Transforming Asynchronous Systems with Crash-Stop Failures and Failure Detectors to the General Omission Model*

Carole Delporte-Gallet[1] , Hugues Fauconnier[1], Felix Freiling[2]
Lucia Draque Penso[2], Andreas Tielmann[1]

[1] LIAFA University of Paris 7 - Denis Diderot
[2] University of Mannheim
Laboratory for Dependable Distributed Systems

University of Paris 7
LIAFA
Technical Report TR 2007-001
&
University of Mannheim
Department for Mathematics and Computer Science
Technical Report TR 2007-001

February, 2007

Abstract

This paper studies the impact of omission failures on asynchronous distributed systems with crash-stop failures. For the large group of problem specifications that are restricted to correct processes, we show how to transform a crash-stop related problem specification into an equivalent omission one. For that, we provide transformations for algorithms and failure detectors, such that if and only if an algorithm using a failure detector satisfies a problem specification, then the transformed algorithm using the transformed failure detector satisfies the transformed problem specification. Our transformed problem specification is ensured to be non-trivial, and moreover, the transformation reveals itself to be in a reasonable sense weakest failure detector preserving.

Our results help to use the power of the well-understood crash-stop model to automatically derive solutions for the general omission model, which has recently raised interest for being noticeably applicable for security problems in distributed environments equipped with security modules such as smartcards [10], [11], [1].

*Work of the authors was supported by the PROCOPE-project.

Keywords: Transformations, Asynchronous Systems, Failure Detectors, Crash-Stop, General Omission

1 Introduction

Message omission failures, which have been introduced in [12] and been refined in [16] put the blame of a message loss to a specific process instead of an unreliable message channel. This property has lead to the development of reductions from security problems in the byzantine failure model [13], such as eletronic commerce and voting [11], fair-exchange [1] and secure multiparty computation [10], to well-known distributed problems in the general omission model, such as consensus [6], where both process crashes and message omissions may take place. The general omission model can be obtained from the byzantine failure model by considering processes as parties that contain a tamper proof security module, such as a smartcard, which executes randomized [11] or deterministic [15] distributed algorithms and exchanges authenticated and cryptographed messages. Hence, in such a scenario, note that security module messages must go through their untrusted hosting parties, which are able to drop messages but not modify them. Therefore, the only failures that may appear to the trusted security modules are message omissions and process crashes (due to self-destruction of the security modules or destruction by a malicious party). Thus, since the blame of a message drop (an omission) is put only at the security modules of the malicious parties, the security modules of the honest parties remain correct (i.e. failure-free), even if another party decides to drop messages from/to it.

In this paper, we want to provide the general omission model with the benefits of a well-understood system model like the crash-stop model. We show, that in asynchronous systems, both models are equivalent for problem specifications that are only related to correct processes[1]. This means, that we are able to transform an algorithm that uses a failure detector (as introduced in [5]) to solve a problem specification in the crash-stop model into an algorithm that uses a transformed failure detector to satisfy the equivalent problem in the general omission model. We show the problem equivalence by proving also the inverse implication, that is, that if a transformed algorithm using a transformed failure detector satisfies a transformed problem specification, then the original algorithm using the original failure detector solves the original problem specification (see also Figure 1). Moreover, our transformation preserves also the "is weaker than" relation [4] between failure detectors. This means, that if a failure detector is a weakest failure detector for a certain (crash-stop) problem, then its transformation is a weakest of all transformed failure detectors for the transformed problem.

For clarity, we use the term correct only for processes that do not make any failure at all. Processes that do not crash are called crash-correct and processes that are able to send

[1] This type of problem specifications are somehow similar to non-uniform specifications. They are defined similarly to the ones in [3].

