Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

An approach for the formal development of fault-tolerant systems using CSP

Fernando Luís Dotti

Relatório Técnico N^0 055

Porto Alegre, Setembro de 2009

Resumo

This document reports on ongoing work[1] towards a framework for specification and reasoning about fault-tolerant systems using CSP. In special, this framework should support system reengineering to become fault-tolerant. It is considered that the kinds of faults that affect, and how they affect, are often not well known until an initial model of the system is built, without considering faults. A reengineering perspective is interesting because it separates concerns during design and because it may be useful for existing systems that have to adapt to other environments.

More specifically, the contributions are: (i) the proposal of a general method for system reengineering to consider fault-tolerance. Such proposal makes explicit which kinds of proofs should be developed while reengineering a system for fault-tolerance; (ii) a method to represent classical types of failures in parts of a system under reengineering. In contrast to the literature, according to the proposed method the structure of parts of the model are kept intact, instead of being modified for representing failure modes. We show that the represented failures respect a classical hierarchy of failure modes. As another advantage, the proposed method allows the explicit, formal and localized definition of patterns of failures that may affect the system. These ideas are exemplified using a simple CSP model. The addition of tolerance mechanisms for failure detection and correction are briefly discussed.

[1]Work initiated during post-doctoral visit to York University - UK. Acknowledgements to Dr. Ana Cavalcanti for discussions on this topic and to CNPq for financial support.

1 Introduction

The development of fault-tolerant systems is not a trivial task. There are several indications [Rom07, Cri89] of the introduction of software errors while developing fault-tolerance mechanisms. Therefore, the use of formal methods in the design of fault-tolerant systems is natural and not a new direction.

Mechanisms for fault-tolerance are discussed since several years [LA90] and the engineering of such mechanisms is very much dependent on system structuring. The formalization of fault-tolerant systems requires the ability to reason about modules, state, interfaces and processes. The use of process calculi is thus natural, and CSP is often employed in this area. Refinement based approaches are also important in this context since they offer an approach to bridge the gap between specification and implementation and offer a methodological way to detalize a system by introducing mechanisms to cope with specific requirements. In this case, we could envisage the introduction of detection and correction mechanisms as refinement steps. In Section 3 we survey some of the most important efforts in this direction.

The investigation of approaches to the formal development and reasoning about fault-tolerant systems is still an active area. In general, this area, and the present contribution, aim at providing abstractions and methods in order to model and reason about: (i) failure modes; (ii) behaviour of the system in presence of faults; (iii) effectiveness of fault tolerance mechanisms.

While we discuss means to achieve such general goals, we contribute in this document to some specific directions. One of them is that we support a reengineering approach to achieve fault-tolerance. This means that we explicitly consider that fault-tolerance for a specific system is often not well understood until a first model of the system, considering an idealized world (without faults), is built. An initial existing model helps to understand and characterize the most important fault assumptions for the specific system. Therefore, a reengineering approach is well suited: we would like to step-wise transform the system to become fault-tolerant.

A second specific way we contribute, is to the representation of faulty behaviors. According to our methodology (see section 4) we identify parts of the system that may be affected by certain classes of faults. We then proceed to model the behaviour of that affected parts in the presence of a selected fault. Our approach here is to provide a "library"of definitions that allow one to wrap the part of the system to represent its behaviour when affected. During this process, the interface (observable behavior) of that specific part of

the system is modified. Fault-tolerance means in this context will be those that detect and react to behaviours of the extended interface. Although not developed in this document, approaches for detection and correction are discussed.

This text is organized as follows: in Section 2 we review main Dependability concepts that may be useful in the rest of the text; in Section 3 we survey the most related work in the literature - here we focus mainly on process based approaches for the formal development of fault-tolerant systems using refinement; in Section 4 we give an overview on how we understand the process of design and reasoning about fault-tolerant systems; in Section 5 we detail the developed notions for fault modelling; in Sections 6 and 7 we discuss how we consider the construction of detection and correction mechanisms, as well as comment on future work.

2 Discussion of main Dependability Concepts related to this document

Here we discuss some of the dependability concepts which are central to the rest of this material. Most of the terms are based on [LRAL04]. Where appropriate, additional references are given.

Fault, Error, Failure. In the dependability vocabulary the terms fault, error and failure are central and we briefly discuss them here. A failure is an 'event that occurs when the delivered service deviates from correct service'; error is 'the part of the system state that may lead to subsequent failure'; and fault is the 'hypothesized cause of an error', the actual defect in the system.

There is a causal chain relating fault, error and failure, in this order. A fault may be active or dormant. An active fault causes an error. An error may be propagated until it affects an external state, which is visible, part of the system interface, and thus a failure takes place.

Boundaries/Interfaces. The use of the above mentioned terms are very much dependent of the context being discussed. In the above discussion the notion of interface or boundary is implicit. While crossing boundaries, the chain of fault, error and failure may be (re)interpreted in a wider context.

The failure, perceived in the interface of a part of a system, may be regarded as a fault to the rest of the system, which leads to an error and again to a failure in a wider context.

For instance, a fault may be the wrong coding of a process in a group of cooperating processes. For the sake of discussion, the affected process is called 'faulty process', even if it is in a stage where the fault has not yet occured. The fault is dormant and can be activated by a specific input to the faulty process. Once activated, the process switches to a state that characterizes an error, and eventually this error affects the process interface, causing a failure. The external behaviour of the faulty process, a failure from the point of view of the process, may thus be considered as a fault to the group of cooperating processes. For instance, the faulty process may crash (if a divide by zero coding was the fault), or it may output spurious values (if a wrong computation of the result was coded), or it may even omitt the output.

Fault tolerance is directly related to 'control' this propagation chain, or in other words, to stop this propagation or to reduce the severity of the failure propagated to wider contexts. Therefore, confinement regions, regions that stop error propagation, are needed. It is highly desirable that such regions match also system boundaries.

Fault prevention, removal and tolerance. The dependability of a system may be enhanced by: fault prevention, fault removal and fault tolerance. Fault prevention means to prevent the introduction of faults (for instance, by formal development one believes to prevent introduction of software faults). Fault removal means to reduce the number of faults in a system (for instance through maintenance, changing parts of the system). Fault tolerance is the ability of the system to avoid failures in the presence of faults, i.e. to avoid the deviation from the service specification even if faults occur.

Fault tolerance levels. In practice fault tolerance may have several degrees. In [KA00] three levels of fault tolerance are defined as a combination of liveness and safety properties that are preserved by a fault-affected system: the system is said to be *masking* fault tolerant when it remains in safe and live states even in the presence of faults - it is the most desirable form of fault-tolerance. Some mechanisms for masking fault-tolerance may offer degraded performance (e.g. slower service) or degraded functionality (e.g. some

functions not available). The system is said to be *fail safe* fault-tolerant if it remains in safe states but does not respect liveness. The system is said to ve *nonmasking* fault-tolerant if safety is not guaranteed but livenes is. The first form may be very expensive and the second and third forms may be still acceptable depending very much on the application. For completenes, a fourth combination would be no safety and no liveness, which obviously corresponds to no fault-tolerance at all.

Failure Modes. Since faults affect parts of the system, we characterize and discuss the failure modes exhibited by the affected parts of the system. As mentioned above, in a wider context such partial failures are the actual faults to be tolerated and thus fault tolerance mechanisms may be necessary.

According to the literature, the following failure modes can be identified. The discussion here follows [G98], which refers to [SS83, HT94, LSP82, PT86]. Although this taxonomy is employed for distributed systems, they are often used also with other systems.

- Fail-stop: a process fails by halting and the fact that the process has failed is signaled to other processes. This failure mode abstraction provides failure detection for the other parts of the system.
- Crash: a process fails by halting.
- Send omission: a process may perform as in the Crash failure mode and additionaly the process may transmit only a subset of the messages that it actually should (some output messages are lost).
- Receive omission: a process may perform as in the Crash failure mode and additionaly the process may receive only a subset of the messages that it actually should (some input messages are lost).
- General omission: a process may experience both send and receive omission failures.
- Byzantine: a process may experience any failure mode above and additionally allows any behaviour allowed by the system, inuding arbitrary or malicious.

Failure modes are also related in hierarchies representing degrees of severeness, or, more technically, possible computations of a system in presence

of faults. A classical hierarchy involving the above mentioned failure modes is presented in [Cri91]. In [Pow92] the author proposes two dimensions to classify failure modes and then relates them by giving an order from less to more severe failures.

3 Related Work

In this section we survey main related work on the use of formal methods to the specification of fault-tolerant systems. Focus is limited by the use of refinement based formal methods.

