Modeling and Customization of Fault Tolerant Architecture using Object-Z/XVCL

Ling Yuan, Jin Song Dong
School of Computing
National University of Singapore
{yuanling, dongjs}@comp.nus.edu.sg

Jing Sun
Department of Computer Science
The University of Auckland
j.sun@cs. auckland.ac.nz

Abstract

This paper proposes a novel heterogeneous software architecture FTA (Fault Tolerant Architecture). FTA incorporates idealized fault tolerant component concept and coordinated error recovery mechanism in the early system design phase. It can be reused in the high level model design of specific mission critical distributed systems with reliability requirements. The formal model of FTA in the Object-Z language is presented to provide precise idioms to the system designers. Formal proof using the Object-Z reasoning rules are constructed to demonstrate the fault tolerant properties of FTA. By analyzing the customization process, we also present a FTA template, expressed in x-frames using XVCL (XML-based Variant Configuration Language) methodology, to automate the customization process. We apply a sales control system case study to illustrate the customization of FTA.

1. Introduction

In order to satisfy the reliable requirements of mission critical distributed systems, fault tolerant techniques are employed to deal with the exceptions. Fault tolerance is the property of a system to provide a service complying with the specification in spite of faults occurred or occurring [9]. The concern of the fault tolerance makes the development of the distributed systems more complicated. Software architecture is considered as a critical design methodology to provide a generic framework to guide the development of distributed systems.

How to incorporate fault tolerant mechanisms 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 [7], description of software architectures with respect to their reliability properties [14], and the evolution of component-based software architecture by adding or changing components to guarantee reliability properties [3]. In this paper, we propose a novel heterogeneous software architecture FTA (Fault Tolerant Architecture). FTA integrates the fault tolerant mechanisms in the early design phase, which can be reused in the development of distributed systems with reliability requirements.

In practice, different kinds of concurrency might coexist in a distributed system which thus would require a generic supporting framework for controlling and coordinating these concurrent activities. The proposed FTA combines several widely used basic architecture styles to guide the development of such systems. These basic architecture styles [1] involve pipe-and-filter, repository style, and object-oriented organization.

The well-defined semantics and syntax make formal modeling techniques suitable for precisely specifying and formally verifying architecture designs. The formalisms of an architecture style can be used to provide precise, explicit common idioms and patterns to the software system designers [11]. Object-Z [4] is an object oriented structure that can describe internal state transitions and interface communications of software components. Compared to other formal languages such as Z, CSP, TCL, Object-Z has inheritance and instantiation mechanisms. These two mechanisms can help the specific distributed system developers to reuse the formal model of FTA. Furthermore, we could formally prove the fault tolerant properties of FTA by using the Object-Z reasoning rules.

In order to automate the customization process, we use XVCL technique (XML-based Variant Configuration Language) [8] to customize the formal model of FTA to the models of specific systems. A case study SCS (Sales Control System) [2] is presented to illustrate the customization process.

The remainder of this paper is organized as follows. Section 2 introduces the background knowledge about formal language Object-Z. Section 3 describes FTA including literal description, formal model and formal proof of fault tolerant properties. Section 4 presents a template for customization. Section 5 presents a case study SCS (Sales Control System) to illustrate how to produce the formal model of SCS automatically, and shows that SCS can preserves the fault tolerant properties. Section 6 concludes the paper and presents the future work.

2. The Object-Z Formal Language

Object-Z [4] 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. The inheritance and instantiation mechanisms of Object-Z can help the customization process. Below is a simple example *queue* to describe basic features of Object-Z. The *Queue[Item]* class schema is reused later to specify *FTComponent*, *CoordinatingComponent* and *SharedResource* class by the inheritance mechanism.

