user-friendly interface of batch mode execution and interactive proof scripting notation to the system designers. As the proof scripting notation supported by ProofLite enables a semi-literate proving style where specification and proof scripts reside in the same context, we investigate to extend the template based on the PVS model of GFTSA to involve generic fault tolerant properties accompanying with generic proof scripts by using XVCL and Prooflite techniques. Therefore, we can build a template for the PVS specification and proof scripts of developed systems guided by GFTSA, based on the PVS model of GFTSA and proof scripts of generic fault tolerant properties. As shown in Figure 7.4, when developing a safety critical distributed system guided by GFTSA, we can build x-frames for this developed system by adapting the primitive x-frames in the template. Based on these built x-frames, we can generate the PVS specification and proof scripts for the developed system automatically by running the XVCL processer. Based on the generated specification and proof scripts, we can mechanically verify the fault tolerant properties of developed system in the batch mode of PVS supported by ProofLite technique.

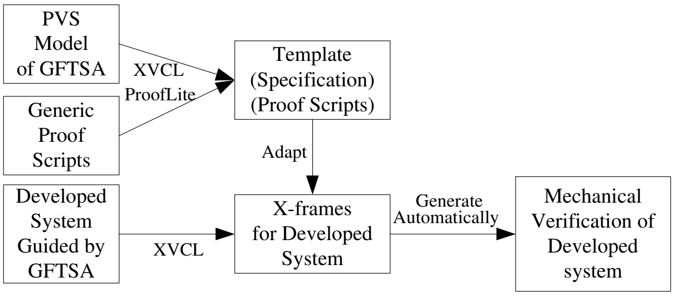

Figure 7.4: Mechanical Verification Process.

The primitive x-frames in the template is composed of the x-frames for the specification, and the x-frame for the proof scripts. These x-frames are illustrated in the following.

7.3.1 The x-frames in the Template for the Specification

When we develop safety critical distributed systems guided by GFTSA, the PVS model of developed systems can be customized from the PVS model of GFTSA by importing the corresponding theories of GFTSA. In order to make this customization process more efficient, we investigate to use XVCL technique [35] to build a template. Following the XVCL methodology, the template is built as generic x-frames based on the theories of GFTSA accompanying with XVCL commands, which mark the variation points. When developing specific systems, these variation points can be instantiated according to specific requirements. The x-frames in the template are related with the corresponding theories of GFTSA, shown in Figure 7.5. The x-frames involved in the template, namely *constants*, *object*, *sr*, *connector*, and *cc*, are all written as the combination of PVS specification language and XVCL commands. These x-frames is built for corresponding theories in the PVS specification of developed systems, which already have been presented and clarified in the Section 6.4.

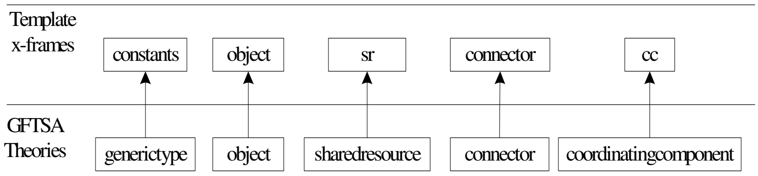

Figure 7.5: Relation between Template and GFTSA.

7.3.2 The x-frame in the Template for the Proof Scripts

Based on the x-frames for the specification, shown in the Figure 7.5, we can generate the PVS model of developed system guided by GFTSA. Furthermore, in order to provide the proof scripts for the fault tolerant properties of developed systems, we investigate to build the x-frame for the proof scripts by using the XVCL and ProofLite techniques. This x-frame, namely *ftsystem*, replaces the *ftsystem* x-frames presented in the Section 6.4 by involving proof scripts notation. This x-frame is built for the *ftsystem* theory of developed systems, which imports all the components & connectors theories of developed system to specify how these components & connectors synchronize together and the fault tolerant properties of such systems. By using the ProofLite technique, we can put the generic proof scripts written as the combination of ProofLite scripting notation and XVCL commands following the corresponding fault tolerant properties in the *ftsystem* x-frame.

