Factorizing Fault Tolerance

I.S.W.B. Prasetya, S.D. Swierstra*

Abstract

This paper presents a theory of component based development for exception-handling in fault tolerant systems. The theory is based on a general theory of composition, which enables us to factorize the temporal specification of a system into the specifications of its components. This is a new development because in the past efforts to set up such a theory have always been hindered by the problem of composing progress properties.

1 Introduction

The design of software systems becomes more and more dependent on the use of components. This is for a good reason. The approach stimulates the separate development of software components, which in turn greatly enhances the components' reusability, maintainability, portability, and scalability. A theory for reasoning about the behavior of a complicated system, starting from the behavior of its components, is crucial for such an approach; it enables us to infer the behavior of a system of concurrent components by looking at the individual specification of the components. A prerequisite for this approach is that components can be specified and verified independently from the actual environment they will be functioning in.

Most software systems of nowadays are capable of detecting and handling faults on its own, although the degree of fault tolerance may of course vary. This paper presents a theory of components composition for a certain class of fault tolerant systems. Such a theory has not been sufficiently addressed in the past. Most approaches are restricted to a rather loose composition of synchronized components such as layering [6, 1, 10], which is too weak to express, for example, overriding of the main function of a system by a fault handler. This is not surprising because the needed underlying theory of composition is thus far lacking: such a theory essentially provides laws of the form:

$$\frac{P \textbf{ sat } A \ , \ Q \textbf{ sat } B}{P [\!] Q \textbf{ sat } C_{(A,B)}}$$

which states that if components P and Q satisfy respectively properties A and B, then the composite $P[\!]Q$ satisfies C, which is some combination of A

*Universiteit Utrecht, Informatica Instituut, Postbus 80.089, 3508 TB Utrecht, Nederland. Email: wishnu@cs.uu.nl, doaitse@cs.uu.nl

and B. Applying the law backwardly direction enables us to factorize a global specification of a system into local specifications of its components, which in their turn can be analyzed and verified separately. However, the 'compositionality' of reactive properties, which are used a lot in specifying fault tolerance, is difficult. Many theories were developed [7, 3, 4, 12, 13], but all fail for various reasons to satisfactorily address the issue. In this paper we first present a more general theory of composition based on a quite old approach by Singh [14] to exploit information on when and where (on which data segment) a component will temporarily suspend its interference. The resulting theory is quite elegant and expressive. Its consistency has been checked in a theorem prover HOL [11]. The proof code and theory listing are available at `www.cs.uu.nl/~wishnu`. Compared to other approaches such as [3, 4] our approach uses more constrained synchronization conditions, but what we gain is a great reduction in the amount of 'extra' information required to get a progress specification to compose well.

Building up on this theory we present a theory for composing components of fault tolerant systems. In particular, we consider the use of exceptions and exception handling. We will also show how one can use the theory to test a system against an 'adversary'. In most other approaches [5, 2, 6, 8, 10] the presence of an adversary is treated implicitly, resulting in confusion between the 'normal' behavior of a system, and its behavior under faulty conditions. Because the adversary is now just another component, we can also test the system against different adversaries. The system's reaction may then be calculated using the composition theory.

Let us emphasize that this paper is not intended to provide a complete formalization of various fault tolerance techniques, but rather, it provides a skeleton theory. The user can extend it or treat it as a template to build his/her own specific fault tolerance model.

Sections Overview
Section 2 presents the underlying formalism and a general composition theory. Section 3 describes our model of exception handling in systems, and presents a theory to build such a system in a modular way. Only systems handling one kind of fault are considered. Section 4 generalizes the theory in Section 3 to a scenario with multiple faults.

2 A Basic Composition Theory

Earlier efforts [7, 3, 4, 12, 13] to set up a composition theory for temporal properties have always been hindered by the problem of composing progress properties. The problem can be described as follows. Suppose a component P can progress from p to q, written as $P \vdash p \mapsto q$. What would be a reasonable condition for Q (another component) such that the parallel composition of P and Q (written $P[\!]Q$) preserves the progress? In a non-component based approach we would just fix the code of Q, which is an easy solution, but will render P non-reusable. So, to increase P's reusability, we prefer the conditions on Q to

```
[] IF Prun /\ Ppc=m0 /\ gtStuckExc THEN Ppc:=m1
[] IF Prun /\ Ppc=m1 THEN gate,Ppc := resetGate(),m2
[] IF Prun /\ Ppc=m2 THEN gtStuckExc,Prun,Ppc := False,True,m0
[] IF Qrun /\ Qpc=m0 /\ senDeadExc THEN Qpc := m1
[] IF Qrun /\ Qpc=m1 THEN log,Qpc := append "sensor dead" log, m2
[] IF Qrun /\ Qpc=m2 THEN senDeadExc,Qrun,Qpc := False,True,m0
```

Figure 1: An example of a UNITY program

be as weak as possible.

A first attemp might be to require that Q preserves p. This would make a powerful composition law, except that it is not valid. During its progress from p to q, the component P can pass through some intermediate states outside p. Q would then be unconstrained, and may thus block P's progress. We can require Q to preserve *all* intermediate states passed by P during its progress to q. The good thing is that this constraint on Q is almost the weakest possible. The bad thing is that a progress specification of a component must now explicitly mention 'extra information', namely all its intermediate states. This can be unacceptable for several reasons: (1) too cumbersome, (2) the information is not always available, and (3) changes in implementation are likely to change the set of intermediate states.

In our approach we parameterize a temporal property of a component with a set of variables V for which the property may be sensitive to external changes (changes resulting from external actions). If there is a way for P to indicate that it is in the process of progressing from p to q, for example by raising a flag a, the progress can be preserved in $P[\!]Q$ by requiring Q to suspend interference on V while a holds. This is a stronger requirement than the one proposed before, but the approach also reduces the amount of 'extra information' needed to compose progress specifications, which hopefully will result in a reasonable compromise.

The programming logic we use is based on UNITY. A UNITY program is a set of terminating, atomic, and guarded actions –if the guard is false the action can still be executed (enabled) though with no effect (skip). An execution of a UNITY program is infinite, in which at each step an action is non-deterministically selected but the selection is weakly fair (no action can be indefinitely ignored). There are no primitive control structures (like sequencing, loop, or branch), but they can be programmed. Figure 1 shows an example of a UNITY program. It is the translation of a fault handler (FH0) in Figure 4 –this will be explained later. The code in Figure 1 shows a program consisting of a list of actions, separated by `[]` symbols.

Before presenting our composition theory, the next subsections introduce a some new notions (like sensitivity to external changes and temporary non-interference).

Some Note on Notation

In the sequel P and Q are UNITY programs; a, b, p, q, r, s, and J are predicates;

α and β are actions; and V and W are sets of variables. We write $J \vdash p$ to mean $(\forall s :: J\ s \Rightarrow p\ s)$, meaning that under assumption J, p holds everywhere. We use $\mathbf{var}(P)$ and $\mathbf{loc}(P)$ to denote respectively: (1) the set of all variables read or written by P, and (2) the set of all local variables of P (variables writable only by P). □

2.1 Predicate Confinement

Total functions f and g of the same type, are partially equal over subdomain V, written $f =_V g$, if $f\ x = g\ x$ for all $x \in V$. The set of all functions of f such that $f =_V g$ is denoted by $[g]_V$. A program state is a function from variables to values, so we can talk about partially equal states in the above sense. A state predicate is a function from state to **bool**, or equivalently a set of states. Given a predicate p and a set of variables V, we define p to be *confined* by V if it only constrains the value of variables in V. This can be formally defined as follows:

$$p\ \mathbf{conf}\ V\ =\ (\forall s.\ p\ s = ([s]_V \subseteq p)) \tag{1}$$

For example, $x > y + z$ is confined by $\{x, y, z\}$, but not by any set smaller than $\{x, y, z\}$. Notice also that a predicate which is confined by V cannot be falsified by a program only updating variables outside V. A predicate confined by V is also confined by a larger set. Predicate confinement is also closed under predicate operators. Formally:

$$V \subseteq W\ \wedge\ p\ \mathbf{conf}\ V\ \Rightarrow\ p\ \mathbf{conf}\ W \tag{2}$$

$$p, q\ \mathbf{conf}\ V\ \Rightarrow\ (p \vee q\ \mathbf{conf}\ V) \wedge (\neg p\ \mathbf{conf}\ V) \tag{3}$$