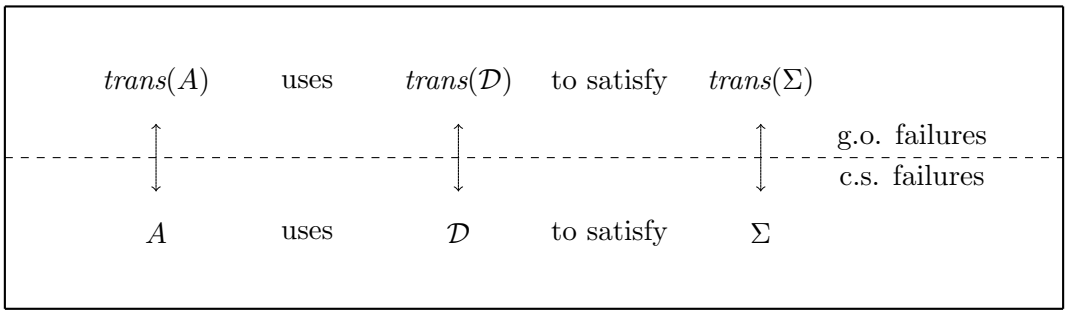

Figure 1: The Three Transformations

and receive messages to/from correct processes (possibly indirect) are called connected. This definition of connected processes has been introduced in [8]. To simplify, we consider only permanent omissions. Note that this restriction is not limiting, as we simply delegate the masking of transient omissions to the underlying asynchronous communication layer.

The intuition behind our problem transformation is to exchange every crash-correct in the specification with crash-correct & connected (and likewise for the failure detector transformation). This means, that everything demanded for crash-correct processes in the original problem specification is now demanded for crash-correct & connected processes. To transform a crash-stop algorithm into one that is able to satisfy such a transformed specification, we augment it with two additional send-primitives (added as new communication layers). With these new primitives, we are able to "simulate" a crash-stop environment for the transformed algorithms. For crash-correct & connected processes, reliable communication is possible and not crash-correct & connected processes are not able to influence them.

Since it is in general not possible to provide guarantees for processes that are not connected, we limit ourselves to transform only problem specifications that refer exclusively to the state of processes before they crash. As an extension of our result, it would be possible to consider the specific case where only less than half of the processes are allowed to crash. In such a scenario, it would be possible for a process to check whether it is connected or not. It simply has to ping all other processes (e.g., before every step) and if it receives from less than the majority of the processes an answer, it keeps waiting forever and satisfies at least the safety properties of a problem specification.

The problem of automatically increasing the fault tolerance of problems in environments with crash-stop failures has been extensively studied before (e.g., [2], [14], [7], or [3]). The results of [14], [7], and [3] assume in contrast to ours synchronous systems and no failure detectors. In [14], several transformations from crash-stop to send omission, to general omission, and to Byzantine faults are proposed. In [7], round-based algorithms with broadcast primitives are transformed into crash-stop-, general omission-, and Byzantine-

tolerant algorithms. Asynchronous systems are considered in [2], but in the context of link failures instead of omission failures and also without failure detectors. The types of link failures that are considered in [2] are eventually reliable and fair-lossy links. Eventually reliable links can lose a finite (but unbounded) number of messages and fair-lossy links satisfy that if infinitely many messages are sent over it, then infinitely many messages do not get lost. To show our results, we extend the system model of [2] such that we can model omission failures, failure patterns, and failure detectors. Another definition for a system model with crash-recovery failures[2], omission failures, and failure detectors is given in [9].

To the best of our knowledge, this is the first paper that investigates an automatic transformation to increase the fault tolerance of distributed algorithms in asynchronous systems augmented with failure detectors. We here give a transformation from the crash-stop model to the general omission model.

We organize this paper as follows. In section 2, we define our formal system model, in section 3, we define our general problem and algorithm transformations, in section 4 we define our main theorem and sketch the proof, and finally, in section 5, we summarize and discuss our results. The detailed proof can be found in the appendix.

2 Model

The asynchronous distributed system is assumed to consist of n distinct processes $\Pi = \{p_1, \ldots, p_n\}$. Each pair p_i, p_j of processes is connected via a direct communication channel. The asynchrony of the system means, that there are no bounds on the relative process speeds and message transmission delays. To allow an easier reasoning, a discrete global clock $\mathcal{T}$ is added to the system. The discrete range of the clock ticks is the set of natural numbers $\mathbb{N}$. The processes do not have access to the clock, it is only used for making statements about the system. The system model used here is derived from that in [2]. It has been adapted to model also failure detectors and permanent omission failures.

Algorithms An *algorithm* A is defined as a vector of *local algorithm modules* (or simply *modules*) $A(\Pi) = \langle A(p_1), \ldots, A(p_n) \rangle$. Each local algorithm module $A(p_i)$ is associated with a process $p_i \in \Pi$ and defined as a deterministic infinite state automaton. The local algorithm modules can exchange messages via send and receive primitives. We assume all messages to be unique.