The x-frame for ftsystem

In the *ftsystem* x-frame, we present two generic fault tolerant properties that a safety critical distributed system needs to preserve: one is that when a global exception raised in an *Object*, the system can tolerate this exception which means that any other *Object* can handle this exception and recover to normal execution process; the other is that when two global exceptions raised in different *Objects* concurrently, the system can handle this situation which means that other *Object* in the system can recover to normal execution process. The proof scripts written as the proof scripting notation of ProofLite are put following these two properties. In the proof scripts, since the proof command *"lemma name"* are used to introduce the lemma named *name* to the sequent, we also add several named lemmas to the x-frame for *ftsystem* theory to help the verification. These named lemmas are generic with some parameters which can be instantiated according to specific requirements of different safety critical distributed systems, shown in the *AXIOM*.

```
<x-frame name="ftsystem" language="PVS">

<value-of expr="?@systemname?"/>: THEORY
```

```
  BEGIN

     IMPORTING: <value-of expr="?@componentnames?"/>

   .........

   <value-of expr="?@racomname?"/>_member: AXIOM

     member(<value-of expr="?@agname?"/>,racomname.g_excepts)

   .......

   <value-of expr="?@agftname?"/>: LEMMA

    (EXISTS (obj: <value-of expr="?@racomname?"/>.Object):

     inter_state(obj)=<value-of expr="?@agname?"/> AND ue_rec(obj)=0)

        IMPLIES

      (FORALL (obj: <value-of expr="?@recomname?"/>.Object):

   .......

%|- <value-of expr="?@agftname?"/> : PROOF

%|- (then (flatten) (skolem!)

%|-  (lemma "<value-of expr="?@racomname?"/>_member") (prop)

.......

%|-  (lemma "<value-of expr="?@recomname?"/>_stateChange")

%|-  (instantiate -1 ("obj!1" "ccp!1"))(assert))

%|- QED

  <value-of expr="?@tgftname?"/> : LEMMA

  (EXISTS (obj1:

   <value-of expr="?@racomname1?"/>.Object),

    (obj2:<value-of expr="?@racomname2?"/>.Object):

  inter_state(obj1)=<value-of expr="?@tgname1?"/> AND ue_rec(obj1)=0 AND
```

```
     inter_state(obj2)=<value-of expr="?@tgname2?"/> AND ue_rec(obj2)=0)
       IMPLIES
        (FORALL (obj: <value-of expr="?@trecomname?"/>.Object):
 .......
%|- <value-of expr="?@tgftname?"/> : PROOF
%|- (then (flatten) (skolem!) (prop)
........
%|-  (lemma "<value-of expr="?@trecomname?"/>_UniExceptHandle2")
%|-  (instantiate -1 ("obj!1")) (assert))
%|- QED
 <value-of expr="?@systemname?"/>
</x-frame>
```

In this x-frame, for the property that a global exception raised, shown in the first *LEMMA*, the name of such property is expressed as variable *agftname*, the *Object* raised the exception are expressed as variable *racomname*, the raised exception are expressed as variable *agname*, and the other *Object* which can handle this exception are expressed as *recomname*. In the proof script for this property, the names of lemma used in the proof command *"lemma name"* are all generic with these variables. For the property that two global exception raised concurrently, shown in the second *LEMMA*, the name of such property is expressed as variable *tgftname*, the two *Objects* are expressed as *racomname1* and *racomname2*, two raised exceptions are expressed as *tgname1* and *tgname2*, and the other *Object* can recover to the normal execution is expressed as *trecomname*. The proof command *"lemma name"* in the proof script for this property are all generic with these variables.

By the support of XVCL and ProofLite techniques, we build a template involving the x-frames not only for the specification, but also for the proof scripts of developed safety critical distributed systems. By adapting these x-frames, we can auto-generate the PVS specification and proof scripts for the developed systems. Based on the generated specification and proof scripts, we can mechanically verify the fault tolerant properties of developed systems in batch mode of PVS.

7.4 Case Study-EPS (Electronic Power System)

In this section, we present a case study of an Electric Power System (EPS) to illustrate how we can generate the PVS specification and proof scripts of EPS from our built template. Based on the generated PVS specification and proof scripts, we can mechanically verify the fault tolerant properties of EPS in batch mode of PVS supported by ProofLite technique.

7.4.1 Electronic Power System(EPS)

As the primary source of power throughout the country, the electric power industry is a key critical infrastructure application domain. Electric power is generated, transmitted, and distributed by a complex system of power companies, utilities, brokers, and merchants. We build model topology of EPS corresponding to the

United States electric power grid information systems, as outlined in the NERC operating manual [14]. The lower levels of the model topology are an abstraction of the power grid in terms of power companies, generating stations, and substations. The higher levels of the model topology are control area, control region, and interconnection, shown in Figure 7.6.

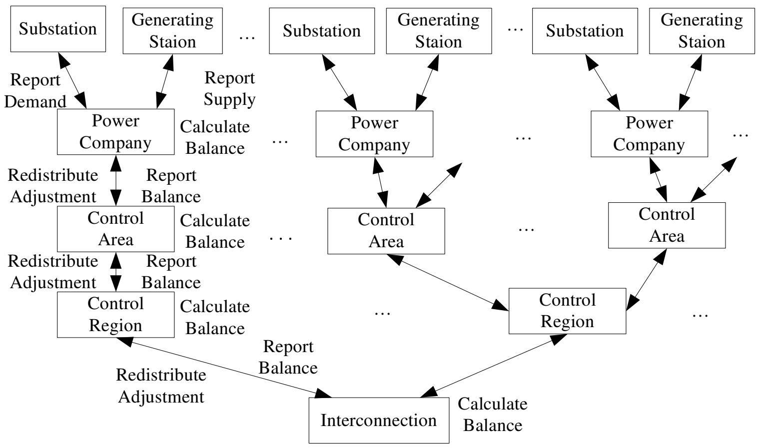

Figure 7.6: The Model Topology of EPS.