Queue[*Item*]

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

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

Join
$\Delta(items)$
$item? : Item$
$items' = items ^\frown \langle item? \rangle$

Leave
$\Delta(items)$
[operation schema]
$item! : Item$
$items \neq \langle\, \rangle$
$items = \langle item! \rangle ^\frown items'$

3. The Fault Tolerant Architecture (FTA)

3.1. The Overall Description of FTA

FTA involves object-oriented organization, pipe-and-filter architecture, and the repository style [1]. FTA can help develop mission critical distributed systems called FTS (Fault Tolerant Systems) , which have two kinds of concurrency- competitive and cooperative [6]. Competitive concurrency indicates that concurrent activities compete for some common resources, but without explicit cooperation. Cooperative concurrency means that concurrent activities cooperate and communicate with each other. FTS are composed of a set of components, called *FTComponents*, a set of *Connectors*, a set of *SharedResources* and a *CoordinatingComponent*, as shown in Fig. 1.

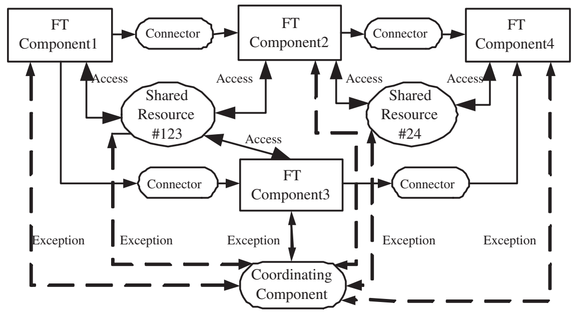

Figure 1. The Fault Tolerant Architecture.

The *FTComponent* derives from the object-oriented organization to accommodate the distributed environment, which can implement a separate task and potentially execute in parallel with other *FTComponents*. In order to help the *FTComponents* execute concurrently in the distributed system, the *connector* in FTA connects the out_port of one *FTComponent* and in_port of another *FTComponent*. This concept is similar to the pipe communication pattern in the pipe-and-filter architecture. The cooperative concurrency is modeled by the *FTComponents* interacting with each other via the *connectors* to cater for common goals. That *FTComponents* compete for *SharedResource* models the competitive concurrency, which derives from the repository style. The competitive concurrency need to guarantee the transaction semantics [5].

In order to satisfy the reliability requirements of mission critical distributed systems, several fault tolerant mechanisms are incorporated in FTA. Each *FTComponent* has its own exception context to deal with the exceptions occurring in the execution process, similar to the concept of idealized fault tolerant component [10].

We classify the exceptions raised in the *FTComponent* into two types: local exceptions and global exceptions. The influence of a local exception is limited within a single *FTComponent*. Global exceptions, on the other hand, affect the control flows of more than one *FTComponent* within a distributed system. FTA incorporates a coordinated error recovery mechanism to deal with these exceptions. Once a local exception is raised in one *FTComponent*, the *FTComponent* can call the corresponding exception handler in its own exception context to cope with the exception. If this exception can not be handled successfully, a global exception is signalled, which can be transferred to the *CoordinatingComponent*. If a global exception is originally raised in an *FTComponent*, this global exception is also passed to the *CoordinatingComponent*. The *CoordinatingComponent* broadcast the global exception to the related *FTComponents* and *SharedResources* within a distributed system. These components need to replace the normal execution with the exception handling execution.

When several global exceptions are raised in different *FTComponents* concurrently, these global exceptions are passed to the *CoordinatingComponent* concurrently. The *CoordinatingComponent* uses exception graph mechanism to resolve these concurrently raised exceptions into an unique global exception called universal exception which covers all the raised exceptions. When the *CoordinatingComponent* obtains the universal exception, it propagates this exception to all the related *FTComponents* and *SharedResources* involved in the distributed system. Furthermore the *FTComponents* call the corresponding exception handler in their own exception contexts to deal with this exception. The state of each *SharedResource* need to be restored to its prior normal state.