2.2 Capturing Temporary Non-interference

We introduce a relation preserves to capture temporary non-interference by a component on a given set of variables. Note that the *temporariness* plays an important role because the set of variables un-interfered by a program typically changes during the execution of the program. We write $p \vdash Q$ preserves $V|q$, stating that once p holds Q will maintain p and also preserve the value of variables in V, and that this period of non-interference is ended by establishing q. Formally:

Definition 2.1 : PRESERVES
$J, p \vdash Q$ preserves $V|q\ =\ (\forall \alpha, p' : \alpha \in Q \wedge p'\ \mathbf{conf}\ V : \{J \wedge p \wedge p'\}\ \alpha\ \{(p \wedge p') \vee q\})$
□

The α in the formula range over the actions of Q. We have also added an extra parameter J which is intended to be a stable predicate of Q. This is just a predicate which cannot be falsified by any action of Q. The J parameter is not an essential addition, but helps in the presentation later. Since actions are assumed to be terminating, it does not matter whether the Hoare triple above means partial or total correctness. Here is an example of a preserves property of the RTU component in Figure 4:

$$I_{\mathsf{RTU}}, r \neq [] \ \vdash\ \mathsf{RTU}\ \mathsf{preserves}\ \{r\}|\mathsf{false} \tag{4}$$

The variable r is an 'output port' of RTU. The value [] represents an empty port. So the above states that if r contains something, the RTU promises not to overwrite its value. I_{RTU} is just some invariant of the RTU.

There are some special cases of the preserves relation. The property $J, \mathsf{true} \vdash Q$ preserves $V|q$ states that the only way Q can interfere on V is by establishing q. The property $J, \mathsf{true} \vdash Q$ preserves $V|\mathsf{false}$, which we also write as simply $J \vdash Q$ preserves V, states that Q never interferes with V. This is the same as saying that Q has no write access to V, or in other words V only contains local variables of other components.

Theorems 2.2 and 2.3 below are properties of preserves which will be used later. The first follows easily from the definition of preserves , the second follows from the definition and properties (2) and (3) of confinement.

Theorem 2.2 :

$$\frac{J, p \vdash P \text{ preserves } V|q}{J \wedge J', p \vdash P \text{ preserves } V|q}$$

Theorem 2.3 :

$$\frac{V \subseteq W \ , \ J, p \vdash P \text{ preserves } W|q}{J, p \vdash P \text{ preserves } V|q}$$

□

2.3 Properties and Composition

In UNITY temporal properties of programs are expressed in terms of unless, ensures, and $\mapsto$ (leads-to) operators. Their meaning is quite standard –see for example [3]. However we will have to extend these operators to fit in our theory. For example now we would like to write $P, V, J \vdash p \mapsto q$ to express that P's progress from p to q is sensitive to external changes to variables in V, but not to external changes to non-V variables. The latter is in many cases too restrictive for P: it is more realistic to limit this 'insensitivity' to (external) changes to a given set of states. This constraining set of states is specified by the predicate parameter J in the specification. Because all changes are contained within J, we can expect that both the component and its environment preserve J. Or, in other words, J is stable.

Here is now how we define unless and ensures:

Definition 2.4 : STABLE, UNLESS, AND ENSURES

$$\begin{array}{lcl} P \vdash \mathsf{stable}\ J & = & (\forall \alpha : \alpha \in P : \{J\}\ \alpha\ \{J\}) \\ P, V, J \vdash p \text{ unless } q & = & (P \vdash \mathsf{stable}\ J)\ \wedge\ (p, q \textbf{ conf } V) \wedge \\ & & (\forall \alpha : \alpha \in P : \{J \wedge p \wedge \neg q\}\ \alpha\ \{p \vee q\}) \\ P, V, J \vdash p \text{ ensures } q & = & (P, J, V \vdash p \text{ unless } q)\ \wedge\ (\exists \alpha : \alpha \in P : \{J \wedge p \wedge \neg q\}\ \alpha\ \{q\}) \end{array}$$

□

Note that relative to [3] we add an additional constraint to unless and ensures requiring p and q to be confined to V. Hence, both p and q are insensitive to changes to variables outside V, and hence the described behavior is also insensitive to those changes. Note also that we only require the J parameter to be stable. In many cases J is actually also an invariant, but there are situations, like when composing P with a component that can only support a weaker invariant, where we really need to consider predicates which becomes stable dy-

Let *Rel* be either unless, ensures, $\mapsto$, or until.

Theorem 2.5 : PRE-STRENGTHENING AND POST-WEAKENING

$$\frac{P, V, J \vdash q \mapsto r \ , \ J \vdash p \Rightarrow q \ , \ J \vdash r \Rightarrow s \ , \ p, s \ \mathbf{conf} \ V}{P, V, J \vdash p \mapsto s}$$

The r-argument of unless or ensures can also be weakened as above, but the q-argument cannot be strengthened though it can be replaced by p if $J, V \vdash p = q$ (the infamous Substitution Law).

Theorem 2.6 : PROG.-SAFETY-PROG.

$$\frac{P, V, J \vdash p \ Rel \ q \qquad P, V, J \vdash r \ \text{unless} \ s}{P, V, J \vdash (p \wedge r) \ Rel \ (q \wedge r) \vee s}$$

Theorem 2.7 :

$$\frac{J \vdash q' \Rightarrow q \qquad P, V, J \vdash (p \wedge \neg q') \ Rel \ q}{P, V, J \vdash p \ Rel \ q}$$

Theorem 2.8 : UNL. COMPOSITION

$$\frac{P, V, J \vdash p \ \text{unless} \ q \qquad Q, V, J \vdash p \ \text{unless} \ q}{P [\!] Q, V, J \vdash p \ \text{unless} \ q}$$

Theorem 2.9 : ENS. COMPOSITION

$$\frac{P, V, J \vdash p \ \text{ensures} \ q \qquad Q, V, J \vdash p \ \text{unless} \ q}{P [\!] Q, V, J \vdash p \ \text{ensures} \ q}$$

Figure 2: Some standard UNITY laws expressed in the new logic.

namically rather than all the way from the start (invariant); however, we will retain the habit of calling J an *invariant*.

And here is how $\mapsto$ is now defined:

Definition 2.10 : LEADS-TO

For all P, V, J, the relation $(\lambda\, p\ q.\ P, V, J \vdash p \mapsto q)$ is defined as the smallest relation $\rightarrow$ satisfying:

Lifting:	if $P, V, J \vdash p$ ensures q then $p \rightarrow q$.
Transitivity:	if $p \rightarrow q$ and $q \rightarrow r$ then $p \rightarrow r$.
Disjunctivity:	let W be non-empty; if for all $i \in W$ we have $p_i \rightarrow q$ then $(\exists i : i \in W : p_i) \rightarrow q$.

□

Notice that $P, V, J \vdash p \mapsto q$ actually specifies progress from $J \wedge p$ to q. A derived operator which we will use later is the so-called until:

$$P, V, J \vdash p \ \text{until} \ q \quad = \quad P, V, J \vdash p \mapsto q \ \wedge \ P, V, J \vdash p \ \text{unless} \ q \qquad (5)$$

All UNITY laws in [3] can be lifted to apply to the new operators –with small adjustments. Some of those laws are displayed in Figure 2. A complete listing of can be found on-line at `www.cs.uu.nl/~wishnu`.