The substation reports the demand for power to its parent power company. The generating station controls and reports the supply of power being generated to its parent power company. The power company accepts data from substation and generating station, calculates the balance of power, reports the surplus or deficit to its parent control area, then balances supply and demand with any interchange adjustment accordingly. The control area accepts power balances from power companies, calculates and reports the control area balance to its parent control region, then redistributed any adjustment accordingly. The control region accepts power balances from its control areas, calculates and reports the control region balance to its parent interconnection, then redistributes any adjustment accordingly. The interconnection accepts power balances from its control regions, calculates its interconnection balance and swaps power with other interconnections according to demand, then redistributes power interchanges amongst its control regions according to demand. The hierarchical relationship among substation, generating station, power company, and control area can be applied to the hierarchical relationship among control area, control region, and interconnection. Therefore, we focus our development of EPS on the concurrency among substation, generation station, power company and control area. According to the box-and-line patterns of GFTSA shown in Figure 3.1, the EPS is composed of four *Objects*, namely *Substation*, *GeneratingStation*, *PowerCompany*, and *ControlArea*, six *Connectors*, namely *SPC, PCS, PCA, CAP, PCG, and GPC*, one *CoordinatingComponent*, namely *CC*, and two *SharedResources*, namely *PCDB* and *CADB*.

7.4.2 Generation of PVS Specification and Proof Scripts

Referring to the Figure 7.4, when developing a safety critical distributed system guided by GFTSA, we need to build x-frames for the developed system by adapting the corresponding x-frame in the template shown in Section 7.3. Figure 7.7 describes x-frame adaptation relationships between the x-frames of EPS and the built template. The *epsconstant* is built to declare the global constants which will be used by all the components & connectors theories in the EPS model. The *gsation*, *subsation*, *powercompany*, and *controlarea* are built for *GeneratingStation*, *Substation*, *PowerCompany*, and *ControlArea* correspondingly. The *gpc*, *pcg*, *pcs*, *spc*, *pca*, and *cap* x-frames are built for six connectors in the EPS. The *epscc* x-frame is built for the *CC* component, and the *pcdb* and *cadb* x-frames are built for the *SharedResource PCDB* and *CADB* correspondingly. The *eps* x-frame is built to describe how these components & connectors synchronize and the proof scripts for the fault tolerant properties of EPS.

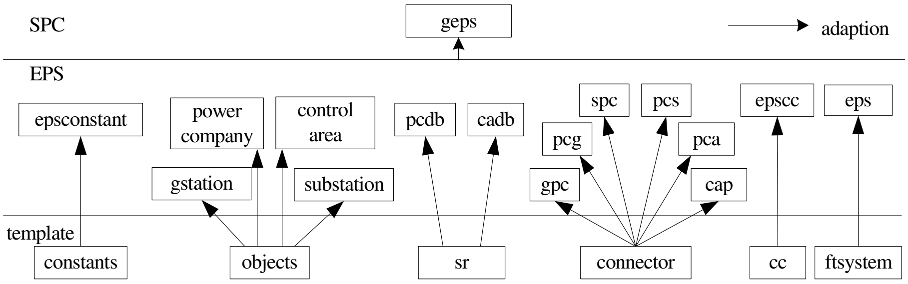

Figure 7.7: The x-frame Adaption Relationship of EPS.

When building these x-frames for EPS, we just need to instantiate the variation points defined in the x-frames of template according to the specific requirements of EPS . For example, when building *epscc* x-frame for the *CoordinatingComponent* of EPS, *epscc* needs to adapt *cc* x-frame and give values to the defined variables.

```
 <x-frame name="epscc" language="PVS">
<adapt x-frame="cc.xvcl">
 <set var="ccname" value="epscc"/>
 <insert break="egraph"> <![CDATA[IF check(PCDBAttacked, ge) OR
       check(CADBAttacked,ge) THEN  DBAttacked
                                   ELSE except
```

```
      ENDIF   ]]>
 </insert>
</adapt> </x-frame>
```

In this *epscc* x-frame, the defined variable *ccname* in the *cc* x-frame, shown in the Section 6.4.3, are given the value *epscc*. The function *egraph* is also defined that when the *CoordinatingComponent* in the EPS receives *PCDBATTACKED* or *DBATTACKED* exception, the resolved universal exception needs to be set as *DBATTACKED* exception.

Referring to the adaption relationship, the *eps* x-frame is built via adapting the *ftsystem* x-frame, shown in Figure 7.7. In the *eps* x-frame, we need to set values to the variables defined in the *ftsystem* according to the specific fault tolerant properties of EPS. The fault tolerant properties that EPS can handle involve *eps_ft1* and *eps_ft2* properties. The *eps_ft1* is that EPS can deal with a global exception, namely *PCDBAttacked*, raised in the *PowerCompany*, and the *eps_ft2* is that EPS can deal with two global exceptions, namely *PCDBAttacked* and *CADBAttacked*, which are raised concurrently in the *PowerCompany* and *ControlArea*. According to these two properties, we can build *eps* x-frame as follows.

```
<x-frame name="eps" language="PVS">
......
<set var="agftname" value="eps_ft1"/>
<set var="agname" value="PCDBAttacked"/>
<set var="racomname" value="powercompany"/>
<set var="recomname" value="substation"/>
```

```
 <set var="tgftname" value="eps_ft2"/>
 <set var="racomname1" value="powercompany"/>
 <set var="racomname2" value="controlarea"/>
 <set var="tgname1" value="PCDBAttacked"/>
 <set var="tgname2" value="CADBAttacked"/>
 <set var="trecomname" value="substation"/>
 <adapt x-frame="ftsystem.xvcl"/>
</x-frame>
```

In this *eps* x-frame, for the fault tolerant property *eps_ft1*, the variable *agname* for a raised global exception is set as *PCDBAttacked*, and the variable for the *Object* raising the exception is set as *powercompany*. Following this methodology, the variables defined in the *ftsystem* x-frames are all set values according to the specific fault tolerant properties.