3.2. Formal Model of FTA

In the previous section, we presented FTA in a box-and-line fashion, accompanied with the literal explanations of the architecture style and fault tolerant mechanisms. Such description cannot provide precise, explicit common idioms and patterns of FTA to the system designers. In this section, Object-Z [4] is used to model FTA formally. The formal model of FTA[1] includes the *global types*, *FTComponent*, *Connector*, *CoordinatingComponent*, *ShareResource*, and *FTSystem* class schemas.

3.2.1 FTComponent

The *FTComponent* class schema describes the normal activities and error recovery activities of *FTComponent* in FTA. Note that the *Queue[item]* in section 2 is inherited with instantiation and rename mechanisms of Object-Z.

The constant variables *n_states*, *l_excepts* and *g_excepts* represent three different sets of states that an *FTComponent* 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 them into a set of *in_ports* and a set of *out_ports*. The constant variable *comp_msgs* represents a set of messages that an *FTComponent* 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 the state transition of the *FTComponent*. The *except_context* and *exception_handle* functions model the fault tolerant mechanisms described in the section 3.1.

[1]Due to the space limit, the complete formal model of FTA is presented at *http : //www.comp.nus.edu.sg/ ∼ yuanling/fta.pdf*.

FTComponent

$Queue[\downarrow SharedResource][sr\_qlist/items, ans\_sr?/item?,$
$\quad to\_sr!/item!, FromSR/Join, ToSR/Leave]$

$n\_states : \mathbb{P}\, NORMAL$
$l\_excepts : \mathbb{P}\, LE$
$g\_excepts : \mathbb{P}\, GE$
$in\_ports, out\_ports : \mathbb{P}\, PORT$
$comp\_msgs : \mathbb{P}\, MSG$
$coop\_msg : PORT \rightarrow MSG$
$transition : NORMAL \times (PORT \times MSG) \rightarrow$
$\quad OBSTATE \times (PORT \times MSG)$
$except\_context : LE \cup GE \nrightarrow EH$
$except\_handle : EH \rightarrow OBSTATE$

...

$inter\_state : OBSTATE$
$checkpoint : NORMAL$
$ue\_rec : SIG$

...

INIT
$inter\_state \in n\_states$
$checkpoint = inter\_state$
$ue\_rec = 0$

LocalExceptHandle
$\Delta(inter\_state)$

...

GlobalExceptPropagate
$exception! : GE$

...

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

...

UniExceptHandle
$\Delta(inter\_state, ue\_rec)$

...

Transition
$\Delta(inter\_state,$
$\quad checkpoint)$

$inter\_state \in n\_states$
$checkpoint' = inter\_state$
...

SRRequest
$req\_ob! : \downarrow$
$FTComponent$
$req\_sr! : \downarrow SharedResource$
$sr? : \downarrow SharedResource$

...

ToSR
$msg! : MSG$

...

FromSR
$ans\_ob? : \downarrow$
$FTComponent$

...

The *inter_state* denotes the current state of *FTComponent*, the *checkpoint* records the prior normal state of an *FTComponent*, the *ue_rec* indicates whether an *FTComponent* has received an *universal exception* from the *CoordinatingComponent*. The *Transition* operation denotes the state transitions of *FTComponent* according to the *transition* function. The *LocalExceptHandle*, *GlobalExceptProp-*

agate, *UniExceptReceive*, and *UniExceptHandle* operation schemas specify how the *FTComponent* deals with local and global exceptions. The *SRRequest*, *FromSR*, and *ToSR* operation schemas model how the *FTComponents* compete for the *SharedResources*.

3.2.2 Connector

The *Connector* class schema describes that a *connector* is responsible for connecting *send_port* of an *FTComponent* and *receive_port* of another *FTComponent* to transfer the message represented by *msg*.