In [Pel91] we find one of the earlier papers discussing how to design and verify fault-tolerant systems. In this paper, Peleska uses CSP and illustrates the ideas through a simple example where a server has to communicate with its environment by outputing a value for each input received. In the event of a server's crash, a spare server resumes the service.

The method (although the author does not uses this expression) followed is briefly described below.

- A top level specification of the system is provided. This is an 'implicit specification', or as [Sch99] refers, a property oriented specification. Peleska uses traces and failures to express safety and liveness properties.

  Example: traces are used to state that the traces of a system contains input event(s) and that the prefix before the input is finite. This specifies that the system will accept inputs within finite steps (a liveness property). Failures are used to state that the number of outputs in a trace is less than or equal to the number of outputs (safety). Failures are used (together in the same specification) to state that if the system has equal number of inputs and outputs, then inputs are not refused (liveness); and to state that if the number of inputs is greater than outputs, then outputs are not refused. The author then proves that this specification implies that outputs happen in bounded steps.

- The alphabet of the system is augmented to contain fault events.

  Example: crash events are introduced, one for the main server and another for the spare server (backup).

- A property oriented specification of the system in presence of faults is provided.

Example: the number of crashes is restricted to one, either the main server or the spare crashes, but not both. Since the example supposes that in presence of one crash the system will have the same interface behavior, then the same specification, concerning input and outputs, has to be satisfied. This specification accepts all behavior that the former one (without fault) does, and includes all computations where a single crash happens.

- Then, following an invent and verify approach, an explicit specification of the target fault-tolerant system is provided.

  Example: this explicit specification states that in the event of one crash the spare server is activated. It also introduces two FIFO buffers for input and output, which are assumed to be provided by the environment and thus are not influenced by an eventual crash. These buffers extended with some control help to eliminate possibility of loosing or duplicating messages in the event of a crash of the main server and switching to the spare server. The case of spare server crash is also provided, but is simply stops the spare server and the service is continued by the main server.

- The correctness of the system is proved in the absence of faults. The approach follows the compositional characteristic of the explicit specification, which is build by several processes in parallel. For each of the parallel processes, the specification of the system in absence of faults is proved w.r.t. that process.

- The correctness of the system is proved in presence of faults. The approach followed is to show that when no faults happen, the correct behavior is the same as the already proved one. When faults happen, the complementar behavior has then to be proved correct.

  Example: it is proved that when switching to the spare server (after a crash), the correctess is maintained.

The implicit specification of the system in presence of fault augments the behavior of the system without faults to the cases where faults happen. The obtention of the faulty system is dependent on the designer. This step is important because it formalizes the fault assumption and it is desired that it is constructed in a meaningful way because it bases the whole process of designing the fault tolerance mechanisms.

Although the paper has a methodological argument, the explanation followed very much the case study. Since the case simplified some of the aspects of fault-tolerant systems, they where not discussed. One of the aspects is that the fault occurrence (crash) directly triggers the correction action, switching to the new service. Error detection has been abstracted - it is actually reasonable to assume that some detection mechanism is in place, providing detection. The correction actions are also atomic, since the switch to the spare process is in one step. The case did not raise discussion about levels of fault-tolerance (e.g. masking, non-masking, fail-safe) or failure models since it offers masking tolerance for the fault assumption and the switch back to normal operation is immediate after the crash. The correctness of more complex correction actions were not discussed.

In [Nor93], Jens Nordahl presents an approach to the verification of fault tolerance properties of designs, using CSP [Hoa85], following the failures and divergences model. A system is regarded as a collection of components and a design determining how components interact. Each component may internally have the structure of a system. The design relies on a series of assumptions on the behavior of the components.

The steps followed are summarized below:

- A property oriented specification of the correct system is given.

  Example: in a buffer, the output channel trace has to be a prefix or equal to the input trace.

- According to the paper, a 'failure mode' characterizes a system that may behave incorrectly. In the extreme case the failure mode is equal to correct behavior - actually, no failures are considered. In other situations, the failure mode specifies how the system may fail.

  A failure mode is given in terms of a property oriented specification. Several failure modes may be specified for a system.

  Example: a buffer may output corrupted values.

- Since a recursive architecture is adopted (system composed by components and these may be systems), the failure mode of one component may be regarded by a fault affecting the system (fault-error-failure propagating through boundaries). Also, from the failure modes of the components and knowing the design of the system (how components interact) it is possible to derive the failure mode of the system.

Example: a system composed by two unbounded buffers in a pipeline architecture, i.e. the external input is to the first buffer, the output of the first buffer is input of the second and the output of the second is the external output. Failure modes are in three levels: correct behavior; no spurious output (but possible loss); and byzantine (also spurious). In this case, the failure mode of the system is given by the weakest of its components (most incorrect).

- The design of a system is said to be fault-tolerant if the system implements its specification assuming that the components implement a combination of failure modes.

  The local fault assumptions on the components normally must be complemented by global fault assumptions. For instance: a local fault assumption states that a process may crash. A golbal fault assumption could state that at most N processes may crash. In this paper, global fault assumptions could be expressed by choosing the several components failure modes accordingly to respect the global fault assumption.

- Case study. A service is provided by a main process. If it fails, a spare process will resume operation. A coordinator routes requests to the main process. If the process reports a failure, the coordinator routes the requests to the spare process. If this in turn reports also a failure, the service stops and the user (external) is signaled through an event.

- A property oriented specification for the design is given. It is divided in three cases, traces where no failures are signaled, traces in which one failure is signaled, and traces with two failures signaled. For the three cases, safety and liveness properties are provided. Safety is given in terms of acceptable traces and liveness is stated by restricting refusals for certain traces (obliging the process to be able to proceed with expected synchronizations).

- The author then proves that given the components (which may fail - except the coordinator), the design tolerates component failures (restricted to the global fault assumptions) offering a correct service.

The author does not address how to obtain a fault affected system. No explicit specification of any component part is given. The paper is focused on showing that a design is fault-tolerant considering some components.

Both components and designs are provided in terms of property oriented specifications.

In his thesis [Liu89], Liu aims to provide a formal framework to the description of fault-tolerant programs. The scope of faults considered are those caused by underlying software or hardware, called physical faults.

The whole idea could be summarized as follows:

- Suppose a specification Sp and a program P that satisfies the specification in a faul-free environment;

- The effect of physical faults on P is modelled as *fault actions* leading to erroneous states (violating Sp). A fault program F models the fault environment of P. The environment interference by F on P is regarded as a fault transformation which transforms program P into program $F(P)$, where $F(P) = P[]F$ (the union of sets of actions of P and F). The fault transformation embeeds the fault assumptions.

  One of the aspects considered importat is the need to model faults in order to reason about fault tolerant systems. The author calls 'failure semantics' of a program its behavior when running on a fault-prone hardware. This failure semantics is obtained from the semantics of the program and of the actions representing the faults of the system.

- It is expected that the behavior of $F(P)$ will not satisfy Sp. Thus, *fault-tolerant transformation(s)* are introduced. A fault-tolerant transformation T introduces fault-tolerant actions and thus $F(T(P))$ may satisfy Sp.

- The refinement of program P towards an implementation will add information to it. According to the author, since the refinement may change representation of resorces, on which the faults act, a refinement of the fault properties has to take place along with the refinement of the program. The same argumentation as above is valid also for the refinement of fault-tolerance mechanisms.

The author uses UNITY and refinement calculus [Bac88] to develop the ideas.

4 Method Overview

In this section we overview the envisaged method for the construction of a fault-tolerant system.

Aim: start from model M conforming to specification M_A, add fault-tolerance w.r.t. fault F and achieve M_{Ft} that is guaranteed to be conform to M_A in presence of fault F.

- It will not, in general, be the case that $M \sqsubseteq M_{Ft}$. It is part of the objective to find out how to relate M, F, and M_{Ft} via refinement to prove that functionality is preserved.

Obtaining initial description of the system: comprised by M_A, M, and F, as follows:

- elaborate an abstract model of the system M_A that captures its desired observable behavior;
- elaborate fault assumption F, a description of the types of faults that may affect the system and to which fault-tolerance mechanisms shall be constructed;
- elaborate M, a model of the system. M should be detailed enough such that, together with a fault assumption F, it is known which parts of M will be affected by F. This and the previous step may complement each other as the system is detailed.
- show that M satisfies M_A, $M_A \sqsubseteq M$, under assumption that no fault occurs.