Figure 3 shows some of the new laws. The Locality Lift Law states that a property which is sensitive to external changes on variables in V is also sensitive to external changes on any set that includes V. The J-strengthening Law allows us to strengthen the J-parameter; weakening is however not allowed. The J parameter of $\mapsto$ is also not disjunctive. The J-Leftshift and R-Shift laws are

In the following theorems Rel represents either unless , ensures , or $\mapsto$.

Theorem 2.11 : LOCALITY LIFT

$$\frac{P, V, J \vdash p \; Rel \; q}{P, V, J \cup W \vdash p \; Rel \; q}$$

Theorem 2.12 : J-STRENGTHEN

$$\frac{P \vdash \text{stable } J', \quad J' \vdash J, \quad P, V, J \vdash p \; Rel \; q}{P, J', V \vdash p \; Rel \; q}$$

Theorem 2.13 : J-LEFTSHIFT

$$\frac{P, V, J \vdash (J' \wedge p) \; Rel \; q \qquad P \vdash \text{stable } J \wedge J' \;,\; p \textbf{ conf } V}{P, V, J \wedge J' \vdash p \; Rel \; q}$$

Theorem 2.14 : J-RIGHTSHIFT

$$\frac{P, V, J \wedge J' \vdash p \; Rel \; q \qquad P \vdash \text{stable } J \;,\; J' \textbf{ conf } V}{P, V, J \vdash (J' \wedge p) \; Rel \; (J' \wedge q)}$$

Theorem 2.15 : SINGH

$$\frac{\begin{array}{l} P, V, J \vdash p \; Rel \; q \\ Q \vdash \text{stable } J \\ a, b \textbf{ conf } V \cup W \\ J, a \vdash Q \text{ preserves } V|b \end{array}}{\begin{array}{l} P[\!]Q, V \cup W, J \\ \quad \vdash (p \wedge a) \; Rel \; ((q \wedge a) \vee \neg a \vee b) \end{array}}$$

Theorem 2.16 : SCHEDULING

$$\frac{\begin{array}{l} P, V, J \vdash a \wedge p \; Rel \; q \\ P, V, J \vdash a \text{ unless } q \\ Q \vdash \text{stable } J \\ J, a \vdash Q \text{ preserves } V|\text{false} \end{array}}{P[\!]Q, V, J \vdash a \wedge p \; Rel \; q}$$

Theorem 2.17 : TRANSPARENCY

$$\frac{P, V, J \vdash p \; Rel \; q \qquad Q \vdash \text{stable } J, \quad J \vdash Q \text{ preserves } V}{P[\!]Q, V, J \vdash p \; Rel \; q}$$

Theorem 2.18 : UNTIL COMPOSITION

$$\frac{P, V, J \vdash p \text{ until } q \qquad Q \vdash \text{stable } J, \quad J, p \vdash Q \text{ preserves } V|q}{P[\!]Q, V, J \vdash p \text{ until } q}$$

Figure 3: Composition Laws

used to shift part of the J parameter in and out of the 'precondition' (the p) in $p \; Rel \; q$.

In Sigh Law, the $V, J, a \vdash Q$ preserves $V|b$ condition states that Q will temporarily cease its interference on V as soon as $J \wedge a$ holds, and that this will last at least until b holds. The law states that during this period of non-interference, if it lasts long enough, P's local property of the form $p \; Rel \; q$ can be preserved in the composition $P[\!]Q$. If the period prematurely ends, then at least we know that b should hold.

The Scheduling Law is a very useful corollary of Sigh Law. Suppose a component P can behave as $p \; Rel \; q$ and that this property is only sensitive to external changes to variables in V. Moreover, at the start of this behavior P sets a 'flag' a to indicate its wish to realize $p \; Rel \; q$ uninterfered. This flag stays high until q holds. If Q promises to suspend interference on V as long as a is high, the law states that P's property $p \; Rel \; q$ will be respected by Q. The Until Composition Law is just a special case of the Scheduling Law. The Transparancy Law is another corollary of the Singh Law, stating that properties of P sensitive to external changes on variables in V are respected by any partner Q that does not write to V. This, for example, is trivially the case if V only contains P's local variables.

The Singh Law is a fundamental composition law in our theory. The law appears first in Singh's unpublished June-89 Notes on UNITY [14]. This work did not get much support however. There were problems: the set up was not formal enough and some technicalities are quite subtle and may render the law

inconsistent if not treated properly [9]; and the proto-form of the law used a fixed parameter V, which severely restricts its expressibility. The application is also not explored enough (for example the Scheduling Law was not identified). We reinvent the theory, and manage to overcome all those problems challenging Singh's earlier attempt.

3 Fault Tolerance by Exception Handling

Exception is a familiar fault tolerance technique. Typically, exceptions are handled by a separate component which may have the capability to override some part of the main system. We want to formalize this and customize the basic theory of the previous section to reason about composition of components in exception-handling based systems.

For the rest of this paper we simply write '*fault tolerant system*' rather than the mounth-full '*exception-handling based fault tolerant system*'.

3.1 How it Works

Although a system can internally generate faults, it is useful to think of faults as being externally generated. Firstly because external faults are less controllable, and hence we assume for the worse. Secondly, it enables us to concentrate on analyzing the system's fault-free behavior, which usually is the main concern of the system. To test a system's actual reaction to faults, we can compose it with an *adversary* that actually injects faults. The resulting behavior can be calculated using the composition theory.

Consider now a fault tolerant system FTS, consisting of a main component, mainly responsible for realizing whatever the desired behavior of FTS in the absence of faults (so-called the *main functions* of FTS), and a fault handler. The fault tolerance strategy is as follows:

1. The main component is responsible for detecting faults.
2. When a fault is detected, the main component suspends those activities which may be affected by the fault and throws an exception.
3. Throwing an exception activates the fault handler.
4. When the fault handler has finished, it awakes the suspended activities of the main component and suspends its own activities that may interfere with the main component.

Note that to allow a maximum degree of parallelism we specifically require that suspending a component *only suspends relevant activities* –a more naive strategy would be to simply suspend the whole component.

To give a better idea, Figure 4 shows the code of a simple *Remote Terminal Unit* (RTU) used to monitor the state of a number of water gates and sensors and to drive the gates in a flood control system. What RTU does is to repeatedly:

1. wait for a command arriving at port `c`.
2. in two parallel blocks, rearrange the gates according to the command and read the sensors.
3. send the gates and sensors data out via port `r`.

The program `RTU` has three components: `Main`, `FH0`, and `FH1`. `Main` is the component implementing the actual task of `RTU`. `FH0` is level 0 fault handler, and `FH1` is level 1 fault handler. The program detects and handles three kinds of faults: gate failure, sensor failure, and memory corruption. See the table below. Faults are multi levelled: higher level faults can disrupt not only the main component, but also lower level fault handlers.

fault	level	test	exception raised	handling
gate failure	0	`gate[i] =gtStuck`	`gtStuckExc`	reset the gates
sensor failure	0	`sensor[i] =senDead`	`senDeadExc`	log the event
mem. corruption	1	`checkMem() =CheckSumErr`	`memCorruptExc`	reset FTS1

The code is in some C-like language. The symbol `||` denotes parallel composition. Throwing (signaling or raising) an exception is done by the `THROW e` statement, which simply switches `e` to true. From that point on a fault handler listening to `e` can become active. The statement `SUSPEND P` suspends process `P` and all its internal concurrent processes. `RELEASE e,P,Q,...` lower the exception again and at the same time enables processes P, Q, and others in the list. So, for example, the code of `FTS1` contains a process (parallel to `Main` and `FH0`) modelling some regular check on the checksum of the memory segment on which `Main` and `FH0` run. A checksum error would indicate memory corruption, in which case `Main` and `FH0` have to be reset. This is coded by throwing `memCorruptExc` exception, which will activate the fault handler `FH1`, and suspend `Main` and `FH0`. Translation to UNITY is quite standard. The program in Figure 1 shows the translation of component `FH0` to UNITY. Notice that the variables `Ppc` and `Qpc` are program counters used to encode sequencing of `FH0`'s actions. The variables `Prun` and `Qrun` guard the actions of `FH0` and can be switch on and off to resume or suspend some part of `FH0`.