Connector

$send\_port, receive\_port : PORT$

$\exists\, send\_ob, receive\_ob : \downarrow$
$\quad FTComponent;\ msg : MSG \mid$
$\quad send\_ob \neq receive\_ob \land$
$\quad send\_port \in send\_ob.out\_ports \land$
$\quad receive\_port \in receive\_ob.in\_ports$
$\quad\quad \bullet\ msg = send\_ob.coop\_msg(send\_port) \land$
$\quad\quad\quad receive\_ob.coop\_msg(receive\_port)$
$\quad\quad\quad\quad = msg$

3.2.3 CoordinatingComponent

The *CoordinatingComponent* class schema describes how the *CoordinatingComponent* implements the coordinated error recovery mechanism when a global exception is raised or several global exceptions are raised concurrently. The *except_graph* is a function to resolve several concurrently raised exceptions into an *universal exception* represented by *uni_exception* that can cover all the raised exceptions. The state variable *exceptions* represents the sequence of received exceptions. The *ExceptRec* and *ExceptGraph* operation schemas specify how the *CoordinatingComponent* implements the coordinated error recovery mechanism.

CoordinatingComponent

$Queue[GE][exceptions/items, exception?/item?,$
$\quad ExceptRec/Join]$

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

ExceptGraph

$\Delta(exceptions)$
$uni\_exception! : GE$

$exceptions \neq \langle\, \rangle$
$uni\_exception! = except\_graph(exceptions)$
$exceptions' = \langle\, \rangle$

3.2.4 SharedResource

The *SharedResource* class schema models how the *SharedResource* can guarantee the transaction semantics when receiving messages from *FTComponents* and preserve consistent state when facing exceptions.

SharedResource

$Queue[\downarrow FTComponent][ob\_qlist/items,$
$\quad req\_ob?/item?, ans\_ob!/item!,$
$\quad\quad ObList/Join, Available/Leave]$

$states : \mathbb{P}\, SRSTATE$
$trans : SRSTATE \times MSG \to SRSTATE$

$\ldots$

$semaphore : SIG$
$sr\_state : SRSTATE$
$checkpoint : SRSTATE$

$\ldots$

INIT

$semaphore = 0$
$checkpoint = sr\_state$

ObList

$req\_sr? : \downarrow$
$\quad SharedResource$

$\ldots$

Available

$\Delta(semaphore)$
$ans\_sr! : \downarrow SharedResource$

$\ldots$

Except

$\Delta(semaphore, ob\_qlist,$
$\quad sr\_state)$
$uni\_exception? : GE$

$\ldots$

Trans

$\Delta(semaphore, sr\_state,$
$\quad checkpoint)$
$to\_sr? : \downarrow SharedResource$
$msg? : MSG$

$\ldots$

The *states* represents a set of states that the *SharedResource* can be in and the *trans* is used to model the state transition of the *SharedResource* when it receives a message from *FTComponent*. The state variables *semaphore*, *sr_state* and *checkpoint* represent the signal to show whether the *SharedResource* is accessed by an *FTComponent*, the current state of *SharedResource* and the recorded prior normal state of *SharedResource* respectively. The *ObList*, *Available*, and *Trans* operation schemas specify how the *SharedResource* can guarantee the transaction semantics. The *Except* operation describes how the *SharedResource* deals with exceptions.

3.2.5 FTSystem

The *FTSystem* class schema describes how the components and connectors in FTA which constitute an FTS (Fault Tolerant System) are synchronized. The constant variable *critical* represents the set of *FTComponents* whose *Fail* state can cause the whole FTS to stop. The *Result_Control* is a function to check the execution result of FTS. The instances of components and connectors in the FTS are all declared in the state schema. The secondary variable *ob_fail* records a set of *FTComponents* in the *Fail* state. How the *FTComponents* interact with each other, compete for the *SharedResources*, and how to deal with exceptions are specified in the declared operations.