By running the XVCL processor with the *geps SPC* file which adapts all of the 15 x-frames of EPS, we can generate the PVS specification and proof scripts of EPS automatically[1]. Figure 7.8 shows the model design of SCS in the box-and-line fashion guided by the pattern of GFTSA.

---

[1] The PVS specification and proofs scripts of EPS is presented in `http://www.comp.nus.edu.sg/~yuanling/eps-pvs.pdf`.

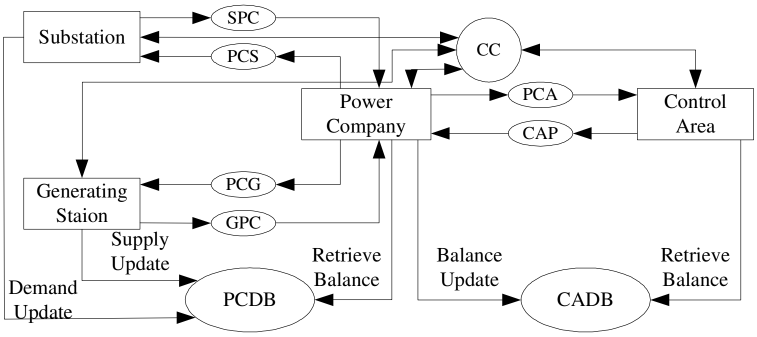

Figure 7.8: GFTSA Architecture View of EPS sub-System.

7.4.3 Mechanical Verification of EPS

Based on the generated PVS specification and proof scripts, we can mechanically verify the fault tolerant properties of EPS in batch mode of PVS supported by ProofLite technique. The generated *eps* theory is shown as follows, which involves not only the *eps_ft1* and *eps_ft2* fault tolerant properties of EPS, but also the corresponding proof scripts written as ProofLite proof scripting notation.

```
eps : THEORY
  BEGIN
  IMPORTING  gpc, pcg, spc, pcs, pca, cap, epscc, pcdb, cadb,
             gstation, substation, powercompany, controlarea
  powercompany_member: AXIOM
   member(PCDBAttacked, powercompany.g_excepts)
  ......
   eps_ft1: LEMMA
    (EXISTS (obj: powercompany.Object):
```

```
     inter_state(obj)=PCDBAttacked AND ue_rec(obj)=0) IMPLIES
     (FORALL (obj: substation.Object):
      member(inter_state(UniExceptHandle(obj)),substation.n_states))
%|- eps_ft1 : PROOF
%|- (then (flatten) (skolem!) (lemma "powercompany_member") (prop)
%|-  (replace -2 (-2 -1) rl) (lemma "powercompany_prop")
  .......
%|-  (instantiate -1 ("obj!1" "ccp!1")) (assert) (prop) (hide -4)
%|-  (lemma "substation_stateChange") (instantiate -1 ("obj!1" "ccp!1"))
%|- QED
     eps_ft2: LEMMA
    (EXISTS (obj1: powercompany.Object),(obj2:control.Object):
     inter_state(obj1)=PCDBAttacked AND ue_rec(obj1)=0 AND
      inter_state(obj2)=CADBAttacked AND ue_rec(obj2)=0) IMPLIES
        (FORALL (obj: substation.Object):
          member(inter_state(UniExceptHandle(obj)),substation.n_states))
%|-  eps_ft2 : PROOF
%|- (then (flatten) (skolem!) (prop) (lemma "powercompany_member")
.......
%|-  (lemma "substation_UniExceptHandle1") (replace -3 (-3 -1) rl)
%|-  (lemma "substation_member2") (replace -2 (-2 -1) rl)
%|-  (lemma "substation_UniExceptHandle2")
%|- QED
  END epsft
```

ProofLite technique provides a user-friendly interface to the PVS batch mode execution. Based on the proof scripts written as the proof scripts notation of ProofLite accompanying with fault tolerant properties, we can just use command *proveit* to execute the theorem prover in batch mode. Therefore, by running the command *proveit eps*, we can mechanically verify the fault tolerant properties of EPS in batch mode, and the verification result will be output to the *epsft.out* file. After checking out the verification results in the output file *epsft.out*, which are both *true*, we can conclude that the developed EPS guided by GFTSA can preserve the fault tolerant properties to satisfy high reliability requirements.

7.5 Conclusion

GFTSA is proposed to guide the development of safety critical distributed systems. In this chapter, we present a case study of LDAS to illustrate how we can develop specific safety critical distributed systems guided by GFTSA, generate the PVS model of LDAS from the template based on PVS model of GFTSA, and mechanically verify the fault tolerant properties of LDAS by using the theorem prover of PVS. In order to make the mechanical verification for the developed systems guided by GFTSA more systematic, we extend the template based on PVS model of GFTSA to involve not only generic PVS specification, but also generic proof scripts. This template is built as generic and adaptable x-frames, based on the PVS model of GFTSA and generic proof scripts of fault tolerant properties. The primitive x-frames in the template are written as the combination of PVS specification language, and ProofLite proof scripting notation, together with XVCL commands. By customizing this template, we can not only generate the PVS specification of developed systems, but also the proof scripts for the fault tolerant properties of these systems.