Fault Modelling in M to obtain M_F:

- in M identify part of system that can be affected by F, called AP;
- generate model of affected part in presence of fault F, obtaining AP_F.
  - affected part is one which fails, according to some failure modes as above discussed, depending on the fault assumption F. The

observable behavior of AP_F is the the one defined by AP, augmented by possible behavior defined by the appropriate failure mode.

- AP_F will be generated by wrapping AP to represent the augmented behavior. In contrast to other techniques, the structure of AP is maintained, but input and output channels to/from AP will be connected to the wrapper and disconnected from M. The wrapper will assume the place of AP in M, connecting with the respective channels between AP and M.
- the combination of AP and the wrapper forms AP_F. The specification of wrappers is further discussed in this document, in Section 5, where it is discussed how to represent the several failure modes mentioned before. Moreover, a refinement relation among wrappers is shown, which is consistent with classical failure mode hierarchies.
- it has to be shown that:

  (A) no interference - $M(AP) \sqsubseteq_F M(AP_F||B)$, where B should block failure behavior F of $M(AP_F)$. This means that the extended model still exhibits the idealized model's behavior, when no faults happen.

  (B) effectiveness of fault modelling: assuming that F occurs, AP_F behaves faulty - it deviates from AP's behavior.

- at this point we have represented the effect of F in M obtaining M_F. M_F is obtained from M by using AP_F in the place of AP.

  Next is to make M_F F-tolerant

Introducing Fault-Tolerance in M_F to obtain M_{Ft}:

- if M_F satisfies M under the assumption that F may take place, M is F-tolerant, i.e. $M = M_{Ft}$, and no further step is needed, otherwise follow steps below:
- identify confinement region for AP_F, CR. CR is a part of the system that: (i) contains AP_F and (ii) is supposed to be enriched with mechanisms to tolerate F. Thus, the interface of CR should not change for the rest of the system.

- according to kind of fault and ft-mechanisms, modify CR to become F tolerant, obtaining CR_{Ft}. Here the discussion on patterns for fault tolerance could be developed.

- prove:

  - (A) no inteference: assuming that no fault occur, CR_{Ft} behaves as CR - i.e. under no fault, model M_{Ft} obtained by substituting CR by CR_Ft satisfies M;

  - (B) fault tolerance: assuming that fault does occur, the same as (A).

Defining Failure Mode in case of impossibility of fault-tolerance measures:

- in case the fault cannot be tolerated completely, the region cannot be called 'confinement', and we call 'handling region' HR.

- HR cannot provide transparency to failures but may be designed to minimize side-effects (for instance behaving as fail-stop or offering degraded service). Since HR does not confine the failure, the interface of HR will have to be modified, impacting the rest of the system. The new behavior has to be properly characterized as the failure mode of the system.

- hierarchies (nesting) of HRs is possibe, which would from the point of view of fault-tolerance model error confinement and propagation regions and from the point of view of modelling would be the units of reengineering and reasoning. A HR could reduce the severity of the fault in AP_F. For instance, AP_F could behave according to byzantine and HR could behave according to fail-stop when AP_F fails.

- prove:

  - (A) no interference, just like above, under no fault;

  - (B) effectiveness of failure mode: under fault F, HR will fail in a well defined way - characterize failure modes.

These steps give a basic structure to the reengineering process to achieve fault-tolerance. Following these steps, fault assumptions, tolerance mechanisms and failure modes become explicitly and formally defined in the system architecture.

5 Wrappers for Fault Modelling

As mentioned above, we start from a model M, that supposes that no fault takes place. F and AP are also defined. The aim of the wrapping process is to obtain AP_F, which is AP in presence of fault F. AP_F is needed to be able to reason about the fault-tolerant system to be built.

Using wrappers, AP's structure is totally preserved. To allow control and reasoning about faults, we need to be able to enable and disable fault behavior. Thus the wrapper shall have channels to allow activation and deactivation of faulty behavior, leaving this control outside the wrapper. This allows for a better modelling of local and global fault assumptions. *Local fault assumptions* define the possibility of specific parts of a system failing according to specific types of fault. *Global fault assumptions* state conditions for the activation of a fault. Such conditions may be relative to the same affected part (for instance, how many times a process/module may suffer from crash; or if a second crash may take place while still recovering from the previous) or relative to other parts of the system (for instance, at most one process will be crashed). Global fault assumptions restrict fault occurrences allowed by local fault assumptions.

The focus here is to represent the behavior of the following failure modes: crash, omission (send, receive and general) and byzantine. The use of words fault and failure here depend on the context. A failure of AP_F is considered a fault in the wider context M. Therefore in this discussion, modeling failures of AP or faults in M, caused by AP, have the same meaning. The literature [G98] [Cri91] presents a hierarchy of failure modes (crash, ommissions, byzantine), however a more formal characterization of their possible computations is not discussed. Through the wrapping process here proposed we turn this characterization more precise and formally show, using refinement, that this hierarchy exists.

We suppose that AP communicates with the rest of the model through input and output channels. The manifestation of failure modes of AP, to the rest of the model, takes place as anomalies in the communication behavior

of AP. To each input and output channel of AP is attached an instance of a 'canal' as discussed bellow. This instance allows communication anomalies to be selected (crash, omission and byzantine) and affect one specific communication channel of AP. Since AP may have several communication channels, the wrapper is built by instantiating and configuring one 'canal' to each existing communication channel of AP.

Bellow we present the interface to create one such channel. The 'Unidirectional Bounded Blocking Canal' implements a FIFO with limited size. When it is full, it blocks inputs. When empty, blocks outputs.

```
-- Unidirectional Bounded Blocking Canal
-- Fg - synchronizes with a specification of global failures

UBBCanalFg(in, out, k, crashIn, crashOut, omission, byzantine) =  ...
```

- When wrapping an input channel of AP, *in* will synchronize with the environment and *out* will synchronize with AP's input channel. When wrapping an output channel of AP, we invert direction: *in* will synchronize with AP's output and *out* with the environment. k is the size of the queue.

- $crashIn$, $crashOut$, *omission* and *byzantine* are used to synchronize with a global description of possible patterns of failures and thus generate the communication anomalies.

- $crashIn$ is used to model the behaviour of an input channel when AP is crashed. This means that inputs in *in* are lost and *out* is not offered, so AP does not have any input.

- $crashOut$ is used to model the behaviour of an output channel when AP has crashed. It does not accept *in* meaning that the synchronization with AP is not offered - AP blocks with respect to this channel. It offers *out* because these represent signals or messages already offered previously by AP.

- *omission* allows the channel to lose an input. If it happens in an input channel, it models a receive omission, if in an output channel, it models a send omission. On both, it models general omission. A model exhibiting

- *byzantine* allows the modeling of a change of values in the communication.

```
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- Unidirectional Bounded Blocking Canal -
-- Fg - synchronizes with a specification of global failures


UBBCanalFg(in, out, k, crashIn, crashOut, omission, byzantine) =
               UBBCFg(in, out, <>, k, crashIn, crashOut,  omission, byzantine)

-- empty
UBBCFg(in, out, <>, k, crashIn, crashOut, omission, byzantine) =
               if k>0       -- test of initialization parameter
               then UBBCInputFg
                    (in, out, <>, k, crashIn, crashOut, omission, byzantine)
               else STOP

-- not empty
UBBCFg(in, out, <y>^s, k, crashIn, crashOut,  omission, byzantine) =
               if k>0
               then  ( UBBCInputFg
                       (in, out, <y>^s, k, crashIn, crashOut,  omission, byzantine)
                       []
                       UBBCOutputFg
                       (in, out, <y>^s, k, crashIn, crashOut,  omission, byzantine) )
               else  UBBCOutputFg
                       (in, out, <y>^s, k, crashIn, crashOut, omission, byzantine)

UBBCOutputFg(in, out, <y>^s, k, crashIn, crashOut, omission, byzantine) =
              ( out!y -> UBBCFg
                         (in, out, s, k+1, crashIn, crashOut, omission, byzantine) )
                []
                crashIn -> UBBCCrashedInFg
                           (in, out, <y>^s, k, crashIn, crashOut, omission, byzantine)
                             -- this canal goes to crashed process:
                             --            accepts in, does not offer out
                             --            messages from others may be send
                []
                crashOut -> UBBCCrashedOutFg
                             (in, out, <y>^s, k, crashIn, crashOut, omission, byzantine)
                      -- this canal comes from crashed process:
                      --           does not accept in (blocks crashed process),
                      --           accepts out: messages in transit may be delivered


UBBCInputFg(in, out, b, k, crashIn, crashOut, omission, byzantine) =
             (in?x -> UBBCFg(in, out, b^<x>, k-1, crashIn, crashOut, omission, byzantine) )
              []
             (crashIn -> UBBCCrashedInFg
                         (in, out, b, k, crashIn, crashOut, omission, byzantine) )
              []
             (crashOut -> UBBCCrashedOutFg
                          (in, out, b, k, crashIn, crashOut, omission, byzantine) )
              []
```

```
            (omission -> in?x -> UBBCFg
                        (in, out, b, k, crashIn, crashOut, omission, byzantine) )
             []
            (byzantine -> in?x ->(|~| y:reqId @
                  (UBBCFg(in, out, b^<y>, k-1, crashIn, crashOut, omission, byzantine))) )



UBBCCrashedInFg(in, out, b, k, crashIn, crashOut, omission, byzantine) =
           (in?x -> UBBCCrashedInFg(in, out, b, k, crashIn, crashOut, omission, byzantine))
                              -- this canal goes to crashed process:
                              --            accepts in, does not offer out.
                              --            messages from others may be send
                              --            inputs may be simply loss

UBBCCrashedOutFg(in, out, <>, k, crashIn, crashOut, omission, byzantine) = STOP
                      -- recovery ?

UBBCCrashedOutFg(in, out, <y>^s, k, crashIn, crashOut, omission, byzantine) =
                               -- this canal comes from crashed process:
                               --           does not accept in (blocks crashed process),
                               --           accepts out: messages in transit may be delivered
        (out!y -> UBBCCrashedOutFg(in, out, s, k+1, crashIn, crashOut, omission, byzantine) )
```