$$
\begin{array}{l}
\textit{FTSystem} \\
\hline
\quad \textit{critical} : \mathbb{P} \downarrow \textit{FTComponent} \\
\quad \textit{Result\_Control} : \mathbb{P} \downarrow \textit{FTComponent} \\
\qquad\qquad \rightarrow \textit{RESULT} \\
\hline
\quad \forall \textit{fobs} : \mathbb{P} \downarrow \textit{FTComponent} \bullet \exists \textit{fob} : \textit{fobs} \bullet \\
\qquad \textit{fob} \in \textit{critical} \Rightarrow \textit{Result\_Control}(\textit{fobs}) = \textit{stop} \\
\quad \forall \textit{fobs} : \mathbb{P} \downarrow \textit{FTComponent} \bullet \forall \textit{fob} : \textit{fobs} \bullet \\
\qquad \textit{fob} \notin \textit{critical} \Rightarrow \textit{Result\_Control}(\textit{fobs}) = \textit{tolerate} \\
\\
\hline
\quad \textit{obs} : \mathbb{P} \downarrow \textit{FTComponent} \\
\quad \textit{cs} : \mathbb{P} \downarrow \textit{connector} \\
\quad \textit{coco} : \downarrow \textit{CoordinatingComponent} \\
\quad \textit{srs} : \mathbb{P} \downarrow \textit{SharedResource} \\
\quad \Delta \\
\quad \textit{ob\_fail} : \mathbb{P} \downarrow \textit{FTComponent} \\
\hline
\quad \ldots \\
\hline
\\
\textsc{Init} \\
\hline
\quad \ldots \\
\hline
\\
\textit{SystemRecover} \\
\hline
\quad \textit{Result\_Control}(\textit{ob\_fail}) = \textit{tolerate} \\
\quad \forall \textit{ob} : \textit{obs} \bullet \textit{ob}.\textsc{Init} \\
\hline
\textit{Transition} \mathrel{\widehat=} \bigwedge \textit{ob} : \textit{obs} \bullet \textit{ob.Transition} \\
\textit{ExceptPropagate} \mathrel{\widehat=} \bigwedge \textit{ob} : \textit{obs} \bullet \\
\qquad \textit{ob.GlobalExceptPropagate} \parallel \textit{coco.ExceptRec} \\
\textit{ExceptGraph} \mathrel{\widehat=} \textit{coco.ExceptGraph} \parallel (\bigwedge \textit{ob} : \textit{obs} \bullet \\
\qquad\qquad \textit{ob.UEReceive} \wedge \bigwedge \textit{sr} : \textit{srs} \bullet \textit{sr.Except}) \\
\textit{ObReqSR} \mathrel{\widehat=} \bigwedge \textit{ob} : \textit{obs} \bullet \textit{ob.SRReq} \parallel \\
\qquad\qquad (\bigwedge \textit{sr} : \textit{srs} \bullet \textit{sr.ObList}) \\
\textit{SRAnsOb} \mathrel{\widehat=} \bigwedge \textit{sr} : \textit{srs} \bullet \textit{sr.Available} \parallel \\
\qquad\qquad (\bigwedge \textit{ob} : \textit{obs} \bullet \textit{ob.FromSR}) \\
\textit{ObAccSR} \mathrel{\widehat=} \bigwedge \textit{ob} : \textit{obs} \bullet \textit{ob.ToSR} \parallel (\bigwedge \textit{sr} : \textit{srs} \bullet \textit{sr.Trans}) \\
\hline
\end{array}
$$

3.3 Reasoning about FTA

Reasoning about the formal model of the system enables the designer to gain confidence in the correctness of the system development [13]. Reasoning about FTA mainly involves showing that the formal model of FTA can preserve fault tolerant properties, which are expressed as theorems. The proof process needs to demonstrate that fault tolerant properties can be derived from the formal model of FTA by using Object-Z reasoning rules. The following items [2] show the fault tolerant properties that FTA can preserve.

1. When a global exception is raised by a *FTComponent* in the FTS, all of the *FTComponents* & *SharedResources* in the FTS should be informed about the exception. This property can be formally expressed as the following theorem.