Histories A *local history of a local algorithm module* $A(p_i)$, denoted $H_A[i]$, is a finite or an infinite sequence $s_i^0\ e_i^1\ s_i^1\ e_i^2\ s_i^2 \ldots$ of alternating states and events of type send, receive, queryFD, or internal. We define $H_A[i]/_t$ to be the maximal prefix of $H_A[i]$ where

[2] In an environment with crash-recovery failures, crashed processes are allowed to recover and participate again in the distributed computation.

all events have occurred before time t. A *history* H_A of $A(\Pi)$ is a vector of local histories $\langle H_A[1], H_A[2], \ldots, H_A[n] \rangle$.

Failures and Failure Patterns A *failure pattern* $\mathcal{F}$ is a function that maps each value t from $\mathcal{T}$ to an output value that specifies which failures have occurred up to time t during an execution of a distributed system. Such a failure pattern is totally independent of any algorithm. A *crash-failure pattern* $C : \mathcal{T} \rightarrow 2^{\Pi}$ denotes the set of processes that have crashed up to time t ($\forall t : C(t) \subseteq C(t+1)$).

Additionally to the crash of a process, it can fail by not sending or not receiving a message. We say that it *omits* a message. The message omissions do not occur because of link failures, they model overflows of local message buffers or the behavior of a malicious adversary with control over the message flow of certain processes. It is important that for every omission, there is a process responsible for it. As we already mentioned, we consider only permanent omissions and leave the treatment of transient omissions over to the underlying asynchronous communication layer. There are two types of permanent omissions: permanent send omissions and permanent receive omissions. Intuitively, a process has a permanent send omission if it always fails by not sending messages to a certain other process after a certain point in time. Analogously, a process has a permanent receive omission if it always fails by not receiving messages from a certain other process after a certain point in time. In the following, to do not get confused, we use p_s if we think of the sending process and p_d for the destination. The permanent omissions are modeled via send-/receive-omission failure patterns:

$$O_S : \mathcal{T} \rightarrow 2^{\Pi \times \Pi} \quad \text{and} \quad O_R : \mathcal{T} \rightarrow 2^{\Pi \times \Pi}$$

If $(p_s, p_d) \in O_S(t)/O_R(t)$, then process p_s/p_d has a permanent send/receive-omission to process p_d/p_s at time t. All the failure patterns defined so far can be put together to a single failure pattern $\mathcal{F} = (C, O_S, O_R)$.

We here define some predicates processes might fulfill depending on the failure pattern and the time t.

$$\begin{aligned}
\textit{crashed}(\mathcal{F}, t) &:= \{p \mid p \in C(t)\} \\
\textit{crash-correct}(\mathcal{F}, t) &:= \{p \mid p \notin C(t)\} \\
\textit{send-omissive}(\mathcal{F}, t) &:= \{p_s \mid \exists p_d : (p_s, p_d) \in O_S(t)\} \\
\textit{receive-omissive}(\mathcal{F}, t) &:= \{p_d \mid \exists p_s : (p_s, p_d) \in O_R(t)\}. \\
\textit{omissive}(\mathcal{F}, t) &:= \textit{send-omissive}(\mathcal{F}, t) \cup \textit{receive-omissive}(\mathcal{F}, t)
\end{aligned}$$

The following predicates are used to formalize our notion of connected processes. We first define a process p to be directly-reachable from a process q, if every message sent by q to p will be received by p. This means, that there occurs no omission in the direction from q to p and the crash-correctness of q implies the crash-correctness of p. Reachable is then the

transitive closure of directly-reachable. To define the connected processes, we formalize the notion of "are able to send/receive messages to/from correct processes" with the definition of out-/in-connected. More formally:

$$\begin{aligned}
\textit{directly-reachable}(\mathcal{F},t) &:= \{(p,q) \mid (q,p) \notin O_S(t) \wedge (p,q) \notin O_R(t) \\
&\wedge \quad q \in \textit{crash-correct}(\mathcal{F},t) \rightarrow p \in \textit{crash-correct}(\mathcal{F},t)\} \\
\textit{reachable}(\mathcal{F},t) &:= \{(p_d,p_s) \mid (p_d,p_s) \in \textit{directly-reachable}(\mathcal{F},t) \\
&\vee \quad \exists r \in \Pi : ((p_d,r) \in \textit{reachable}(\mathcal{F},t) \wedge (r,p_s) \in \textit{reachable}(\mathcal{F},t))\} \\
\textit{in-connected}(\mathcal{F},t) &:= \{p_d \mid \exists c \in \textit{correct}(\mathcal{F}) : (p_d,c) \in \textit{reachable}(\mathcal{F},t)\} \\
\textit{out-connected}(\mathcal{F},t) &:= \{p_s \mid \exists c \in \textit{correct}(\mathcal{F}) : (c,p_s) \in \textit{reachable}(\mathcal{F},t)\} \\
\textit{connected}(\mathcal{F},t) &:= \textit{in-connected}(\mathcal{F},t) \cap \textit{out-connected}(\mathcal{F},t)
\end{aligned}$$