As mentioned before, the wrapper is composed by one instance of the above process to each of the communication channels of AP, or AP's observable behavior. To exemplify the idea we now we take the definition of an AP, proceed to wrap it, and investigate its behavior. We consider a simple service interface that, when receiving a request, answers with a response (req and $resp$). Additionally, AP may generate reports (rep) to the rest of the model.

```
-- -- -- channels
-- service channels
-- resp used to respond to req
-- rep is a report that the process P may generate without req

reqId = {1..2}
channel req, resp: reqId
channel rep: reqId

P = Pserv [|{|req|}|] Prep(1)
Pserv = req?n -> resp!n -> Pserv
Prep(repInterval) = req?n -> if repInterval>0
                             then Prep(repInterval-1)
                             else rep.1 -> Prep(1)
```

To wrap the above mentioned process we create three 'canal' processes as mentioned before, linking each one to a channel and the rest of the system

communicates with the wrapper using new channels. We use suffix 'Out' for channels that output some value to the wrapper and suffix In on the other direction. In contrast to previous versions not discussed here, bellow we use AP_{fg} to denote the affected part using the wrapper with global failures. Global failures means that the designer is able to specify a pattern of failures to take place with AP_{fg}.

```
channel reqOut, respIn: reqId
channel repIn: reqId

-- events to synchronize - enable failure occurrence
canalId = {1..3}
direction = {1..2}
channel cr: direction
channel om, by: canalId

APfg =
   ( ( ( P [|{| req, resp, rep|} |]
       (
        ((UBBCanalFg(reqOut, req, 1,  cr.1, cr.2, om.1, by.1)  [|{|cr|}|]
          UBBCanalFg(resp, respIn, 1, cr.2, cr.1, om.2, by.2)) [|{|cr|}|]
          UBBCanalFg(rep, repIn, 1,   cr.2, cr.1, om.3, by.3))
        )
      ) \{| req, resp, rep|}
    ) [| {| cr, om, by |} |] GFA )\{| cr, om, by |}

GFA = SKIP
```

As the reader can see, req, $resp$ and rep are now hidden and $reqOut$, $respIn$ and $repIn$ offered to the rest of the system. The wrapper also synchronizes using cr, om and by. The last ones are used to generate crash, omission and byzantine behavior respectively. Omission and byzantine may take place in each of the input or output channels and therefore an additional parameter, the channel identification, is used.

When one omission takes place, one original synchronization of AP with the rest of the system is lost. When one byzantine behavior takes place, the value of some communication is randomly changed by the wrapper, according to possible pre-defined values.

While omission and byzantine anomalies affect only one instance of communication (or synchronization), crash is slightly different. Upon a crash the rest of AP_{fg}'s behavior is affected. All channels are affected. However, to build the wrapper, we observe different behaviors of channels depending on wether they input or output data to/from a crashed process. In the above

modeling, $cr.1$ is the crash occurrence that makes the wrapper behave as process P has crashed. $cr.2$, on the other side, would model the channel behavior when the rest of the system had crashed and P would still be normal. $cr.2$ is here shown for completeness, since we are investigating the effects of failures in P.

The process GFA models global assumptions about failure occurrences. It is a description of the possible failures in the whole system. The designer has a formal, explicit and localized (in one place) characterization of the failure patterns being investigated in the system. This process runs in parallel with the rest of the system and synchronizes with the wrapper(s) to generate the specified failures. Below we exemplify the use of this process to generate several kinds of failures.

```
-- no failures
GFA = SKIP

-- one instance of crash
GFAc = cr.1 -> SKIP

-- any pattern of receive omissions
GFAro = RUN({om.1})

-- any pattern of send omissions
GFAso = RUN({om.2,om.3})

-- any pattern of omissions - send and receive
-- as will be shown, this will include the crash behavior
GFAgo = RUN({|om|})

-- byzantine behavior
-- synchronizations in by.{1..3} corrupt communication in channels 1 to 3
-- the byzantine behavior is characterized to include all others and,
-- additionally, corrupting communication
-- the definition below includes omissions, which then includes crashes,
-- and causes corruption in output communications from $AP_fg$
GFAb =  RUN({|om,by.2,by.3|})
```

Considering the above specified failure modes, we compare the behaviors of AP when affected by them. The hierarchy below means that behaviors on upper levels include behaviors on lower levels. Each edge means that the lower level is a refinement of the upper level. Where not stated otherwise in the pictures, the edges of this picture mean both traces and failures refinement.

```
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- COMPARING THE FAILURE SEMANTICS -
--
-- HIERARCY OF FAILURES FROM MORE(TOP) TO LESS(DOWN) SEVERE
--
--          Byzantine
--             |
--      GeneralOmission
--        /          \
-- SendOmission ReceiveOmission
--           \   /->T but not FD (due to reports)
--           Crash
--             |
--        No Failures
```

The above shows that the provided definitions for the wrapper can be useful to reason about failure modes as discussed in the literature. The hierarchy of failures is represented in a formal setting and refinement can be used to reason about it. If tolerance mechanisms are built considering one of these failure modes, any less severe failures can be tolerated too.

For the complete model and other analyses see Appendix 1.

6 Discussion on introduction of fault-tolerance

With the above discussed wrapper we have introduced the possibility of representing failure modes in specific parts of the system and of specifying failure patterns to investigate. The next steps are to devise detection and correction mechanisms.

Failure Detection. An important aspect is how failure detection should take place. In general we can have an abstract and a concrete view. In the abstract view we consider that failure detection is provided by some mechanism. The failure is simply signaled, triggering recovery mechanisms. This abstraction is common in this kind of system since detection can be a complex task, specially in distributed systems. Traditional literature provide abstractions [CT96] that classify detectors in accuracy and completeness properties.

The concrete view is one where the failure detector mechanism itself is a part of the model. Following a refinement approach it is desired to start with

the first, an abstract notion, and implement it through refinement, reaching the second.

With respect to failure detection, an initial approach to follow is to consider that any failure is perfectly detected. By perfect detection it is meant that all failures are detected, with non false positives, and the detection is immediate. Our initial proposition for a perfect failure detection module is simply to leave the failure occurrences (specified by failure process definition (GFA)) visible to a detection module. This can be achieved simply using multiparty synchronization.

By supposing a perfect failure detector we can investigate the correction possibilities in an idealized setting, isolating imperfection of the detection process. If correction is not possible in such setting, then it is not possible at all. However, such a functionality is very rare in a real setting. Thus one track of future work is to model abstractions of failure detectors. Analogously to the wrapping strategy for failure representation, we could start from an idealized (perfect) failure detector and stepwise enlarge its behavior to represent possible, real imperfections in terms of accuracy and completeness. With such an hierarchy we could then choose the most plausible one to a given reality, as well as compare system properties with different classes of detectors.

Correction. Although there are certain classical forward and backward error correction mechanisms [LA90] error correction strategies can also be very dependent of the application and may need to be tailored to them. Whatever the case, but most importantly for the second, the proposed framework helps to specify and reason about such mechanisms.