A case study of EPS is used to illustrate how we can generate the PVS specification and proof scripts of EPS from the extension template. Based on the generated specification and proof scripts, we can mechanically verify that EPS can preserve fault tolerant properties in batch mode of PVS supported by ProofLite technique.

Chapter 8

Conclusion and Future Work

This chapter summarizes the main contributions of the thesis and discussion possible directions for further research.

8.1 Conclusion

Distributed systems are becoming increasingly widespread in business and scientific computing environments, which often give rise to complex concurrent and interacting activities. Due to no small measure to their complexity, distributed systems are prone to faults and errors. For safety critical distributed systems, which have high requirements for reliability, fault tolerant techniques are necessary to provide a practical way to satisfy the reliability requirements. The concern of the fault tolerant properties makes the development of distributed systems more complicated. In order to address this problem, this thesis investigates to propose a novel heterogenous software architecture to ease the complexity of the development of the distributed systems with high reliability requirements.

One important contribution of this thesis is the building of a novel software architecture, namely Generic Fault Tolerant Software Architecture (GFTSA), which can provide a framework to guide the development of distributed systems with high reliability requirements. On the one hand, the architecture style of GFTSA combines several widely used basic architecture styles: object-oriented organization, pipe-and-filter, and repository style, which can provide a framework to guide the development of distributed systems involving both cooperative and competitive concurrency. On the other hand, in order to satisfy the reliability requirements of the distributed systems, GFTSA incorporates the fault tolerant techniques in the early system design phase, which provides an efficient way for system designers to reuse these techniques. Since interactive and concurrent properties of distributed systems, the fault tolerant techniques incorporated in GFTSA needs to concern the consequence of the exceptions not only to the component which raises the exception, but also to other components interact with this component. The exceptions occurred in the distributed environment are classified into local exceptions and global exception according to their influence to the interactive components. The fault tolerant techniques incorporated in GFTSA involve *idealized fault tolerant component* and *coordinated error recovery mechanism*, which can help deal with the local exceptions and global exceptions raised in the distributed environment.

In order to provide explicit and precise idioms & patterns to the system designers, another contribution of this thesis is to formally model the proposed GFTSA by using the formal language Object-Z. Many researchers have used formal language Z to formalize the state & computation of software architectures. Object-Z is an extension of Z to accommodate the object-orientated style. Compared to formal language Z, Object-Z can improve clarity of large specification through enhanced structuring, which can be used to model the static and dynamic features of GFTSA in a very explicit and understandable way. The components and connector in GFTSA all are represented as class schemas, which can be reused to develop the high level model of safety critical distributed systems by using the inheritance & instantiation mechanisms of Object-Z.

How the software architecture can be reused via customization in the development of specific systems is an interesting issue in the software architecture community. Another contribution of this thesis is to build a template based on the Object-Z model of GFTSA by using XVCL technique. This template is composed of generic and adaptable x-frames, which are written as the combination of Object-Z formal language and XVCL commands. This template can be customized to generate the Object-Z model of distributed systems with high reliability requirements automatically according to specific requirements. The customization process can be small or large change to the template, which cannot be totally supported by the inheritance & instantiation mechanisms of Object-Z, but can be supported by the XVCL technique.

Since the main intention of GFTSA is to guide the development of distributed systems with high reliability requirements, the significant properties that GFTSA needs to preserve are the fault tolerant properties, which can satisfy the high reliability requirements of such systems. Based on the Object-Z model of GFTSA, we can formally reason about the fault tolerant properties of GFTSA manually by using the reasoning rules of Object-Z. Since Object-Z has no tool support for verifying the models, the manual verification is laborious and error prone. Another interesting contribution of this thesis is to embed the GFTSA model in PVS to achieve mechanical verification support for reasoning about the fault tolerant properties. The powerful theorem prover of PVS can prove many results systematically and automatically. By using the theorem prover of PVS, we can mechanically verify the fault tolerant properties of GFTSA successfully.

The developed distributed systems guided by GFTSA also need to preserve fault tolerant properties to satisfy the reliability requirements. Since the theorem prover of PVS can mechanically verify the fault tolerant properties of GFTSA successfully, we investigate to apply the theorem prover of PVS in the verification of developed systems. Another interesting contribution of this thesis is to present a template approach for the auto-generation of specification and proof obligations at the customized system level from GFTSA. This template is built as generic and adaptable x-frames, which are written as the combination of PVS specification language, and ProofLite notation, accompanying with XVCL commands. The x-frames involved in the template are built based on the PVS model of GFTSA and generic proof scripts. When developing a safety critical distributed system, by customizing this template, we can generate not only the PVS model, but also the proof scripts for the fault tolerant properties of this system. The customized proof scripts for the fault tolerant properties can be applied directly to the theorem prover of PVS to mechanically verify these properties in the batch mode of PVS. This batch model of PVS supported by ProofLite technique can help us just use one command to verify these fault tolerant properties. Therefore, we do not need to input proof commands interactively to guide the theorem prover of PVS to verify properties.