Instead of *crash-correct*, we write also cc and instead of *crash-correct*$\cap$ *connected*, we write *cc+c*. We define for every predicate φ:

$$\varphi(\mathcal{F}) := \bigcup_{t \in \mathcal{T}} \{\varphi(\mathcal{F},t) \mid \forall t' \geq t : \varphi(\mathcal{F},t) = \varphi(\mathcal{F},t')\} \quad \text{(e.g., } \varphi = \textit{connected}\text{)}.$$

This means, that $\varphi(\mathcal{F})$ is the set where the failure pattern does not change anymore (at least in relevance to φ). We define the point in time when a process stops/some processes stop fulfilling a predicate φ:

$$t_{not(\varphi,\mathcal{F},p,q,\ldots)} := \max\{t \mid (p,q,\ldots) \in \varphi(\mathcal{F},t)\}.$$

If $(p,q,\ldots) \in \varphi(\mathcal{F})$, then we say that $t_{not(\varphi,\mathcal{F},p,q,\ldots)} = \infty$.

We define an *environment* $\mathcal{E}$ to be a set of possible failure patterns. $\mathcal{E}^t_{c.s.}$ denotes the set of all failure patterns where only crash-stop faults occur and at most t processes crash. $\mathcal{E}^t_{g.o.}$ denotes the set of all failure patterns where crash-stop and omission faults may occur and at most t processes are not crash-correct and connected (clearly, $\mathcal{E}^t_{c.s.} \subseteq \mathcal{E}^t_{g.o.}$).

Failure Detectors A failure detector provides (possibly incorrect) information about the failure pattern that occurs in an execution [5]. A *failure detector history FDH with range* $\mathcal{R}$ is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}$. *FDH*(p,t) is the value of the failure detector module of process p at time t. A *failure detector* $\mathcal{D}$ is a function that maps a failure pattern $\mathcal{F}$ to a *set* of failure detector histories with range $\mathcal{R}_\mathcal{D}$. $\mathcal{D}(\mathcal{F})$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern $\mathcal{F}$.

Reliable Links A reliable link does not create, duplicate, or lose messages. Formally, the link from p_i to p_j is *reliable in history H according to failure pattern* $\mathcal{F}$, if H satisfies:

L1: *(No Creation)* For all messages m, if p_j receives m from p_i, then p_i sends m to p_j.

L2: *(No Duplication)* For all messages m, p_j receives m from p_i at most once.

L3: *(No Loss)* For all messages m, if p_i sends m to p_j, $(p_i, p_j) \notin O_S(\mathcal{F})$, $(p_j, p_i) \notin O_R(\mathcal{F})$, and p_j executes receive actions infinitely often, then p_j receives m from p_i.

We specify, that our underlying communication channels ensure reliable links.

Problem Specifications Let Π be a set of processes and A be an algorithm. We define $\mathcal{H}(A(\Pi), \mathcal{E})$ to be the set of all tuples $(H_A, \mathcal{F})$ such that H_A is a history of $A(\Pi)$ and $\mathcal{F} \in \mathcal{E}$. A *system* $\mathcal{S}(A(\Pi), \mathcal{E})$ of $A(\Pi)$ is a subset of $\mathcal{H}(A(\Pi), \mathcal{E})$. A *problem specification* Σ is a set of tuples of histories and failure patterns and a system $\mathcal{S}$ satisfies a problem specification Σ, if $\mathcal{S} \subseteq \Sigma$. Take Consensus as an example: It is specified by making statements about some variables *propose* and *decide* in the states of a history (e.g., the value of *decide* has eventually to be equal at all (crash-)correct processes).