Regarding the first case, since we are using a refinement approach, one important contribution would be the proposition of standard refinement steps that accomplish the introduction of classical correction mechanisms. To achieve that, we should first define the elements of the abstract model that must be present to allow the application of such a refinement step. Then we should show that to any model obtained, starting from the abstract one and applying the refinement step, the desired properties are respected. This would show that the refinement step is correct in general. This is a rough definition of a possible avenue for future work.

7 Final Considerations

This document reports on ongoing work towards a framework for specification and reasoning about fault-tolerant systems using CSP. A general method for reengineering systems to become fault-tolerant has been presented and discussed. This method will suffer modifications as each of its steps is developed. This document presents in more detail the developed notions for failure modeling. This is aimed to represent the behavior of a selected kind of failure, from a classical hierarchy in the literature, in affected parts of the system under reengineering. The failure modeling approach is shown to be compatible with existing comportamental hierarchies of failures. Another interesting aspect is the ability the designer will have to specify failure patterns to reason about the whole system.

A basic abstraction for failure detection is provided simply by considering the failure occurrences, specified by the designer, as immediately detected by an abstract failure detection module. This allows the future modeling of failure detection anomalies to represent other classes of detectors, closer to reality. Despite work in each of the above mentioned topics, future directions also comprise the addition of known tolerance mechanisms using standard refinement steps, build and proved in advance.

Referências

[Bac88] R. J. R. Back, *A calculus of refinements for program derivations*, Acta Inf. **25** (1988), no. 6, 593–624.

[Cri89] Flaviu Cristian, *Exception handling*, Blackwell Scientific Publications, 1989.

[Cri91] F. Cristian, *Understanding fault-tolerant distributed systems*, Communications of the ACM **34** (1991), no. 2, 56–78.

[CT96] T. D. Chandra and S. Toueg, *Unreliable failure detectors for reliable distributed systems*, Journal of the ACM **43** (1996), no. 2, 225–267.

[G98] F. C. Gärtner, *Specification for fault tolerance: a comedy of failures*, Tech. Report TUD-BS-1998-03, Department of Computer Science - Darmstadt University of Technology, Germany, 1998.

[Hoa85] C. A. R. Hoare, *Communicating sequential processes*, Prentice Hall, 1985.

[HT94] Vassos Hadzilacos and Sam Toueg, *A modular approach to fault-tolerant broadcasts and related problems*, Tech. report, Ithaca, NY, USA, 1994.

[KA00] Sandeep S. Kulkarni and Anish Arora, *Automating the addition of fault-tolerance*, Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium (FTRTFT 2000) Proceedings, number 1926 in Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 82–93.

[LA90] P. A. Lee and T. Anderson, *Fault tolerance: Principles and practice*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[Liu89] Zhiming Liu, *Fault-tolerant programming by transformations*, Ph.D. thesis, University of Warkick, 1989, p. 190.

[LRAL04] Jean-Claude Laprie, Brian Randell, Algirdas Avizienis, and Carl Landwehr, *Basic concepts and taxonomy of dependable and secure computing*, IEEE Trans. Dependable Secur. Comput. **1** (2004), no. 1, 11–33.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease, *The byzantine generals problem*, ACM Trans. Program. Lang. Syst. **4** (1982), no. 3, 382–401.

[Nor93] Jens Nordahl, *Design for dependability*, DCCA-3: Proceedings of the Third Working Conference on Dependable Computing for Crictical Applications (C.E. Landwehr, ed.), Springer Verlag, 1993, pp. 65–90.

[Pel91] Jan Peleska, *Design and verification of fault tolerant systems with csp*, Distributed Computing **5** (1991), no. 2, 95–106.

[Pow92] David Powell, *Failure mode assumptions and assumption coverage*, FTCS, 1992, pp. 386–395.

[PT86] Kenneth J. Perry and Sam Toueg, *Distributed agreement in the presence of processor and communication faults*, IEEE Trans. Software Eng. **12** (1986), no. 3, 477–482.

[Rom07] Alexander Romanovsky, *A looming fault tolerance software crisis?*, SIGSOFT Softw. Eng. Notes **32** (2007), no. 2, 1–4.

[Sch99] Steve Schneider, *Concurrent and real time systems: The csp approach*, John Wiley & Sons, Inc., New York, NY, USA, 1999.

[SS83] Richard D. Schlichting and Fred B. Schneider, *Fail-stop processors: An approach to designing fault-tolerant computing systems*, ACM Trans. Comput. Syst. **1** (1983), no. 3, 222–238.