Looking back to our whole thesis work, when we develop a specific safety critical distributed system, there are two ways we can go. The one way is that firstly we can generate the Object-Z model of specific system by adapting the template based on the Object-Z model of GFTSA, secondly, in order to mechanically verify the fault tolerant properties of developed system, we can generate the PVS model and proof scripts of developed system by adapting the template based on the PVS model of GFTSA and generic proofs scripts, finally, we can mechanically verify the fault tolerant properties of developed system in batch mode. Another way is that we directly generate the PVS model and proof scripts of developed system by adapting the template based on the PVS model of GFTSA and generic proof scripts, and mechanically verify the fault tolerant properties of developed system in batch mode. For the first way, the system designers can not only get the Objet-Z model, but also the PVS model. The Object-Z model can provide precise analysis and documentation, and the PVS model can support mechanical verification. But the system designers need to be familiar with both Object-Z and PVS formal languages, and take more effect to generate these two models. For the second way, the system designers do not need to move to the Object-Z model, and generate the PVS model and proof scripts of developed system directly. Since the PVS model also can provide the formal specification of developed system, I recommend that the system designers can directly go the second way to generate the PVS model and proof scripts of developed system by adapting the template based on the PVS model of GFTSA and generic proof scripts.

8.2 Future Work

In this thesis, we propose a novel heterogenous software architecture GFTSA to guide the high level system design of distributed systems with high reliability requirements. In order to satisfy reliability requirements of such systems, our proposed GFTSA incorporated *idealized fault tolerant component* and *coordinated er-*

ror recovery mechanism to deal with the exceptions occurred in the distributed environment. These fault tolerant techniques only can handle specific set of exceptions, some other exceptions, such as the inconsistent global states problem[54, 82], cannot be handled successfully. One of our future works is to incorporate more powerful fault tolerant techniques, such as selective checkpointing & rollback schemas [37] in GFTSA to deal with these complicated exceptions.

In order to make the mechanical verification of developed systems guided by GFTSA more efficient, we have built a template for the PVS specification and proof scripts of fault tolerant properties for such systems. This template involves generic proof scripts for two generic fault tolerant properties. In the future work, this template can be further extended to involve more generic fault tolerant properties accompanying with generic proof scripts.

GFTSA is proposed to guide the development of distributed systems with high reliability requirements. Since Object-Z language is a good modeling techniques that can provide explicit and precise structure and fault tolerant features of models to the users, by customizing the built template based on the Object-Z model of GFTSA, we can generate the Object-Z models of developed systems. However, our generated formal models for developed systems are high level model design, how these models can be transformed to the executive models is another further research direction for us. FT-SR [81] is a programming language developed for designing fault-tolerant distributed systems, which is the extensions to the concurrent programming language SR [6]. The distinguishing feature of FT-SR is its flexibility of structuring systems according to any structuring paradigms. This feature makes us choose programming language FT-SR to build the executive model of distributed system with high reliability requirements. Our future work is to build the rules to transform the Object-Z models to the executive models in FT-SR.

Bibliography

[1] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, 1995.

[2] R. Allen and D. Garlan. A formal approach to software architectures. In *Proceedings of IFIP'92*, 1992.

[3] V. Ambriola, P. Ciancarini, and C. Montangero. Software process enactment in Oikos. In *Proceedings of the Fourth ACM SIGSOFT*, pages 183–192, California, 1990.

[4] T. Anderson. *Resilient Computing Systems.* Collins Professional and Technical Books, 1985.

[5] T. Anderson and R. Kerr. Recovery blocks in action:a system supporting high reliability. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 447–457, San Francisco, 1976.

[6] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice.* The Benjamin/Cummings Publishing Company, 1993.

[7] C. Atkinson. *Object-Oriented Reuse, Concurrency, and Distribution.* Addison-Wesley, 1991.

[8] A. Avizienis. The N-Version Approach to Fault Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(2):1491–1501, 1985.

[9] L. M. Barroca and J. A. McDermid. Formal methods:use and relevance for the development of safety-critical systems. *Computer*, 35(6):579–599, 1992.

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

[11] R. Campbell and B. Randell. Error recovery in asynchronous system. *IEEE Transactions on Software Engineering*, SE-12(8):881–826, 1986.

[12] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Computer System*, 3(1):63–75, 1985.

[13] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design.* Addison-Wesley, 2000.

[14] North American Eletric Reliability Council. *NERC Operating Manual.* www.nerc.com/standards, January 1991.

[15] F. Cristian. Understanding fault-tolerant distributed systems. *Communication of the ACM*, 34(2):56–78, February 1991.

[16] F. Cristian. Exception handling and tolerance of software faults. *Software Fault Tolerance*, pages 81–107, 1994.

[17] D.Bjϕoner, C.W.George, B.Stig.Hansen, H.Laustrup, and S.Prehn. *A railway system, coordination'97, case study workshop example.* Technical Report 93, UNU/IIST, P.O.Box 3058,Macau, 1997.