3.2 On Terminology

Some literature distinguishes between fault, error, failures, and so on. It is imaginable that people have different ideas about what they mean, so before we go on with formalization we describe first what we mean with 'fault'. Here, we define a fault as *a set of states* of a program that the developer would consider as *abnormal*. Note that such a set of states can be defined by a predicate. For example, in the program in Figure 4 the predicate ($\exists i : 0 \leq i < 4 :$ `gate[i]` $=$ `gtStuck`) specifies a fault in which one of the gates controlled by RTU becomes stuck. A fault e has *occurrences* or *instances*. An occurrence of e is just the

```
PROG RTU::
VAR PUBLIC  c,r : port ;
VAR gtStuckExc, senDeadExc, memCheckSumErrExc: bool ;
    gate, sensor: array of int ;

DECL
    {-------------------------------------------------------------------------}
    PROG Main::
    VAR i,command : int
       [ REPEAT FOREVER
          [ command := fetch c ;
            PROG P::
              [ gate := setGate command ;
                FOR (i:=0, i<4, i++)
                IF gate[i]=gtStuck THEN [THROW gtStuckExc; SUSPEND P ]
            ||
            PROG Q::
              [ sensor := checkSensor() ;
                FOR (i:=0, i<10, i++)
                IF sensor[i]=senDead THEN [THROW senDeadExc; SUSPEND Q ];
            r := send (gate,sensor) ] ]
    {-------------------------------------------------------------------------}
    PROG FH0::
      [ [IF gtStuckExc THEN [gate := resetGate();
                             RELEASE gtStuckExc, P ]]
        ||
        [IF senDeadExc THEN [append "sensor dead" to log ;
                             RELEASE senDeadExc, Q ] ] ]
    {-------------------------------------------------------------------------}
    PROG FTS1::
      [ Main ||
        FH0  ||
        REPEAT FOREVER
           [ IF checkMem()=CheckSumErr
                THEN [ SUSPEND Main; SUSPEND FH0 ;
                       SUSPEND FTS1 RAISE memCorruptExc ] ]
    {-------------------------------------------------------------------------}
    PROG FH1::
      [ IF memCorruptExc
           THEN [ ...do something ;
                  RELEASE memCorruptExc, Main, FH0, FTS0 ] ]
    {-------------------------------------------------------------------------}

ENDDECL {-- of RTU --}

[FTS1 || FH1]
```

Figure 4: Example: RTU

occurrence of one of the states in e at some point during an execution of the system.

A fault e is said to be *removed* when the system is brought to a state outside e. As a fault may not be handled immediately, it may cause further inconsistencies. Consequently, the removal of a fault does not necessarily mean that the system is again in a correct state. We do not define the goal of fault handling as the recovery of disrupted behavior because there are situations where recovery is either unnecessary, impossible, or too costly. Rather, we broadly define it as a system's reaction to the fault to bring itself to some defined states. Recovery is then just a special kind of fault handling. For example, what the RTU in Figure 4 does in the case of sensor failure is to simply log the occurrence of the fault. It does not even try to remove it. Obviously the designer hopes that the fault will corrected once reported. In many situations, this is all that we can do.

3.3 Single Fault System

Let us first consider a system capable of only detecting and handling a single fault. This is simpler and enables us to focus on the composition of the main component and the fault handler. We will show later how the result may be extended to a multiple faults system.

Let in the sequel FTS be a fault tolerant system, consisting of a main component Main and a fault handler FH. The fault handler handles a single fault characterized by the predicate fault. When Main detects an occurrence of fault, it establishes a predicate detect. Establishing detect can be thought of as modelling the exception event. The final result of handling fault is modelled by a predicate handled. Recall that we do not require fault handling to remove the fault, but if it does then we have handled $\Rightarrow \neg$fault.

Let in the sequel J^{Main} and J^{det} be two invariants of Main and J^{FH} be an invariant of FH. J^{Main} is needed by Main to perform its main function; J^{det} is needed by Main to do fault detection; and J^{FH} is needed by FH to do fault handling. Note that fault may destroy J^{Main}. The detection and handling mechanism must on the other hand be strong enough to withstand the kind of fault they are supposed to handle. So, fault is incapable of destroying J^{det} and J^{FH}. It also follows that J^{det} and J^{FH} should be on their own invariants of the whole FTS, since if J^{Main} fails because of fault, then FTS should still be able to detect and handle fault. So, we assume:

(a)	Main $\vdash$ stable J^{Main}	(c)	FH $\vdash$ stable J^{FH}	(d)	FTS $\vdash$ stable J^{Main}
(b)	Main $\vdash$ stable J^{det}			(e)	FTS $\vdash$ stable $J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$

(6)

Let in the sequel V^{det} and V^{hand} be the sets of variables read or written during, respectively, the detection and handling of fault. Since detection is Main's task and handling is FH's task, we have $V^{\mathsf{det}} \subseteq \mathbf{var}(\mathsf{Main})$ and $V^{\mathsf{hand}} \subseteq \mathbf{var}(\mathsf{FH})$. We assume that fault, detect **conf** V^{det} and detect, handled **conf** V^{hand}, and also $J^{\mathsf{Main}}, J^{\mathsf{det}}, J^{\mathsf{FH}}$ **conf var**(FTS).

3.3.1 Fault Detection

When fault occurs, eventually the system FTS must detect it. So its specification has the form of $\mathsf{FTS} \vdash \mathsf{fault} \mapsto \mathsf{detect}$. The following theorem shows how this specification can be factorized.

Theorem 3.1 :

(1) $\mathsf{Main} \vdash \mathsf{stable}\ J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$, $\mathsf{FH} \vdash \mathsf{stable}\ J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$

(2) $\mathsf{Main}, V^{\mathsf{det}}, J^{\mathsf{det}} \vdash \mathsf{fault} \mapsto \mathsf{detect}$

(3) $J^{\mathsf{FH}}, \neg\mathsf{detect} \vdash \mathsf{FH}\ \mathsf{preserves}\ V^{\mathsf{det}} | \mathsf{false}$

$\mathsf{FTS}, V^{\mathsf{det}}, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{fault} \mapsto \mathsf{detect}$

□

The conclusion of the theorem states that when fault occurs, FTS will eventually raise detect. Condition (1) essentially states that each component respects the other's invariant (see the remark following Theorem 3.2). Condition (2) says that the progress to detect is actually Main's local property. Condition (3) states that while detect does not hold, FH will not interfere with Main's progress to detect. Notice also that condition (3) does not say that the fault handler should suspend *all* activity: only activities that may update variables in V^{det} need to be suspended.

We can easily prove Theorem 3.1 using the general composition theory of Section 2. We will first prove a few lemmas, since we are going to use them again later.

Theorem 3.2 :

(1) $P \vdash J_P \wedge J_Q$, $Q \vdash J_P \wedge J_Q$ (3) $a, b\ \mathbf{conf}\ V \cup W$

(2) $P, V, J_P \vdash p\ Rel\ q$ (4) $J_Q, a \vdash Q\ \mathsf{preserves}\ (V \cup W) | b$

$P[\!]Q, V \cup W, J_P \wedge J_Q \vdash (p \wedge a)\ Rel\ (q \vee \neg a \vee b)$

where Rel be either unless, ensures, $\mapsto$, or until. □

The above is just a variant of the Singh Law. J_P and J_Q can be thought as the invariants of P and Q respectively. $J_P \wedge J_Q$ is then a combined invariant of the composite $P[\!]Q$. Condition (1) of the theorem states that this combined invariant holds in each component (P and Q). Since a component can be expected to maintain its own invariant, this condition essentially says that each component respects the other's invariant. Condition (2) states that we have some local property $p\ Rel\ q$ of P. Conditions (3) are (4) are as in the Sigh Law, stating some confinement and temporary non-interference conditions of Q. Finally the conclusion essentially states that as long as a and $\neg b$ hold the composite will preserve the behavior $p\ Rel\ q$ (also quite the same as in the Sigh Law).