We say that a problem specification Σ is *cc-restricted*, if for all $(H, \mathcal{F}) \in \Sigma$, and for all $(H', \mathcal{F}')$ with $H[i]/_{t_{not(cc,\mathcal{F},p_i)}} = H'[i]/_{t_{not(cc,\mathcal{F}',p_i)}}$ (for all p_i), the following holds: $(H', \mathcal{F}') \in \Sigma$. Analogously, a problem specification Σ is cc+c-restricted, if for all $(H, \mathcal{F}) \in \Sigma$, and for all $(H', \mathcal{F}')$ with $H[i]/_{t_{not(cc+c,\mathcal{F},p_i)}} = H'[i]/_{t_{not(cc+c,\mathcal{F}',p_i)}}$ (for all p_i): $(H', \mathcal{F}') \in \Sigma$.

Intuitively, a cc-restricted problem specification makes only assumptions about cc processes. Consider the specification of non-uniform Consensus (see Table 1). The three properties validity, agreement, and termination are only related to correct processes (and the initial states of all processes) and therefore, the states of a process p_i after its time of disconnection ($t_{not(cc+c,\mathcal{F},p_i)}$) does not have any influence on whether Consensus is reached or not. Therefore, non-uniform Consensus is cc-restricted.

3 From Crash-Stop to General Omission

To improve the fault-tolerance of algorithms, we simulate a single state of the original algorithm with several states of the simulation algorithm. For these additional states, we *augment* the original states with additional variables. Since an event of the simulation algorithm may lead to a state where only the augmentation variables change, the sequence of the original variables may *stutter*. We call a history H' a stuttered and augmented extension of a history H ($H \leq_{sa} H'$), if H and H' differ only in the value of the augmentation variables and all additional states are caused by differences in these variables.

Transformation of Problem Specifications To transform a problem specification, we first show a transformation of a tuple of a trace and a failure pattern. Based on this transformation, we transform a whole problem specification. The intuition behind this transformation is that for crash-correct restricted problem specifications, everything demanded for crash-correct processes is only demanded for crash-correct & connected processes after

the transformation. More formally:

$$(H', \mathcal{F}') \in \mathit{trans}((H, \mathcal{F})) \quad \Leftrightarrow \quad \forall p_i \in \Pi : \; H[i]/_{t_{not(cc,\mathcal{F},p_i)}} \leq_{sa} H'[i]/_{t_{not(cc+c,\mathcal{F}',p_i)}}$$

This implies that for all p_i: $t_{not(cc,\mathcal{F},p_i)} \leq t_{not(cc+c,\mathcal{F}',p_i)}$.

$$\mathit{trans}(\Sigma) = \{(H', \mathcal{F}') \mid (H', \mathcal{F}') \in \mathit{trans}((H, \mathcal{F})) \wedge (H, \mathcal{F}) \in \Sigma\}$$

A transformation of non-uniform Consensus, where properties of certain propose- and decision-variables of (crash-)correct processes are specified would lead to a specification where the same properties are ensured for the states of crash-correct & connected processes, because only histories with the same states (disregarding the augmentation variables) are allowed in the transformation at this processes (see also table 1). We also take the states of processes *before* they become disconnected into account, because they (e.g., their initial states for the propose variables) may also have an influence on the fulfillment of a problem specification, although they are after their disconnection not allowed to have this influence anymore.

	Consensus	*trans*(Consenus)
Validity:	The decided value of every cc process must have been proposed.	The decided value of every *cc+c* process must have been proposed.
Agreement:	No two cc processes decide differently.	No two *cc+c* processes decide differently
Termination:	Every cc process eventually decides.	Every *cc+c* process eventually decides

Table 1: Transformation of non-uniform Consensus

Transformation of Failure Detector Specifications We allow all failure detector histories for a failure pattern $\mathcal{F}$ in $\mathit{trans}(\mathcal{D})$ that are allowed in the crash-stop version of $\mathcal{F}$ in $\mathcal{D}$:

$$\mathit{trans}(\mathcal{D})(\mathcal{F}) := \bigcup_{\mathcal{F}'} \{\mathcal{D}(\mathcal{F}') \mid \forall t : cc(\mathcal{F}', t) = cc{+}c(\mathcal{F}, t)\}$$

If we take an Ω failure detector [4] which outputs only failure detector histories with a cc common leader at all cc processes as example, then the transformed failure detector outputs these failure detector histories if and only if they provide a *cc+c* common leader at all *cc+c* processes.