[18] R. de Lemos. Describing evloving dependable systems using co-operative software architecture. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 320–329, 2001.

[19] J. S. Dong and S. Y. Liu. The semantics of extended SOFL. In *Proceedings of 26th Annual International Software and Application Conference*, pages 653–658, August 2002.

[20] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z.* Macmillan, 2000.

[21] R. Duke, G. Rose, and G. Smith. Object-Z: a specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[22] D. Garlan, R. Monroe, and D. Wile. ACME:an architecture description interchange language. In *Proceedings of CASCON'97*, November 1996.

[23] D. Garlan and D. Perry. Software architecture: Practice, potential and pitfalls. *In Proceedings of 16th Int. Conf. on Software Engineering*, 1994.

[24] J. Gary and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[25] P. Guerra, C. Rubira, and R. de Lemos. An idealized fault-tolerant architectural component. In *Proceeding of the 24th International Conference on Software Engineering-Workshop on Architecting Dependable Systems*, 2002.

[26] P. Guerra, C. Rubira, and R. de Lemos. A fault-tolerant software architecture for component-based systems. *Lecture Notes in Computer Science*, 2677:129–149, 2003.

[27] P. Guerra, C. Rubira, A. Romanovsky, and R. de Lemos. Integrating COTS software components into dependable software architecture. In *Proceeding of the 6th ISORC. IEEE Computer Society Press*, 2003.

[28] W. Harrison. Rpde: A framework for integrating tool fragments. *IEEE Software*, SE-4(6), 1987.

[29] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, 2nd edition, 1993.

[30] C.A.R. Hoare. Communicating sequential processes. *CACM*, vol.21(8):666–677, 1978.

[31] J. Hooman. Correctness of real time systems by contruction. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40, 1994.

[32] V. Issarny and J. P. Banatre. Architecture-based exception handling. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences,IEEE*, 2001.

[33] V. Issarny1 and A. Zarras. Software architecture and dependability. In *Formal Methods for Software Architectures*, pages 259–285, November 2003.

[34] P. Jalote and R. H. Campbell. Atomic actions for software fault tolerance using csp. *IEEE Transactions on Software Engineering*, SE-12(1):59–68, 1986.

[35] S. Jarzabek and S. B. Li. Eliminating redundancies with a "composition with adaption" meta-programming technique. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundation of Software Engineering,ACM Press*, pages 237–246, September 2003.

[36] S. Jarzabek and H. Zhang. XML-based method and tool for handling variant requirements in domain models. In *5th IEEE International Symposium on Requirements Engineering*, pages 166–173, August 2001.

[37] M. Kasbekar and C. Narayanan. Selective checkpointing and rollbacks in multi-threaded object-oriented environment. *IEEE Transactions on Reliability*, 48(4):325–337, 1999.

[38] H. Kopetz. *Real-time Systems.* Kluwer Academic Publishers, 1997.

[39] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processding Letters*, 25:153–158, 1987.

[40] J. C. Laprie. Dependability: Basic concepts and terminology. In *Dependable Computing and Fault-Tolerant Systems*, volume 5. Springer-Verlag, 1992.

[41] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice.* Second Edition,Prentice Hall, 1990.

[42] S. Y. Liu. A framework for developing dependable software systems using the SOFL method. In *First Workshop on Dependable Software (DSW2004)*, pages 131–140, Feb 2004.

[43] S. Y. Liu, M. Asuka, K. Komaya, and Y. Nakamura. An approach to specifying and verifying safety-critical systems with practical formal method SOFL. In *Proceedings of Fourth IEEE International Conference on Engineering of Complex Computer Systems*, pages 100–114, August 1998.

[44] S. Y. Liu, A J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: A formal engineering methodology for industrial applications. *IEEE Transactions on Software Engineering*, 24(1):24–25, 1998.

[45] D. Luckham and J. Vera. An event based architecture definition language. *IEEE Transactions on Software Engineering*, vol 21, 1995.

[46] N. A. Lynch, M. Merrit, W. E. Wehil, and A. Fekete. *Atomic Transactions.* Morgan Kaufmann, 1993.

[47] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architecture. *In Proceedings of 5th European Software Engineering Conference*, 1994.

[48] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, April 1998. IEEE Computer Society Press.

[49] B. P. Mahony and J. S. Dong. Timed communicating Object-Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.

[50] B. P. Mahony and J. S. Dong. Deep semantic links of TCSP and Object-Z: TCOZ approach. *Formal Aspects of Computing journal*, 13:142–160, 2002.

[51] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, SE-26(1):70–93, 2000.

[52] M. Moriconi, X.L. Qian, and R. A. Riemenschneider. Secure software architectures. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997.

[53] C. Munoz. Batch proving and proof scripting in pvs. http://research.nianet.org/munoz/ProofLite, 2005.

[54] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[55] S. Owre and J. M. Rushby. Formal verification for fault-tolerant architecture: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, SE-21(2):107–125, 1995.

[56] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. *11th International Conference on Automated Deduction*, pages 748–752, 1992.

[57] S. Owre and N. Shankar. *The formal semantics of PVS.* Computer Science laboratory, SRI International, Menlo Park, CA, 1997.