Proof: (of Theorem 3.2)

$P[\!]Q, V \cup W, J_P \wedge J_Q \vdash p \wedge a\ Rel\ q \vee \neg a \vee b$

$\Leftarrow$ { Sigh Law (Theorem 2.15) }

$(P, V \cup W, J_P \wedge J_Q \vdash p\ Rel\ q)\ \wedge\ (a, b\ \mathbf{conf}\ V \cup W)\ \wedge$
$(Q \vdash \mathsf{stable}\ J_P \wedge J_Q)\ \wedge\ (J_P \wedge J_Q, a \vdash Q\ \mathsf{preserves}\ (V \cup W) | b)$

$\Leftarrow$ { Theorem 2.2 }

$(P, V \cup W, J_P \wedge J_Q \vdash p \; Rel \; q) \;\wedge\; (a, b \; \mathbf{conf} \; V \cup W) \;\wedge$
$(Q \vdash \mathsf{stable} \; J_P \wedge J_Q) \;\wedge\; (J_Q, a \vdash Q \; \mathsf{preserves} \; (V \cup W)|b)$

$\Leftarrow$ { Locality Lift (Theorem 2.11) }

$(P, V, J_P \wedge J_Q \vdash p \; Rel \; q) \;\wedge\; (a, b \; \mathbf{conf} \; V \cup W) \;\wedge$
$(Q \vdash \mathsf{stable} \; J_P \wedge J_Q) \;\wedge\; (J_Q, a \vdash Q \; \mathsf{preserves} \; (V \cup W)|b)$

$\Leftarrow$ { J-Strengthen (Theorem 2.12 }

$(P, V, J_P \vdash p \; Rel \; q) \;\wedge\; (P \vdash \mathsf{stable} \; J_P \wedge J_Q) \;\wedge\; (Q \vdash \mathsf{stable} \; J_P \wedge J_Q) \;\wedge$
$(J_Q, a \vdash Q \; \mathsf{preserves} \; (V \cup W)|b) \;\wedge\; (a, b \; \mathbf{conf} \; V \cup W)$

□

Below are two lemmas which we will use later. The left one is a special case of Theorem 3.2; the other can be proven from Theorem 2.18 in the way similar to Theorem 3.2.

Theorem 3.3 :

$$\frac{(1)\;\; P \vdash J_P \wedge J_Q \;,\; Q \vdash J_P \wedge J_Q \quad (2)\;\; P, V, J_P \vdash p \; Rel \; q \quad (3)\;\; J_Q \vdash Q \; \mathsf{preserves} \; V}{P[\![Q, V, J_P \wedge J_Q \vdash p \; Rel \; q}$$

$$\frac{(1)\;\; P \vdash J_P \wedge J_Q \;,\; Q \vdash J_P \wedge J_Q \quad (2)\;\; P, V, J_P \vdash p \; \mathsf{until} \; q \quad (3)\;\; J_Q, p \vdash Q \; \mathsf{preserves} \; V|q}{P[\![Q, V, J_P \wedge J_Q \vdash p \; \mathsf{until} \; q}$$

□

Now we can easily prove Theorem 3.1:
By Theorem 2.7 it suffices to prove that $\mathsf{FTS}, V^{\mathsf{det}}, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{fault} \wedge \neg \mathsf{detect} \mapsto \mathsf{detect}$ follows from the assumptions of Theorem 3.1. But this is simply an instance of Theorem 3.2.
□

3.3.2 Fault Handling

After detecting a fault, FTS should eventually handle it. The specification has the form of $\mathsf{FTS} \vdash \mathsf{detect} \mapsto \mathsf{handled}$. This is how to factorize it:

Theorem 3.4 :

$$\frac{(1)\;\; \mathsf{Main} \vdash \mathsf{stable} \; J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \;,\; \mathsf{FH} \vdash \mathsf{stable} \; J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \quad (2)\;\; \mathsf{FH}, V^{\mathsf{hand}}, J^{\mathsf{FH}} \vdash \mathsf{detect} \; \mathsf{until} \; \mathsf{handled} \quad (3)\;\; J^{\mathsf{det}} \wedge J^{\mathsf{FH}}, \mathsf{detect} \vdash \mathsf{Main} \; \mathsf{preserves} \; V^{\mathsf{hand}}|\mathsf{false}}{\mathsf{FTS}, V^{\mathsf{hand}}, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{detect} \mapsto \mathsf{handled}}$$

□

As in the case of fault detection, Condition (1) states that each component respects the other's invariant. Condition (3) states that Main activities that may write to V^{hand} will be suspended as long as detect is high –again, this does not mean that Main should suspend *all* activities. Condition (2) says two things. In the first place, it is FH's task to actually perform the fault handling. Furthermore, handled is maintained until the fault handling is completed, which is necessary because otherwise Main can interfere with the FH before the latter finishes its task.

Proof: (of Theorem 3.4) It follows from the definition of until and Theorem 3.3. □

3.3.3 The Main Function

Main functions are the set of expected behavior of FTS in the absence of faults. In other words, they are simply the behavior of Main without the fault detection. They are expressed as a collection of properties of the form $\mathsf{Main}, A, J^{\mathsf{Main}} \vdash p\ Rel\ q$ where Rel is either unless, ensures, $\mapsto$, or until. Recall that A is a set of variables on which $p\ Rel\ q$ is sensitive to external change. Since the fault handler FH is also in FTS, its interference on A may destroy the main functions, and therefore we are interested in conditions that can prevent this.

If this interference occurs during FH's active time (detect high), it means a fault has occurred before and obviously the interference is part of FH's fault handling procedure. The net result of this interference has been formalized in Theorem 3.4, namely the establishment of the predicate handled. It remains to see how FH interferes on A during its 'suspended' state. Interferences can still happen because the conditions of Theorem 3.1 only constrain FH from interfering on V^{det}, and not on A. A possible (but not the weakest) solution is to strengthen this requirement to:

$$J^{\mathsf{FH}}, \neg\mathsf{detect} \vdash \mathsf{FH}\ \mathsf{preserves}\ \mathbf{var}(\mathsf{Main})|\mathsf{false} \tag{7}$$

stating that during its suspension FH will not interfere with Main at all and therefore can only perform internal computation. This yields the following theorem, stating the conditions required to preserve a main function, up to the point when *fault* is detected.

Theorem 3.5 :

(1) $\mathsf{Main} \vdash \mathsf{stable}\ J^{\mathsf{Main}} \wedge J^{\mathsf{FH}}$ (2) $\mathsf{FH} \vdash \mathsf{stable}\ J^{\mathsf{Main}} \wedge J^{\mathsf{FH}}$

(3) $A \subseteq \mathbf{var}(\mathsf{Main})$ (4) $\mathsf{Main}, A, J^{\mathsf{Main}} \vdash p\ Rel\ q$

(5) $J^{\mathsf{FH}}, \neg\mathsf{detect} \vdash \mathsf{FH}\ \mathsf{preserves}\ \mathbf{var}(\mathsf{Main})|\mathsf{false}$

$\mathsf{FTS}, A \cup V^{\mathsf{det}}, J^{\mathsf{Main}} \wedge J^{\mathsf{FH}} \vdash p\ Rel\ q \vee \mathsf{detect}$

□

Proof:

Notice first that since detect **conf** V^{det} and by (2) and (3) it follows that ¬detect **conf** $A \cup V^{\mathsf{det}}$. Now we derive:

$\mathsf{FTS}, A \cup V^{\mathsf{det}}, J^{\mathsf{Main}} \wedge J^{\mathsf{FH}} \vdash p\ Rel\ q \vee \mathsf{detect}$

$\Leftarrow$ { Theorem 2.7 }

$\mathsf{FTS}, A \cup V^{\mathsf{det}}, J^{\mathsf{Main}} \wedge J^{\mathsf{FH}} \vdash p \wedge \neg\mathsf{detect}\ Rel\ q \vee \mathsf{detect}$