Appendix 1 - CSP Model

```
---
-- Description
---
-- A service is used in sessions which are initiated and finalized.
-- After initiating a session and before finalizing, the service may be
-- invocated using srvReq, generating a response srvResp.
-- At the interface, servReq.{1..2} are available concurrently,
-- generating servResp.{1..2} respectivelly.
-- A latter request could be responded before the previous one.
--
--
-- -- -- -- -- C O M M U N I C A T I O N   C A N A L -- -- -- -- -- -- --
--
-- Unidirectional Bounded Blocking Canal - NORMAL

reqId = {1..2}

UBBCanalN(in, out, k) = UBBCN(in, out, <>, k)
UBBCN(in, out, <>, k) = if k>0
                        then UBBCInputN(in, out, <>, k)
                        else STOP
UBBCN(in, out, <y>^s, k) = if k>0
                           then ( UBBCInputN(in, out, <y>^s, k)
                                []
                                  out!y -> UBBCN(in, out, s, k+1)  )
                           else out!y -> UBBCN(in, out, s, k+1)

UBBCInputN(in, out, <>, k) =
                        in?x -> UBBCN(in, out, <x>, k-1)

UBBCInputN(in, out, <y>^s, k) =
                        in?x -> UBBCN(in, out, <y>^s^<x>, k-1)


-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- Unidirectional Bounded Blocking Canal - FAILURE MODELLING - LOCAL ASSUMPTIONS
-- only local assumptions
-- 0 = normal
```

```
-- 1 = may crash
-- 2 = may loose
-- 3 = may change values

channel crashFl

UBBCanalFl(in, out, k, f) = UBBCFl(in, out, <>, k, f)

UBBCFl(in, out, <>, k, f) =
                       if k>0
                       then UBBCInputFl(in, out, <>, k, f)
                       else STOP

UBBCFl(in, out, <y>^s, k, f) =
                          if k>0
                          then ( UBBCInputFl(in, out, <y>^s, k, f)
                                 []
                                 out!y -> UBBCFl(in, out, s, k+1, f)  )
                          else out!y -> UBBCFl(in, out, s, k+1, f)

UBBCInputFl(in, out, b, k, f) =
                      if f==0 then ( in?x -> UBBCFl(in, out, b^<x>, k-1, f)  )
                              else
                      if f==1 then ( -- may crash: will loose all future inputs
                                     crashFl -> UBBCInputLossFl(in, out, b, k, f)
                                     []
                                     in?x -> UBBCFl(in, out, b^<x>, k-1, f)
                                   )
                             else
                      if f==2 then ( in?x ->
                                     ( -- may loose message: leave buffer in same state
                                       UBBCFl(in, out, b, k, f)
                                       |~|
                                       UBBCFl(in, out, b^<x>, k-1, f)
                                     ))
                              else
                      if f==3 then ( in?x ->
                                     ( -- may loose message: leave buffer in same state
                                       -- it is shown that this includes crash
                                       UBBCFl(in, out, b, k, f)
                                       -- may change value for any possible
                                       |~|
                                       -- however, the space of y depend on type of x
                                       -- x may be empty type in case of channel rep
                                       -- x may be reqId in case of rep/resp
                                       (|~| y:reqId @ (UBBCFl(in, out, b^<y>, k-1, f)))
                                       -- may execute correctly
                                       |~|
                                       UBBCFl(in, out, b^<x>, k-1, f)
                                     ))
                                else STOP

UBBCInputLossFl(in, out, b, k, f) = (in?x -> UBBCInputLossFl(in, out, b, k, f))



-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
-- Unidirectional Bounded Blocking Canal - FAILURE MODELLING - GLOBAL ASSUMPTIONS
-- crash, omission and byzantine are channels that synchronize with
-- Global Fault Assumptions (GFA) a process with the allowed failure pattern
-- crashIn and crashOut are respective to the process crashed.  If the crashed
-- process consumes from this canal, then crashIn represents no possibility of
-- synchronizing in the input.

UBBCanalFg(in, out, k, crashIn, crashOut, omission, byzantine) =
               UBBCFg(in, out, <>, k, crashIn, crashOut,  omission, byzantine)

-- empty
UBBCFg(in, out, <>, k, crashIn, crashOut, omission, byzantine) =
               if k>0       -- test of initialization parameter
               then UBBCInputFg
                    (in, out, <>, k, crashIn, crashOut, omission, byzantine)
               else STOP

-- not empty
UBBCFg(in, out, <y>^s, k, crashIn, crashOut,  omission, byzantine) =
               if k>0
               then  ( UBBCInputFg
                       (in, out, <y>^s, k, crashIn, crashOut,  omission, byzantine)
                       []
                       UBBCOutputFg
                       (in, out, <y>^s, k, crashIn, crashOut,  omission, byzantine) )
               else  UBBCOutputFg
                       (in, out, <y>^s, k, crashIn, crashOut, omission, byzantine)

UBBCOutputFg(in, out, <y>^s, k, crashIn, crashOut, omission, byzantine) =
              ( out!y -> UBBCFg
                         (in, out, s, k+1, crashIn, crashOut, omission, byzantine) )
                []
                crashIn -> UBBCCrashedInFg
                           (in, out, <y>^s, k, crashIn, crashOut, omission, byzantine)
                             -- this canal goes to crashed process:
                             --            accepts in, does not offer out
                             --            messages from others may be send
                []
                crashOut -> UBBCCrashedOutFg
                            (in, out, <y>^s, k, crashIn, crashOut, omission, byzantine)
                      -- this canal comes from crashed process:
                      --           does not accept in (blocks crashed process),
                      --           accepts out: messages in transit may be delivered


UBBCInputFg(in, out, b, k, crashIn, crashOut, omission, byzantine) =
             (in?x -> UBBCFg(in, out, b^<x>, k-1, crashIn, crashOut, omission, byzantine) )
              []
             (crashIn -> UBBCCrashedInFg
                         (in, out, b, k, crashIn, crashOut, omission, byzantine) )
              []
             (crashOut -> UBBCCrashedOutFg
                          (in, out, b, k, crashIn, crashOut, omission, byzantine) )
              []
             (omission -> in?x -> UBBCFg
                          (in, out, b, k, crashIn, crashOut, omission, byzantine) )
              []
```

```
            (byzantine -> in?x ->(|~| y:reqId @
                  (UBBCFg(in, out, b^<y>, k-1, crashIn, crashOut, omission, byzantine))) )


UBBCCrashedInFg(in, out, b, k, crashIn, crashOut, omission, byzantine) =
           (in?x -> UBBCCrashedInFg(in, out, b, k, crashIn, crashOut, omission, byzantine))
                              -- this canal goes to crashed process:
                              --            accepts in, does not offer out.
                              --            messages from others may be send
                              --            inputs may be simply loss

UBBCCrashedOutFg(in, out, <>, k, crashIn, crashOut, omission, byzantine) = STOP
                       -- recovery ?

UBBCCrashedOutFg(in, out, <y>^s, k, crashIn, crashOut, omission, byzantine) =
                               -- this canal comes from crashed process:
                               --           does not accept in (blocks crashed process),
                               --           accepts out: messages in transit may be delivered
        (out!y -> UBBCCrashedOutFg(in, out, s, k+1, crashIn, crashOut, omission, byzantine) )

-- UBBCInputLossFg(in, out, b, k, crash, omission, byzantine) =
--                       (in?x -> UBBCInputLossFg(in, out, b, k, crash, omission, byzantine))



-- Crash behavior:
--    Crash affects the process.  The process does not consume or produce messages.
--    Messages may be in transit and are in the canals.