[58] S. Owre and N. Shankar. Writing PVS proof strategies. *Design and Application of Strategies/Tactics in Higher Order Logics*, pages 1–15, 2003.

[59] S. Owre, N. Shankar, and J. M. Rushby. *PVS System Guide.* Computer Science laboratory, SRI International, Menlo Park, CA, 1999.

[60] M. Rakic and N. Medvidovic. Increasing the confidence in off-the-shelf components: A software connector-based approach. In *Proceedings of the 2001 symposium on software reusability*, pages 11–18, May 2001.

[61] B. Randell, A. Romanovsky, R. Stroud, J. Xu, J. Zorzo, and A. F. Coordinated atomic actions: From concept to implementation. *Special Issue of IEEE Transactions on Computers*, 1997.

[62] A. Romanovsky, J. Xu, and B. Randell. Exception handling in object-oriented real-time distributed systems. In *1st IEEE international symposium on object-oriented real-time distributed computing*, 1998.

[63] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. *Software: Practice and Experience*, 35(3):195–236.

[64] J. Rushby, F. von Henke, and S. Owre. *An introduction to formal specificaiton and verification using EHDM.* Computer Science laboratory, SRI International, Menlo Park, CA, 1995.

[65] J. M. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, SE-19(1):13–23, 1993.

[66] T. Saridakis and V. Issarny. Fault tolerant software architectures. In *Technical report, INRIA/IRISA*, 1999.

[67] S. Schneider and J. Davies. *A brief history of Timed CSP.* Theoretical Computer Science, 1995.

[68] N. Shankar, S. Owre, , and J. M. Rushby. *PVS Prover Guide.* Computer Science laboratory, SRI International, Menlo Park, CA, 1999.

[69] M. Shaw and D. Garlan. Formulations and formalisms in software architecture. *In computer science today:recent trends and developments, Leture Notes in Computer science*, 1000, 1995.

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

[71] M. Shaw and D. Garlan. *Software Architecture:Perspectives on an Emerging Discipline.* Prentice-Hall,Englewood Cliffs,N.J., 1996.

[72] G. Smith. Extending ω for Object-Z. *9th International Conference of Z Users, Lecture Notes in Computer Science*, 967, 1995.

[73] G. Smith. Formal verification of object-z specifications. Technical Report 95-55, Software Verification Research Centre, University of Queensland, 1995.

[74] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 2000.

[75] M. S. Soe, H. Zhang, and S. Jarzabek. XVCL: A tutorial. In *Proc. of 14th, Int. Conf. on Software Engineering and Knowledge Engineering, SEKE'02,ACM Press*, pages 341–349, July 2002.

[76] J. Spivey. *Understanding Z:A specification language and its formal semantics,vol 3 of Cambridge Tracts in Theorectical Computer Science.* Cambridge University Press, 1988.

[77] J. M. Spivey. *The Z notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 1989.

[78] V. Stavridou and A. Riemenschneider. Provably dependable software architecture. In *Proceedings of the Third ACM SIGPLAN International Software Architecture Workshop*, 1998.

[79] J. Sun and J. S. Dong. Specifying and Reasoning about Generic Architecture in TCOZ. In P. Strooper and P. Muenchaisri, editors, *The 9th Asia-Pacific Software Engineering Conference (APSEC'02)*, pages 405–414. IEEE Press, December 2002.

[80] J. Sun, J. S. Dong, S. Jarzabek, and H. Wang. CAD system family architecture and verification: An integrated approach. *IEE Proceedings Software*, 153(3):102–112, 2006.

[81] V. T. Thomas. FT-SR: A programming language for constructing fault-tolerant distributed systems. Ph.D. Dissertation, Department of Computer Science, The University of Arizona, 1993.

[82] Y. M. Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, 1997.

[83] M. Xie, K.L. Poh, and Y.S. Dai. *Computing System Reliability: Models and Analysis*. Springer, 2004.

[84] J. Xu, B. Randell, A. Romanovasky, C. Rubira, R. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proc. 25th Int. Symp.on Fault-Tolerant Computing*, pages 499–508, Pasadena, June 1995.

[85] J. Xu, B. Randell, A. Romanovsky, R. Stroud, A. Zorzo, E. Canver, and F. von Henke. Rigorous development of a safety critical system based on coordinated

atomic actions. In *29th international symposium on fault-tolerant computing*, 1999.

[86] J. Xu, A. Romanovsky, and R. Campbell. Exception handling and resolution in distributed object systems. *IEEE TPDS*, 11(10), 2000.

[87] L. Yuan, J.S. Dong, and J. Sun. Modeling and customization of fault tolerant architecture using Object-Z/XVCL. In *Proc. Asia Pacific Software Engineering Conference'06(APSEC'06)*. IEEE Computer Society Press.

[88] L. Yuan, J.S. Dong, J. Sun, and H.A. Basit. Generic fault tolerant software architecture reasoning and customization. *IEEE Transactions on Reliability*, vol 55(3):421–435, 2006.

[89] H. Zhang, S. Jarzabek, and M. S. Soe. Xvcl approach to separating concerns in product family asserts. In *Proc. of Generative and Component-based Software Enginering(GCSE 2001)*, pages 36–47, September 2001.