$\Leftarrow$ { Theorem 3.2; the remark above }

$(\mathsf{Main}, A, J^{\mathsf{Main}} \vdash p\ Rel\ q)\ \wedge\ (\mathsf{Main} \vdash \mathsf{stable}\ J^{\mathsf{Main}} \wedge J^{\mathsf{FH}}) \wedge$
$(J^{\mathsf{FH}}, \neg\mathsf{detect} \vdash \mathsf{FH}\ \mathsf{preserves}\ A \cup V^{\mathsf{det}}|\mathsf{false})\ \wedge\ (\mathsf{FH} \vdash \mathsf{stable}\ J^{\mathsf{Main}} \wedge J^{\mathsf{FH}})$

$\Leftarrow$ { we have $A \subseteq \mathbf{var}(\mathsf{Main})$ and $V^{\mathsf{det}} \subseteq \mathbf{var}(\mathsf{Main})$; Theorem 2.3 }

$(\mathsf{Main}, A, J^{\mathsf{Main}} \vdash p\ Rel\ q)\ \wedge\ (\mathsf{Main} \vdash \mathsf{stable}\ J^{\mathsf{Main}} \wedge J^{\mathsf{FH}}) \wedge$
$(J^{\mathsf{FH}}, \neg\mathsf{detect} \vdash \mathsf{FH}\ \mathsf{preserves}\ \mathbf{var}(\mathsf{Main})|\mathsf{false})\ \wedge\ (\mathsf{FH} \vdash \mathsf{stable}\ J^{\mathsf{Main}} \wedge J^{\mathsf{FH}})$

□

3.3.4 Testing with Adversary

In the previous three subsubsections we have formally captured the interaction between the main component and the fault handler of FTS. It is true that occurrences of fault have been taken into account, but since neither component is assumed to be capable of generating fault, we cannot justify yet that the theorems presented so far reflect FTS's actual tolerance to faults. To do this we must subject FTS to a component (usually called *adversary*) that actually injects faults, and subsequently use our theory of composition to calculate the resulting behavior.

This approach has another advantage. Different faults may have different reactive behavior. Certain faults are *permanent*: they persist until removed. The handling of this kind of faults should include their removal, or else the handler will be reinvoked indefinitely. Some faults are *transient*: they are only present for some limited duration. Sometimes, a transient fault does not destroy the safety requirement of an FTS, though it may block some progress, in which case FTS can simply suspend Main until the fault is over, and then resume its operation. By treating an adversary as a component we have the freedom to construct different adversaries to model different faults' behavior, and subsequently calculate FTS' reaction.

To show how to deploy the theory, let us consider the transient kind of faults. Note that the detection and handling of a fault e may fail if the fault rapidly oscillates during a short period (*oscillation burst*). This oscillation can be in the form of a sequence $e, \neg e, e, ...$, or an internal oscillation among the states inside e (remember that a fault is defined as a set of states). For example, the gate failure in the RTU example in Figure 4 is identified with $(\exists i : 0 \leq i < 4 :$ `gate[i] = gtStuck`$)$. That is, a fault occurs if one of the gate is stuck. The program tries to detect this using a loop that scans the gates one by one. If the gates get stuck and unstuck in random order and at high speed –not a realistic situation, but hypothetically possible– then the loop may fail to 'catch' a stuck gate, and therefore fail to detect the fault.

To simplify the discussion let us just consider the correctness of FTS relative to a no-oscillation-burst scenario. Here is then the specification of our adversary:

$$\mathsf{true}, \neg\mathsf{fault} \vdash \mathsf{adv\ preserves}\ \mathbf{var}(\mathsf{FTS})|\mathsf{fault} \tag{8}$$

$$(\mathsf{adv} \vdash \mathsf{stable}\ J^{\mathsf{det}}) \ \wedge \ (\mathsf{adv} \vdash \mathsf{stable}\ J^{\mathsf{FH}}) \tag{9}$$

$$\mathsf{true}, \mathsf{nonOsci} \vdash \mathsf{adv\ preserves}\ V^{\mathsf{det}} \cup V^{\mathsf{hand}}|\mathsf{handled} \tag{10}$$

The first specification states that the adversary does not interfere with FTS except by establishing fault. In other words, the only purpose of the adversary is to inject faults. Note that this implicitly assumes that during the test against adv, FTS is treated as a closed system.

As said in the beginning of this section, although adv may destroy J^{Main}, the fault detection and handling mechanisms have to be strong enough to stay functioning. So, as stated by the second specification (9), the adversary is assumed to respect the invariants of fault detection and handling. Since the adversary *can* generate faults, (9) prevents the trivial implementation of fault detection

and handling that simply excludes faults by maintaining $J^{\mathsf{det}} \Rightarrow \neg\mathsf{fault}$. In the third specification nonOsci is a predicate characterizing moments in which no burst of fault causing oscillation in the values of V^{det} and V^{hand} occurs. The specification states that this no-oscillation period will last at least until FTS has finished handling fault. Notice that this does not restrain the adversary from corrupting other variables. In particular in the multiple faults scenario this means that while in no oscillation burst state for fault e, the adversary can generate other kinds of faults.

Now let us see how this kind of adversary influences our FTS:

Theorem 3.6 : Interference on the Main Function

(1) $\mathsf{FTS}, A \cup V^{\mathsf{det}}, J^{\mathsf{Main}} \wedge J^{\mathsf{FH}} \vdash p \; Rel \; q \vee \mathsf{detect}$
(2) $\mathsf{FTS} \vdash \mathsf{stable}\; J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$
(3) $\mathsf{adv} \vdash \mathsf{stable}\; J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$
(4) $\mathsf{true}, \neg\mathsf{fault} \vdash \mathsf{adv}$ preserves $\mathbf{var}(\mathsf{FTS})|\mathsf{fault}$

$\mathsf{FTS}[\![\mathsf{adv}, \mathbf{var}(FTS) J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash (J^{\mathsf{Main}} \wedge p) \; Rel \; (q \vee \mathsf{fault} \vee \mathsf{detect})$

□

Condition (1) is the same as the conclusion of Theorem 3.5, stating what we know about the main function after possible interference by FH. Condition (2) captures what is said earlier: $J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$ should on its own be an invariant of the FTS so that when J^{Main} is destroyed by fault its detection and handling does not fail too. Condition (3) and (4) are just the specifications of the adversary as in (8) and (9). The conclusion states essentially that we can expect the main function $p \; Rel \; q$ to be preserved when no fault nor exception (detect going high) occurs. The proof is not too difficult and, due to limited space, is left to the reader.

The following theorem states how the adversary can influence fault detection:

Theorem 3.7 : Interference on the Fault Detection

(1) $\mathsf{FTS}, V^{\mathsf{det}}, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{fault} \mapsto \mathsf{detect}$
(2) $\mathsf{adv} \vdash \mathsf{stable}\; J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$
(3) $\mathsf{true}, \mathsf{nonOsci} \vdash \mathsf{adv}$ preserves $V^{\mathsf{det}} \cup V^{\mathsf{hand}}|\mathsf{handled}$
(4) $\mathsf{nonOsci}$ **conf** Z
(5) $\mathsf{FTS}, \mathbf{var}(\mathsf{FTS}) \cup \mathbf{var}(\mathsf{adv}), \mathsf{true} \vdash \mathsf{nonOsci}$ unless handled

$\mathsf{FTS}[\![\mathsf{adv}, Z, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{fault} \wedge \mathsf{nonOsci} \mapsto \mathsf{detect} \wedge \mathsf{nonOsci} \vee \mathsf{handled}$

where $Z = V^{\mathsf{det}} \cup V^{\mathsf{hand}} \cup \mathbf{var}(\mathsf{adv})$. □

Condition (1) of the theorem is the same as the conclusion of Theorem 3.1, stating FTS's promise to detect faults. Condition (2) and (3) are just the specifications of the adversary as in (9) and (10). Condition (4) states that nonOsci is a predicate mentioning only adversary's variables and variables relevant to detection and handling of fault. Condition (5) is an addition requiring that FTS cannot influence the no-oscillation-burst period of adv. The conclusion of the theorem simply states that during a no-oscillation-burst period of adv, an occurring fault will be either detected or handled. The next theorem will states that when it is detected, then it will also be handled, and so we are done.