--    Analyzing interface, a crash may happen before any communication, input or output.
--    Input canal: input messages are not consumed.
--                 sychrnonization canal-process not offered
--    Output canal: messages are not sent - the canal does not offer.
--                 synchronization process-canal not offered


-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- --     MODEL   (NO FAULTS) -- -- -- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- initial model extented to concurrent execution of
-- series of requests responses - each serie has identifier 1 or 2
-- byzantine may change id

-- -- -- channels
-- service channels
-- resp used to respond to req

-- reqId = {1..2}
channel initSession, finishSession
channel srvReq, srvResp: reqId


-- -- -- specification

S1 = initSession -> S2
S2 = S3(1) [|{|finishSession |}|] S3(2)
S3(n) = finishSession -> STOP
```

```
        []
        srvReq.n -> srvResp!n -> S3(n)

-- Design: sevice initiates and then, for each srvReq, internally
--         uses P with req and obtain response with resp,
--         then issues a servResp.
--         From time to time, a rep(ort) is generated by P and
--         consumed without impacting on the service interface
--         P is used through canal(s) as above defined
--         P is the part of the system that may suffer from failures.


-- internal communication with buffer
-- Out and In from poit of view of Service
channel reqOut, respIn: reqId
-- rep is a report that the process P may generate without req
channel repIn: reqId

Service = initSession -> Service2
Service2 = (Service3(1) [|{|finishSession |}|] Service3(2))
                        [|{|finishSession |}|] Service4
Service3(n) =
           srvReq.n -> reqOut.n -> respIn?n -> srvResp.n -> Service3(n)
           [] finishSession -> STOP
Service4 = repIn?r -> Service4
           [] finishSession -> STOP

-- P: on request, generate resp
--    eventually generates rep(ort) on the use of P through req
-- communication with P
channel req, resp: reqId
channel rep: reqId

P = Pserv [|{|req|}|] Prep(1)
Pserv = req?n -> resp!n -> Pserv
Prep(repInterval) = req?n -> if repInterval>0
                             then Prep(repInterval-1)
                             else rep.1 -> Prep(1)

APn =
   ( P [|{| req, resp, rep|} |]
     (
       UBBCanalN(reqOut, req, 1) |||
       UBBCanalN(resp, respIn, 1) |||
       UBBCanalN(rep, repIn, 1)
      )
    ) \{| req, resp, rep|}


APfl =
   ( P [|{| req, resp, rep|} |]
     (
       UBBCanalFl(reqOut, req, 1, 0) |||
       UBBCanalFl(resp, respIn, 1, 0) |||
       UBBCanalFl(rep, repIn, 1, 0)
      )
    ) \{| req, resp, rep|}
```

```
-- events to synchronize - enable failure occurrence
canalId = {1..3}
direction = {1..2}
channel cr: direction
channel om, by: canalId

APfg =
   ( ( ( P [|{| req, resp, rep|} |]
       (
        ((UBBCanalFg(reqOut, req, 1,  cr.1, cr.2, om.1, by.1)  [|{|cr|}|]
          UBBCanalFg(resp, respIn, 1, cr.2, cr.1, om.2, by.2)) [|{|cr|}|]
          UBBCanalFg(rep, repIn, 1,   cr.2, cr.1, om.3, by.3))
        )
      ) \{| req, resp, rep|}
    ) [| {| cr, om, by |} |] GFA )\{| cr, om, by |}

GFA = SKIP

assert APfl [T= APn   -- t
assert APfl [FD= APn    -- t

assert APn [T= APfl   -- t
assert APn [FD= APfl   -- t

assert APfl [T= APfg -- t
assert APfl [FD= APfg   -- t

assert APfg [T= APfl -- t
assert APfg [FD= APfl   -- t

-- The model
Dfl = (Service [| {| reqOut, respIn, repIn|} |] APfl) \{| reqOut, respIn, repIn|}

-- -- -- reasoning
assert S1 [T= Dfl  -- t
assert S1 [FD= Dfl -- t


-- -- -- -- -- -- END OF INITIAL MODEL (NO FAULTS) -- -- -- -- -- -- -- --

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
--- -- -- -- -- -- -- -- -- FAULT MODELLING -- -- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
--- General approach:   customize canal to perform as expected
--

RUN(a) = [] x : a @ x -> RUN(a)

--
-- CRASH
-- customize communication with AP to possibly stop
-- after crash no communicaiton is generated by P and
-- all communication sent to P is accepted with no effect

APfl_c =
```

```
     P [|{| req, resp, rep|} |]
     (
       ((UBBCanalFl(reqOut, req, 1, 1)  [|{|crashFl|}|]
         UBBCanalFl(resp, respIn, 1, 1))[|{|crashFl|}|]
         UBBCanalFl(rep, repIn, 1, 1)) \{|crashFl|}
      ) \{| req, resp, rep |}


APfg_c =
   ( ( ( P [|{| req, resp, rep|} |]
       (
        ((UBBCanalFg(reqOut, req, 1,  cr.1, cr.2, om.1, by.1)  [|{|cr|}|]
          UBBCanalFg(resp, respIn, 1, cr.2, cr.1, om.2, by.2)) [|{|cr|}|]
          UBBCanalFg(rep, repIn, 1,   cr.2, cr.1, om.3, by.3))
        )
      ) \{| req, resp, rep|}
    ) [| {| cr, om, by |} |] GFAc )\{| cr, om, by |}

GFAc = cr.1 -> SKIP

assert APfl_c [T= APfg_c -- t
assert APfl_c [FD= APfg_c   -- t

assert APfg_c [T= APfl_c -- t
assert APfg_c [FD= APfl_c   -- t

--
-- Receive Omission
-- customize communication with AP to possibly loose inputs
-- note that failure parameters change depending on canal

APfl_ro =
     P [|{| req, resp, rep|} |]
     (
       ((UBBCanalFl(reqOut, req, 1, 2)  [|{|crashFl|}|]
         UBBCanalFl(resp, respIn, 1, 0))[|{|crashFl|}|]
         UBBCanalFl(rep, repIn, 1, 0)) \{|crashFl|}
      ) \{| req, resp, rep|}

APfg_ro =
   ( ( ( P [|{| req, resp, rep|} |]
       (
        ((UBBCanalFg(reqOut, req, 1,  cr.1, cr.2, om.1, by.1)  [|{|cr|}|]
          UBBCanalFg(resp, respIn, 1, cr.2, cr.1, om.2, by.2)) [|{|cr|}|]
          UBBCanalFg(rep, repIn, 1,   cr.2, cr.1, om.3, by.3))
        )
        ) \{| req, resp, rep|}
    ) [| {| cr, om, by |} |] GFAro )\{| cr, om, by |}

GFAro =  -- om.1 -> GFAro
         RUN({om.1})

assert APfl_ro [T= APfg_ro -- t
assert APfl_ro [FD= APfg_ro   -- t

assert APfg_ro [T= APfl_ro -- t
assert APfg_ro [FD= APfl_ro   -- t
```

```
--
-- Send Omission
-- customize communication with AP to possibly loose outputs

APfl_so =
     P [|{| req, resp, rep|} |]
     (
       ((UBBCanalFl(reqOut, req, 1, 0)  [|{|crashFl|}|]
         UBBCanalFl(resp, respIn, 1, 2))[|{|crashFl|}|]
         UBBCanalFl(rep, repIn, 1, 2)) \{|crashFl|}
      ) \{| req, resp, rep|}

APfg_so =
   ( ( ( P [|{| req, resp, rep|} |]
       (
        ((UBBCanalFg(reqOut, req, 1,  cr.1, cr.2, om.1, by.1)  [|{|cr|}|]
          UBBCanalFg(resp, respIn, 1, cr.2, cr.1, om.2, by.2)) [|{|cr|}|]
          UBBCanalFg(rep, repIn, 1,   cr.2, cr.1, om.3, by.3))
        )
      ) \{| req, resp, rep|}
    ) [| {| cr, om, by |} |] GFAso )\{| cr, om, by |}

GFAso = RUN({om.2,om.3})
     --      om.2 -> GFAso
     --   [] om.3 -> GFAso

assert APfl_so [T= APfg_so -- t
assert APfl_so [FD= APfg_so   -- t

assert APfg_so [T= APfl_so -- t
assert APfg_so [FD= APfl_so   -- t

--
-- General Omission
-- customize communication with AP to possibly loose outputs

APfl_go =
     P [|{| req, resp, rep|} |]
     (
       ((UBBCanalFl(reqOut, req, 1, 2)  [|{|crashFl|}|]
         UBBCanalFl(resp, respIn, 1, 2))[|{|crashFl|}|]
         UBBCanalFl(rep, repIn, 1, 2)) \{|crashFl|}
      ) \{| req, resp, rep|}

APfg_go =
   ( ( ( P [|{| req, resp, rep|} |]
       (
        ((UBBCanalFg(reqOut, req, 1,  cr.1, cr.2, om.1, by.1)  [|{|cr|}|]
          UBBCanalFg(resp, respIn, 1, cr.2, cr.1, om.2, by.2)) [|{|cr|}|]
          UBBCanalFg(rep, repIn, 1,   cr.2, cr.1, om.3, by.3))
        )
     ) \{| req, resp, rep|}
    ) [| {| cr, om, by |} |] GFAgo )\{| cr, om, by |}

GFAgo = RUN({|om|})
      --     om.1 -> GFAgo
```

```
      --  [] om.2 -> GFAgo
      --  [] om.3 -> GFAgo

assert APfl_go [T= APfg_go -- t
assert APfl_go [FD= APfg_go   -- t

assert APfg_go [T= APfl_go -- t
assert APfg_go [FD= APfl_go   -- t


--
-- Byzantine
-- customize communication with AP to possibly loose outputs

APfl_b =
     P [|{| req, resp, rep|} |]
     (
       ((UBBCanalFl(reqOut, req, 1, 2)  [|{|crashFl|}|]  -- input does not change values
         UBBCanalFl(resp, respIn, 1, 3))[|{|crashFl|}|]  -- output may change
         UBBCanalFl(rep, repIn, 1, 3)) \{|crashFl|}      -- rep has no values - may not change
      ) \{| req, resp, rep|}

APfg_b =
   ( ( ( P [|{| req, resp, rep|} |]
       (
        ((UBBCanalFg(reqOut, req, 1,  cr.1, cr.2, om.1, by.1)  [|{|cr|}|]
          UBBCanalFg(resp, respIn, 1, cr.2, cr.1, om.2, by.2)) [|{|cr|}|]
          UBBCanalFg(rep, repIn, 1,   cr.2, cr.1, om.3, by.3))