   Theorem

$$
\begin{array}{l}
\textit{FTSystem} :: \exists\, \textit{ob} : \textit{obs} \mid \textit{ob.inter\_state} \in \textit{ob.g\_excepts} \\
\qquad \vdash \forall\, \textit{ob} : \textit{obs};\ \textit{sr} : \textit{srs} \bullet (\textit{ob.ue\_rec}' = 1 \\
\qquad\qquad \wedge\ \textit{sr.sr\_state}' = \textit{sr.checkpoint})
\end{array}
$$

2. When two global exceptions are raised concurrently by two *FTComponents* in the FTS, all the *FTComponents* in the FTS can be informed about an universal global exception. This property can be formally expressed as follows.

   Theorem

$$
\begin{array}{l}
\textit{FTSystem} :: \textit{ob}_1, \textit{ob}_2 : \textit{obs} \mid \\
\qquad (\textit{ob}_1.\textit{inter\_state} = \textit{ob}_1.\textit{g\_excepts} \wedge \\
\qquad \textit{ob}_2.\textit{inter\_state} = \textit{ob}_2.\textit{g\_excepts} \wedge \textit{ob}_1 \neq \textit{ob}_2) \\
\qquad\qquad \vdash \forall\, \textit{ob} : \textit{obs} \bullet \textit{ob.ue\_rec}' = 1
\end{array}
$$

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

   Theorem

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

4. The Template of FTA in XVCL

By customizing formal model of FTA, we can build formal models of specific mission critical distributed systems. During the customization process, each class schema in the formal model of FTA can be inherited by the class schemas of specific mission critical distributed systems. Besides inheritance, the customization process involves several other mechanisms such as declaring predicates of constant variables, defining initial schema and newly operation schemas, etc. The class schema below is shown as an example to describe the customization from the FTA model to the formal

[2]Due to the space limit, the complete formal proof of FTA can be found at `http://www.comp.nus.edu.sg/~yuanling/ftap.pdf`.

model of a specific mission critical distributed system. Note that the items decorated with quotation mark can be instantiated according to the requirements of specific systems.

```
“ftcomponnetname”
FTComponent″rename”
“newfunction”
    “newvar”
    ─────────
    n_states = {“nstates”}
    l_excepts = {″lexcepts”}
    ...
...
“newop”
```

The class schemas above consist of fixed points and variable points which can be reused in the high level design of specific mission critical distributed systems by customization. During the customization process, we just need to put effort in the variable part by giving concrete definitions to the items decorated with quotation mark. Therefore, we can build a template of FTA formal model for the customization.

XVCL (XML-based Variant Configuration Language) [8] is a meta-programming technique developed to facilitate building flexible, adaptable and reusable software artifacts. In XVCL technique, all of small or large variation points can be represented as meta-expressions that are instantiated during the customization process according to the specific requirements. Following the mechanism of XVCL, the template of FTA formal model can be built as generic, adaptable fragments, called x-frames. Each x-frame is an XML file composed of Object-Z class schema LaTeX code and XVCL commands. In order to help the flexible reuse of the template of FTA formal model, we build five primitive x-frames[3], called *ftComponent*, *connector*, *coco*, *sr* and *ftsystem*. In the primitive x-frame, each item decorated with quotation mark is represented as a variable with the same name. As shown in Fig. 2, we build the template of FTA formal model in XVCL.

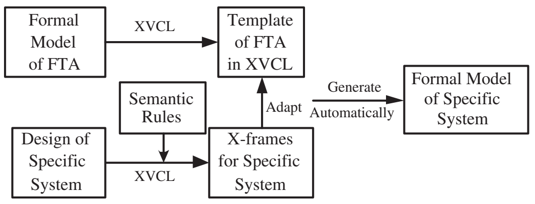

Figure 2. The Customization Process.

[3]The x-frames built for the template of FTA and the semantic rules for guiding the specific systems designers are all presented at `http://www.comp.nus.edu.sg/~yuanling/template.pdf`.