Proof: (of Theorem 3.7) First notice that by PSP law, the conclusion follows from the conjunction of the following:

$\mathsf{FTS}[\![\mathsf{adv}, V^{\mathsf{det}} \cup V^{\mathsf{hand}} \cup \mathbf{var}(\mathsf{adv}), J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$
$\vdash$
(a) $\mathsf{fault} \wedge \mathsf{nonOsci} \mapsto \mathsf{detect} \vee \neg\mathsf{nonOsci} \vee \mathsf{handled}$ and
(b) $\mathsf{nonOsci}$ unless $\mathsf{handled}$

The unless part follows from the 3^{rd} and 5^{th} conditions of Theorem 3.7. The progress part follows from the fact that $\mathsf{handled}$ **conf** V^{det} holds and from Theorem 3.2. □.

The next theorem states that during a no-oscillation-burst period of the adversary, fault handling by FTS can be completed.

Theorem 3.8 : Interference on the Fault Handling

(1) $\mathsf{FTS}, V^{\mathsf{hand}}, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{detect} \mapsto \mathsf{handled}$
(2) $\mathsf{adv} \vdash \mathsf{stable}\ J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$
(3) $\mathsf{true}, \mathsf{nonOsci} \vdash \mathsf{adv}$ preserves $V^{\mathsf{det}} \cup V^{\mathsf{hand}} | \mathsf{handled}$
(4) $\mathsf{nonOsci}$ **conf** $V^{\mathsf{det}} \cup V^{\mathsf{hand}} \cup \mathbf{var}(\mathsf{adv})$
(5) $\mathsf{FTS}, \mathbf{var}(\mathsf{FTS}) \cup \mathbf{var}(\mathsf{adv}), \mathsf{true} \vdash \mathsf{nonOsci}$ unless $\mathsf{handled}$

$\mathsf{FTS}[\![\mathsf{adv}, V^{\mathsf{det}} \cup V^{\mathsf{hand}} \cup \mathbf{var}(\mathsf{adv}), J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{detect} \wedge \mathsf{nonOsci} \mapsto \mathsf{handled}$

□

Except for the first condition, which states that FTS promises to do fault handling, the other conditions are the same as in Theorem 3.7, so we will not repeat their explanation. The proof of the above theorem is quite similar to that of Theorem 3.7.

Now, combining Theorems 3.7 and 3.8 we get the fault tolerance property what we expect, namely:

$$\mathsf{FTS}[\![\mathsf{adv}, V^{\mathsf{det}} \cup V^{\mathsf{hand}} \cup \mathbf{var}(\mathsf{adv}), J^{\mathsf{det}} \wedge J^{\mathsf{FH}}, \vdash \mathsf{fault} \wedge \mathsf{nonOsci} \mapsto \mathsf{handled} \tag{11}$$

stating that during non-burst time of the adversary, an occurring fault can be completely handled by FTS.

4 One-Level, Multiple Faults Handling

The previous section shows a theory for composing a fault tolerant system from a main and a fault handling component. We have also shown how the general composition theory can be used to predict how a system actually reacts to an adversary. However, so far we only discuss systems capable of dealing with only one fault, which is not too realistic; in practice we are typically confronted with multiple faults, each requiring a different way of detection and handling. Fortunately, it is not too difficult to generalize the single fault theory to the multiple faults case.

We will now assume a set F of fault names or codes. For each fault code $e \in F$ the fault that is associated with e is denoted by fault_e. Remember that a fault is here defined as a set of states which are considered as abnormal. This can be described by a predicate. So $\{\mathsf{fault}_e | e \in F\}$ is a set of faults, and F is

like a set of indices. Though obviously there is a distinction between a fault and its code, in the sequel we will often use them interchangeably.

The detection of fault $e \in F$ is denoted by detect_e, and the goal of its handling by $\mathsf{handled}_e$. The invariants needed to detect and handle e are denoted by, respectively, J_e^{det} and J_e^{FH}. The sets V_e^{det} and V_e^{hand} denote the set of variables possibly read or written during, respectively, the detection and handling of e. We also define:

$$\begin{array}{|lcl|lcl|}
\hline
\mathsf{fault} & = & (\exists e : e \in F : \mathsf{fault}_e) & J^{\mathsf{det}} & = & (\forall e : e \in F : J_e^{\mathsf{det}}) \\
\mathsf{detect} & = & (\exists e : e \in F : \mathsf{detect}_e) & J^{\mathsf{FH}} & = & (\forall e : e \in F : J_e^{\mathsf{FH}}) \\
\mathsf{handled} & = & (\exists e : e \in F : \mathsf{handled}_e) & V^{\mathsf{det}} & = & (\cup e : e \in F : V_e^{\mathsf{det}}) \\
\hline
\end{array} \quad (12)$$

It is possible, that when the system is handling an occurrence of fault d an occurrence of another fault e causes the handling of d to fail. This requires multi level fault handling. This will be discussed later, for now we exclude this possibility. This implies that faults in F cannot destroy each other's detection and handling invariants.

The conclusions of Theorems 3.5, 3.1, and 3.4 are actually what we can consider as the specifications of an single fault FTS (since they state FTS' fault tolerance commitment). These specifications must be adapted to reflect multiple faults. Here are the new ones. For all fault $e \in F$:

$$\mathsf{Detection}: \quad \mathsf{FTS}, V_e^{\mathsf{det}}, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{fault}_e \mapsto \mathsf{detect}_e \quad (13)$$

$$\mathsf{Handling}: \quad \mathsf{FTS}, V_e^{\mathsf{hand}}, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{detect}_e \mapsto \mathsf{handled}_e \quad (14)$$

$$\mathsf{MainFunction}: \quad \mathsf{FTS}, A \cup V^{\mathsf{det}}, J^{\mathsf{Main}} \wedge J^{\mathsf{FH}} \vdash p \; Rel \; q \vee \mathsf{detect} \quad (15)$$

The first two specifications specify the detection and handling of each kind of fault. Notice that the properties are specified using to the combined invariant J^{det} and J^{FH} rather than $J_e^{\mathsf{det}} \wedge J_e^{\mathsf{FH}}$ separately. The latter would have been sufficient, but since we have assumed that faults in F do not destroy each other's invariants, we can just as well use the combined invariant –it yields simpler formulas. The third specification above specifies a main function with possible interruptions when faults are detected. Notice that the form is the same as in the single fault case (Theorem 3.5), except that now fault and detect are the disjunctions of all, respectively, fault_e and detect_e. In fact we can view a multiple faults system as an ordinary single fault system, but with additional specifications (13) and (14).