       )
      ) \{| req, resp, rep|}
    ) [| {| cr, om, by |} |] GFAb )\{| cr, om, by |}

GFAb =  RUN({|om,by.2,by.3|})
     --      om.1 -> GFAb
     --   [] om.2 -> GFAb
     --   [] om.3 -> GFAb
     --   [] by.2 -> GFAb
     --   [] by.3 -> GFAb

assert APfl_b [T= APfg_b -- t
assert APfl_b [FD= APfg_b   -- t

assert APfg_b [T= APfl_b -- t
assert APfg_b [FD= APfl_b   -- t

-- analysis only with service part (rep and resp, without rep)

APfl_sr = APfl\{|repIn|}
APfl_c_sr = APfl_c\{|repIn|}
APfl_ro_sr = APfl_ro\{|repIn|}
APfl_so_sr = APfl_so\{|repIn|}
APfl_go_sr = APfl_go\{|repIn|}
APfl_b_sr = APfl_b\{|repIn|}

-- Corresponding models

Dfl_c = (Service [| {| reqOut, respIn, repIn|} |] APfl_c) \{| reqOut, respIn, repIn|}
```

```
Dfl_ro = (Service [| {| reqOut, respIn, repIn|} |] APfl_ro) \{| reqOut, respIn, repIn|}
Dfl_so = (Service [| {| reqOut, respIn, repIn|} |] APfl_so) \{| reqOut, respIn, repIn|}
Dfl_go = (Service [| {| reqOut, respIn, repIn|} |] APfl_go) \{| reqOut, respIn, repIn|}
Dfl_b = (Service [| {| reqOut, respIn, repIn|} |] APfl_b) \{| reqOut, respIn, repIn|}

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- COMPARING THE FAILURE SEMANTICS
--
-- SERVICE HAS REQ/RESP AND REPorts
--
-- CHAIN OF FAILURES FROM MORE(TOP) TO LESS(DOWN) SEVERE
--
--          Byzantine
--             |
--      GeneralOmission
--        /          \
-- SendOmission ReceiveOmission
--           \   /->T but not FD (due to reports)
--           Crash
--             |
--         NoFailure
--             |
--       OriginalModel

-- Comparisions among levels: (AP, Design) x (T, FD)
-- Direction: UP-DOWN
-- each edge above is a refinement relation below
assert APfl_b  [T= APfl_go   -- t
assert APfl_b [FD= APfl_go -- t

assert APfl_go [T= APfl_so -- t
assert APfl_go [FD= APfl_so -- t

assert APfl_go [T= APfl_ro -- t
assert APfl_go [FD= APfl_ro -- t

assert APfl_so [T= APfl_c   -- t
assert APfl_so [FD= APfl_c -- t

assert APfl_ro [T= APfl_c   -- t
assert APfl_ro [FD= APfl_c -- FALSE

assert APfl_c [T= APfl     -- t
assert APfl_c [FD= APfl   -- t


-- Comparisions among levels: (AP, Design) x (T, FD)
-- Direction: BOTTOM-UP
-- each edge above is a refinement relation below
--
assert APfl_go [T= APfl_b    -- f
assert APfl_go[FD= APfl_b  -- f

assert APfl_so [T= APfl_go -- f
assert APfl_so [FD= APfl_go -- f
```

```
assert APfl_ro [T= APfl_go -- f
assert APfl_ro [FD= APfl_go -- f

assert APfl_c  [T= APfl_so   -- f
assert APfl_c  [FD= APfl_so   -- f

assert APfl_c  [T= APfl_ro   -- f
assert APfl_c  [FD= APfl_ro   -- f

assert APfl [T= APfl_c     -- f
assert APfl [FD= APfl_c   -- f


-- comparing omissions

assert APfl_ro [T= APfl_so -- f
assert APfl_ro [FD= APfl_so -- f

assert APfl_so [T= APfl_ro -- f
assert APfl_so [FD= APfl_ro -- f

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- COMPARING THE FAILURE SEMANTICS -
--
-- WITHOUT REPORTS (REPIN) - ONLY REQ/RESP
--
-- CHAIN OF FAILURES FROM MORE(TOP) TO LESS(DOWN) SEVERE
--
--          Byzantine
--             |
--      GeneralOmission == ReceiveOmission
--        /
-- SendOmission
--           \
--            Crash
--              |
--         NoFailure
--             |
--       OriginalModel

assert APfl_b_sr  [T= APfl_go_sr   -- t
assert APfl_b_sr [FD= APfl_go_sr -- t

assert APfl_go_sr [T= APfl_so_sr -- t
assert APfl_go_sr [FD= APfl_so_sr -- t

assert APfl_go_sr [T= APfl_ro_sr -- t
assert APfl_go_sr [FD= APfl_ro_sr -- t

assert APfl_so_sr [T= APfl_c_sr   -- t
assert APfl_so_sr [FD= APfl_c_sr -- t

assert APfl_ro_sr [T= APfl_c_sr   -- t
assert APfl_ro_sr [FD= APfl_c_sr -- t

assert APfl_c_sr [T= APfl_sr   -- t
```

```
assert APfl_c_sr [FD= APfl_sr   -- t


-- Comparisions among levels: (APfl_, Design) x (T, FD)
-- Direction: BOTTOM-UP
-- each edge above is a refinement relation below
--
assert APfl_go_sr [T= APfl_b_sr    -- f
assert APfl_go_sr [FD= APfl_b_sr  -- f

assert APfl_so_sr [T= APfl_go_sr -- f
assert APfl_so_sr [FD= APfl_go_sr -- f

assert APfl_ro_sr [T= APfl_go_sr -- TRUE
assert APfl_ro_sr [FD= APfl_go_sr -- TRUE

assert APfl_c_sr  [T= APfl_so_sr   -- f
assert APfl_c_sr  [FD= APfl_so_sr   -- f

assert APfl_c_sr  [T= APfl_ro_sr   -- f
assert APfl_c_sr  [FD= APfl_ro_sr   -- f

assert APfl_sr [T= APfl_c_sr   -- f
assert APfl_sr [FD= APfl_c_sr   -- f


-- comparing omissions

assert APfl_ro_sr [T= APfl_so_sr -- TRUE
assert APfl_ro_sr [FD= APfl_so_sr -- TRUE

assert APfl_so_sr [T= APfl_ro_sr -- f
assert APfl_so_sr [FD= APfl_ro_sr -- f



--
--
-- GLOBAL
--
APfg_sr = APfg\{|repIn|}
APfg_c_sr = APfg_c\{|repIn|}
APfg_ro_sr = APfg_ro\{|repIn|}
APfg_so_sr = APfg_so\{|repIn|}
APfg_go_sr = APfg_go\{|repIn|}
APfg_b_sr = APfg_b\{|repIn|}
--
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- COMPARING THE FAILURE SEMANTICS
--
-- SERVICE HAS REQ/RESP AND REPorts
--
-- CHAIN OF FAILURES FROM MORE(TOP) TO LESS(DOWN) SEVERE
--
--          Byzantine
--             |
```

```
--      GeneralOmission
--        /          \
-- SendOmission ReceiveOmission
--       \          /->T but not FD (due to reports)
--        \       Crash
--         \      /->T but not FD ()
--          NoFailure
--              |
--       OriginalModel

-- Comparisions among levels: (AP, Design) x (T, FD)
-- Direction: UP-DOWN
-- each edge above is a refinement relation below
assert APfg_b  [T= APfg_go   -- t
assert APfg_b [FD= APfg_go -- t

assert APfg_go [T= APfg_so -- t
assert APfg_go [FD= APfg_so -- t

assert APfg_go [T= APfg_ro -- t
assert APfg_go [FD= APfg_ro -- t

assert APfg_so [T= APfg_c   -- t
assert APfg_so [FD= APfg_c -- t

assert APfg_ro [T= APfg_c   -- t
assert APfg_ro [FD= APfg_c -- FALSE

assert APfg_c [T= APfg     -- t
assert APfg_c [FD= APfg   -- t

assert APfg_so [T= APfg   -- t
assert APfg_so [FD= APfg -- t


assert APfg [T= APfg_so   -- t
assert APfg [FD= APfg_so -- t

-- Comparisions among levels: (AP, Design) x (T, FD)
-- Direction: BOTTOM-UP
-- each edge above is a refinement relation below
--
assert APfg_go [T= APfg_b    -- f
assert APfg_go[FD= APfg_b  -- f

assert APfg_so [T= APfg_go -- f
assert APfg_so [FD= APfg_go -- f

assert APfg_ro [T= APfg_go -- f
assert APfg_ro [FD= APfg_go -- f

assert APfg_c  [T= APfg_so   -- f
assert APfg_c  [FD= APfg_so   -- f

assert APfg_c  [T= APfg_ro   -- f
assert APfg_c  [FD= APfg_ro   -- f
```

```
assert APfg [T= APfg_c     -- f
assert APfg [FD= APfg_c   -- f


-- comparing omissions

assert APfg_ro [T= APfg_so -- f
assert APfg_ro [FD= APfg_so -- f

assert APfg_so [T= APfg_ro -- f
assert APfg_so [FD= APfg_ro -- f

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- COMPARING THE FAILURE SEMANTICS -
--
-- WITHOUT REPORTS (REPIN) - ONLY REQ/RESP
--
-- CHAIN OF FAILURES FROM MORE(TOP) TO LESS(DOWN) SEVERE
--
--          Byzantine
--             |
--      GeneralOmission == ReceiveOmission
--        /
-- SendOmission
--           \
--            Crash
--              |
--         NoFailure
--              |
--       OriginalModel

assert APfg_b_sr  [T= APfg_go_sr   -- t
assert APfg_b_sr [FD= APfg_go_sr -- t

assert APfg_go_sr [T= APfg_so_sr -- t
assert APfg_go_sr [FD= APfg_so_sr -- t

assert APfg_go_sr [T= APfg_ro_sr -- t
assert APfg_go_sr [FD= APfg_ro_sr -- t

assert APfg_so_sr [T= APfg_c_sr   -- t
assert APfg_so_sr [FD= APfg_c_sr -- t

assert APfg_ro_sr [T= APfg_c_sr   -- t
assert APfg_ro_sr [FD= APfg_c_sr -- t

assert APfg_c_sr [T= APfg_sr   -- t
assert APfg_c_sr [FD= APfg_sr   -- t

assert APfg_so_sr [T= APfg_sr   -- t
assert APfg_so_sr [FD= APfg_sr -- t


-- Comparisions among levels: (APfg_, Design) x (T, FD)
-- Direction: BOTTOM-UP
-- each edge above is a refinement relation below
```

```
--
assert APfg_go_sr [T= APfg_b_sr    -- f
assert APfg_go_sr [FD= APfg_b_sr  -- f

assert APfg_so_sr [T= APfg_go_sr -- f
assert APfg_so_sr [FD= APfg_go_sr -- f

assert APfg_ro_sr [T= APfg_go_sr -- TRUE
assert APfg_ro_sr [FD= APfg_go_sr -- TRUE

assert APfg_c_sr  [T= APfg_so_sr   -- f
assert APfg_c_sr  [FD= APfg_so_sr   -- f

assert APfg_c_sr  [T= APfg_ro_sr   -- f
assert APfg_c_sr  [FD= APfg_ro_sr   -- f

assert APfg_sr [T= APfg_c_sr   -- f
assert APfg_sr [FD= APfg_c_sr   -- f


-- comparing omissions

assert APfg_ro_sr [T= APfg_so_sr -- TRUE
assert APfg_ro_sr [FD= APfg_so_sr -- TRUE

assert APfg_so_sr [T= APfg_ro_sr -- f
assert APfg_so_sr [FD= APfg_ro_sr -- f

--
```