The template of FTA formal model is built based on the formal model of FTA with inheritance and instantiation mechanisms of Object-Z which ensure that the fault tolerant properties of FTA can be preserved. The instantiation of variation points in the template by the use of XVCL technique during the adaptation process do not consider the semantic factor. If the formal model of specific system generated by customization of the template need to preserve the fault tolerant properties of FTA, we should establish semantic rules[3] to guarantee the compatibility between the formal model of specific system and that of FTA. These rules present guidelines to the designers when giving a concrete definition to the variables in the x-frames of the template or declaring a new variable.

The following case study illustrates the customization process involving the x-frames building for the specific mission critical distributed system guided by the semantic rules, and the generation of the formal model of the specific system, as shown in Fig. 2.

5. A Case Study - Sales Control System (SCS)

In this section, we use a SCS system case study to demonstrate how FTA can guide the high level design of specific mission critical distributed systems.

5.1. Development of SCS Guided by FTA

The mission critical distributed system SCS (Sales Control System) [2] 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. The SCS consists of a database, a set of control points and a set of sales points. Fig. 3 shows an example of SCS, which is composed of two control points, a database and three sales points.

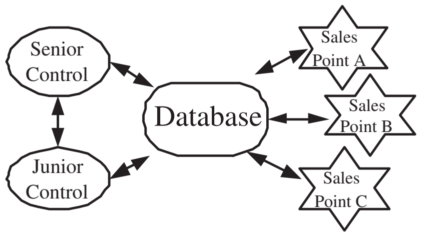

Figure 3. The Sales Control System.

The control point provide the interface that allow 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 need 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 need to guarantee the transaction semantics of the database.

According to the box-and-line patterns of FTA shown in Fig.1, the SCS is composed of five *FTComponents*, 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. Fig. 4 shows the model design of SCS in the box-and-line fashion guided by the pattern of FTA.

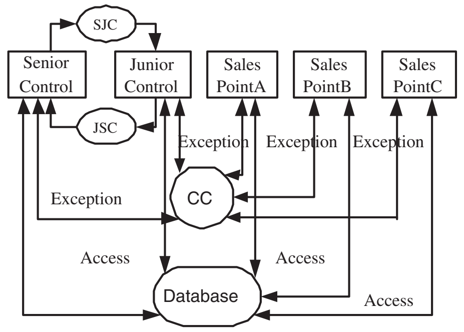

Figure 4. FTA Architecture View of SCS.

5.2 Generation of Formal Model of SCS

In the following, we generate the formal model[4] of SCS by customizing the template of FTA presented in section 4. The five primitive x-frames in the template can be reused during the customization via adaption. Following the mechanisms of XVCL, we can build x-frames for formal model of SCS complying with semantic rules defined in the section 4. Fig. 5 describes x-frame adaption relationship between the SCS and the template of FTA. The *sc*, *jc*, *spa*, *spb*, and *spc* x-frames are built for the five *FTComponents* in the SCS. The *sjc* and *jsc* x-frames are built for the connectors. 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. By running the XVCL processor with the fscs SPC file which adapts all of the ten x-frames of SCS, we can generate the formal model of SCS automatically, which is the last step of customization process shown in Fig. 2. In the following, a representative class schema is presented to illustrate the features of the formal model of SCS.

[4]The x-frames for the SCS and complete formal model of SCS are all presented at `http://www.comp.nus.edu.sg/~yuanling/fscs.pdf`.

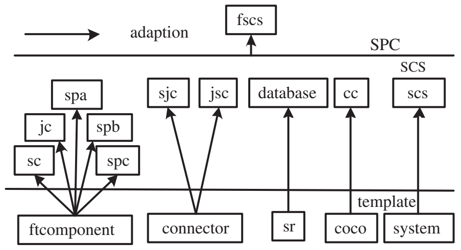

Figure 5. The x-frame Adaption Relationship.