To factorize the new specifications we need to generalize Theorems 3.5, 3.1, and 3.4 (these specify how to do factorization in the single-fault scenario). Since (15) has the same form as in the single fault system, we can simply use Theorem 3.5 to factorize it. Subsequently, By instantiating $V^{\mathsf{det}}, V^{\mathsf{hand}}, \mathsf{fault}, \mathsf{detect}$, and handled to $V_e^{\mathsf{det}}, V_e^{\mathsf{hand}}, \mathsf{fault}_e, \mathsf{detect}_e$, and $\mathsf{handled}_e$ we can reuse Theorems 3.1 and 3.4 to factorize the other two specifications. Merging them together, we obtain the following theorem:

Theorem 4.1 : Factorizing Multiple Faults
FTS satisfies the specifications (13), (14), and (15) provided it meets the following conditions:

(1) $\mathsf{Main} \vdash \mathsf{stable}\ J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$, $\mathsf{FH} \vdash \mathsf{stable}\ J^{\mathsf{det}} \wedge J^{\mathsf{FH}}$
(2) $\mathsf{Main} \vdash \mathsf{stable}\ J^{\mathsf{Main}} \wedge J^{\mathsf{FH}}$, $\mathsf{FH} \vdash \mathsf{stable}\ J^{\mathsf{Main}} \wedge J^{\mathsf{FH}}$
(3) $\mathsf{Main}, V_e^{\mathsf{det}}, J^{\mathsf{det}} \vdash \mathsf{fault}_e \mapsto \mathsf{detect}_e$
(4) $J^{\mathsf{det}} \wedge J^{\mathsf{FH}}, \mathsf{detect}_e \vdash \mathsf{Main}$ preserves $V_e^{\mathsf{hand}} | \mathsf{false}$
(5) $\mathsf{Main}, A, J^{\mathsf{Main}} \vdash p\ Rel\ q$
(6) $\mathsf{FH}, V_e^{\mathsf{hand}}, J^{\mathsf{FH}} \vdash \mathsf{detect}_e$ until $\mathsf{handled}_e$
(7) $J^{\mathsf{FH}}, \neg\mathsf{detect}_e \vdash \mathsf{FH}$ preserves $V_e^{\mathsf{det}} | \mathsf{false}$
(8) $J^{\mathsf{FH}}, \neg\mathsf{detect} \vdash \mathsf{FH}$ preserves $\mathbf{var}(\mathsf{Main}) | \mathsf{false}$
(9) $A \subseteq \mathbf{var}(\mathsf{Main})$

□

Notice that the theorem gives a factorization which is more general than the simplistic sequential handling of faults. In the latter approach, when fault e occurs when a detection or handling process of another fault d is still at work, then the detection and handling of e will have to wait. In the above factorization, the only safety conditions that may impose sequencing are (4) and (7). However, (4) simply states that Main's access to V_e^{det} is blocked when the 'exception' detect_e has been raised, and else Main is allowed to detect e (for which it needs access to V_e^{det}) regardless whether or not it is busy with detecting or handling other faults. Similar reasoning applies to (7). So, concurrent detection and handling of different faults is possible. Also: even though (3) and (6) delegate fault detection and handling of all fault types to respectively Main and FH, by the above argument we are allowed to partition Main and FH into smaller components, each detecting and handling a specific partition of F.

We still have one remaining question: how would FTS now react to an adversary? Obviously we need a different adversary to generate multiple faults. The specification now reads:

$$\mathsf{true}, \neg\mathsf{fault} \vdash \mathsf{adv}\ \text{preserves}\ \mathbf{var}(\mathsf{FTS}) | \mathsf{fault} \tag{16}$$

$$(\mathsf{adv} \vdash \mathsf{stable}\ J^{\mathsf{det}}) \ \wedge \ (\mathsf{adv} \vdash \mathsf{stable}\ J^{\mathsf{FH}}) \tag{17}$$

$$\mathsf{true}, \mathsf{nonOsci}_e \vdash \mathsf{adv}\ \text{preserves}\ V_e^{\mathsf{det}} \cup V_e^{\mathsf{hand}} | \mathsf{handled}_e \tag{18}$$

The first two specifications are just the same as in the single fault adversary, except that the interpretation is slightly different. In particular, the first states that the only way the adversary can influence FTS is by generating a fault – it does not matter which one. In (18) $\mathsf{nonOsci}_e$ characterizes the periods in which the adversary does not repeatedly change the variables in $V_e^{\mathsf{det}} \cup V_e^{\mathsf{hand}}$. Theorem 3.6 states how a single fault adversary influences the main function. However, the conditions constraining adv in the theorem only refer to that part of adv's specifications that for single and multi faults adv remains the same. Consequently the theorem also applies to multi fault systems. Theorems 3.7 and 3.8, stating the adversary's influence on the fault detection and handling, can be reused using the same variables instantiation used to obtain Theorem 4.1. The following theorem summarizes this:

Theorem 4.2 : Reaction to Multiple Faults
If FTS satisfies the specifications (13), (14), and (15), and if adv satisfies (16), (17), and (18), then the composite FTS[]adv satisfies:

(1) $\mathsf{FTS}[\![\mathsf{adv}, \mathbf{var}(\mathsf{FTS}), J^{\mathsf{det}} \wedge J^{\mathsf{FH}}, \vdash (J^{\mathsf{Main}} \wedge p)\ Rel\ (q \vee \mathsf{fault} \vee \mathsf{detect})$

(2) $\mathsf{FTS}[\![\mathsf{adv}, Z, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{fault} \wedge \mathsf{nonOsci}_e \mapsto \mathsf{detect}_e \wedge \mathsf{nonOsci}_e \vee \mathsf{handled}_e$

(3) $\mathsf{FTS}[\![\mathsf{adv}, Z, J^{\mathsf{det}} \wedge J^{\mathsf{FH}} \vdash \mathsf{detect}_e \wedge \mathsf{nonOsci}_e \mapsto \mathsf{handled}_e$

where $Z = V_e^{\mathsf{det}} \cup V_e^{\mathsf{hand}} \cup \mathbf{var}(\mathsf{adv})$. □

4.1 Multi-level Fault Handling

In the RTU example from Figure 4 the occurrence of memory corruption fault is quite disastrous. Not only that it threats RTU's main function, but also the detection and handling of other faults. This kind of situation occurs frequently in practice. The fault handling strategy of previous sections unfortunately cannot deal with this kind of faults. A commonly used technique is *multi level fault handling*. The idea is to divide faults into several levels, each level consisting of a set of faults which cannot not invalidate each other's detection and handling invariants. Faults of higher level may however destroy the invariants of lower level faults. For each level i we would need separate detection by Main and a separate handler. A handler of higher level faults is capable not only to override Main, but also the handlers of lower level faults.

The theory of the previous section can be generalized to multi level fault handling. Let FH_i denote the fault handler of level i faults and FTS_i denote the composition $\mathsf{Main}[\![\mathsf{FH}_0[\![...[\![\mathsf{FH}_i$. The idea is that relative to FTS_{i+1} the composite FTS_i acts as Main and FH_{i+1} acts as the fault handler in the ordinary one-level multiple faults scenario of Section 4. So, the theory developed so far is also applicable here by applying it at each level.

References

[1] A. Arora. *A foundation for fault-tolerant computing.* PhD thesis, Dept. of Comp. Science, Univ. of Texas at Austin, 1992.

[2] A. Arora and M.G. Gouda. Distributed reset. In *Proceedings of the 10th Conference on Foundation of Software Technology and Theoretical Computer Science*, 1990. Also in *Lecture Notes on Computer Science* vol. 472.

[3] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley Publishing Company, Inc., 1988.

[4] P. Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23:107–125, dec 1994.

[5] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, 1974.

[6] Ted Herman. *Adaptivity through Distributed Convergence.* PhD thesis, University of Texas at Austin, 1991.

[7] L. Lamport. An assertional correctness proof of a distributed algoritm. *Science of Programming*, 2:175–206, 1982.

[8] P.J.A. Lentfert and S.D. Swierstra. Towards the formal design of self-stabilizing distributed algorithms. In P. Enjalbert, A. Finkel, and K.W. Wagner, editors, *STACS 93, Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 440–451. Springer-Verlag, February 1993.

[9] I.S.W.B. Prasetya. Variable access constraints and compositionality of liveness properties. In H.A. Wijshoff, editor, *Proceeding of Computing Science in the Netherlands 94*, pages 12–23. SION, Stichting Matematisch Centrum, 1993.

[10] I.S.W.B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms.* PhD thesis, Dept. of Comp. Science, Utrecht University, 1995.

[11] I.S.W.B. Prasetya and S.D. Swierstra. Component-wise verification of distributed systems. Available at `www.cs.uu.nl/~wishnu`, 1999.

[12] Udink. R. *Program Refinement in UNITY-like Environments.* PhD thesis, Utrecht University, 1995.

[13] N. Shankar. Lazy compositional verfication. Available at `www.csl.sri.com/fm-papers.html`., 1999.

[14] A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989.