JuniorControl
FTComponent

$Local\_ExcepthHandle : OBSTATE \rightarrow (PORT \times MSG)$
$Global\_ExceptHandle : OBSTATE \rightarrow Handler$

$Local\_ExceptHandle(NetworkDisconnected)$
$\quad = (JSC\_Out, RequestUpdate)$
$Local\_ExceptHandle(InformationLost)$
$\quad = DatabaseRecover$
$n\_states = \{NormalProcess, AuthorizeRequest\}$
$l\_excepts = \{NetworkDisconnected\}$
$g\_excepts = \{InformationLost\}$
$in\_ports = \{SJC\_In\}$
$out\_ports = \{JSC\_Out\}$
$comp\_msgs = \{ProductUpdate\}$
$coop\_msg = \{(SJC\_In, UpdateApproved),$
$\quad (JSC\_Out, RequestUpdate)\}$
$Trans = \{((NormalProcess, (NULL, NULL)),$
$\quad (AuthorizeRequest, (JSC\_Out, RequestUpdate))),$
$\quad ((AuthorizeRequest, (SJC\_In, UpdateApproved)),$
$\quad (NormalProcess, (NULL, NULL))),$
$\quad ((AuthorizeRequest, (NULL, NetworkDisconnected)),$
$\quad (NetworkDisconnected, (NULL, NULL))),$
$\quad ((NormalProcess, (NULL, InformationLost)),$
$\quad (InformationLost, (NULL, NULL)))\}$
$except\_context = \{(NetworkDisconnected,$
$\quad Local\_ExceptHandle), (InformationLost,$
$\qquad Global\_ExceptHandle)\}$
$except\_handle = \{(Local\_ExceptHandle, AuthorizeRequest),$
$\quad (Global\_ExceptHandle, NormalProcess)\}$

INIT
$inter\_state = NormalProcess$

The *JuniorControl* class represents the *FTComponents* 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 *FTComponent* 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. The global exception *InformationLost* represents that the *Database* has lost some product information. A *Global_ExceptHandle* function is defined to handle this exception.

5.3. Reasoning about SCS

By adapting the template of FTA complying with defined semantic rules, we generate the formal model of SCS. The following theorems are two significant fault tolerant properties of SCS.

1. When the *InformationLostA* is raised in the *SalesPointA*, which represents that the *SalesPointA* can not get the product information from the *Database*, the SCS can tolerate this exception.

   Theorem

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

2. When the *InformationLostA* is raised in the *SalesPointA*, and concurrently the *InformationLostB* is raised in the *SalesPointB*, the SCS can also handle this situation.

   Theorem

$$\begin{aligned} &SCS :: spa.inter\_state = InformationLostA \wedge \\ &\quad spb.inter\_state = InformationLostB \\ &\qquad \vdash \forall scs : SCS;\ ob : scs.obs \bullet \\ &\qquad\quad ob.inter\_state' \in ob.n\_state \end{aligned}$$

Following the same methodology shown in the section 3.3, we can prove that the formal model of SCS can preserve these fault tolerant properties[5].

6. Conclusion

This paper proposes a novel heterogeneous software architecture FTA (Fault Tolerant Architecture) which can guide the development of mission critical distributed systems. FTA integrates fault tolerant mechanisms with functional aspects in the early system design phase and combines several widely used basic architecture styles. Formal model of FTA can be directly reused in the high level design of specific mission critical distributed systems via customization process. XVCL technique makes the customization process more convenient and automatic. The constructed formal proof based on Object-Z reasoning rules can demonstrate that formal model of specific mission critical distributed systems generated by customization can preserve fault tolerant properties. A sales control system is used to illustrate the customization process and demonstrate that the specific mission critical distributed system can preserve the fault tolerant properties.

Formal proofs shown in this paper are all done manually, which is laborious and error-prone. It would be worth trying to prove these theorems using PVS(Prototype Verification System) [12]. PVS is a verification system developed at SRI, which has a powerful interactive theorem prover and its automation suffices to prove many straightforward results automatically.

[5]The complete formal proof of SCS are all presented at `http://www.comp.nus.edu.sg/~yuanling/pscs.pdf`.

References

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

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

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

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

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

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

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

[8] 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.

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

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

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

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

[13] 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.

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