Transformation of Algorithms In our algorithm transformation, we add new communication layers such that some of the omission failures in the system become transparent to the algorithm (see Figure 2). We transform a given algorithm A into another algorithm $A' = \mathit{trans}(A)$ in two steps:

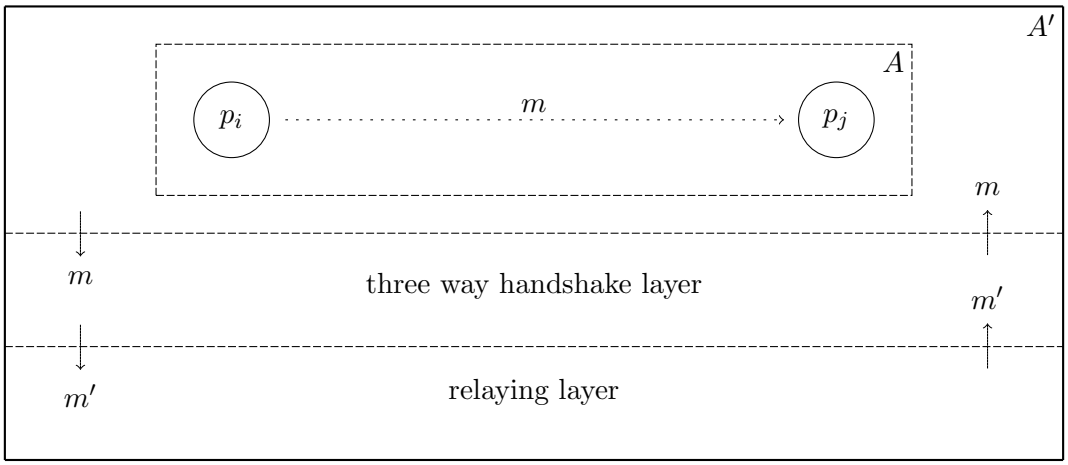

Figure 2: Additional Communication Layers

- In the first step, we remove the send and receive actions from A and simulate them with a *three-way-handshake (3wh) algorithm*. The algorithm is described in Figure 3. The idea of the 3wh-algorithm is to substitute every send-action with an exchange of three messages. This means, that to send a message to a certain process, it is necessary for a process to be able to send *and* to receive messages from it. Moreover, while the communication between connected processes is still possible, processes that are only in-connected or only out-connected (and not both) become totally disconnected. Hence, we eliminate influences of disconnected processes not existing in the crash-stop case.

- Then, in the second step, we remove the send and receive actions from the three way handshake algorithm and simulate them with a *relaying algorithm*. The relaying algorithm is described in Figure 4. The idea of the relay algorithm is to relay every message to all other processes, such that they relay it again and all crash-correct & connected processes can communicate with each other, despite the fact that they are not directly-reachable.

To execute the simulation algorithms in parallel with the actions from A, we add some new (augmentation) variables to the set of variables in the states of A. Whenever a step of the simulation algorithms is executed, the state of the original variables in A remains untouched and only the new variables change their values. Whenever a process queries a local failure detector module $\mathcal{D}(p_i)$, we translate it to a query on $trans(\mathcal{D})(p_i)$.

4 Proof

Algorithm *3wh*

1: **upon event** $\langle$*3wh-send*$(p_i, m, p_j)\rangle$ **do**
2: **trigger** $\langle send(p_i, [1, m], p_j)\rangle$;
3:
4: **upon event** $\langle deliver(p_j, [l, m], p_i)\rangle$ **do**
5: **if** $(l = 1)$ **then**
6: **trigger** $\langle send(p_i, [2, m], p_j)\rangle$;
7: **elseif** $(l = 2)$ **then**
8: **trigger** $\langle send(p_i, [3, m], p_j)\rangle$;
9: **elseif** $(l = 3)$ **then**
10: **trigger** $\langle$*3wh-deliver*$(p_j, m, p_i)\rangle$;

Figure 3: The Three Way Handshake Algorithm for Process p_i.

Main Theorem We now define the main theorem of this work. Assume a problem specification Σ is cc-restricted. Then, if and only if there is an algorithm A that satisfies Σ using a failure detector $\mathcal{D}$ in an environment with at most t crash-stop failures ($0 \leq t \leq n$) and no omission failures, then *trans*(A) satisfies *trans*(Σ) using *trans*$(\mathcal{D})$ in an environment where at most t processes are not crash-correct & connected. This theorem does not only show that our transformation works, it furthermore ensures that we do not transform to a trivial problem specification, but to an equivalent one, since we prove both directions. We say, that $\mathcal{S}^t_{c.s.}(A) := \mathcal{H}(A(\Pi), \mathcal{E}^t_{c.s.})$ and $\mathcal{S}^t_{g.o.}(A') := \mathcal{H}(A'(\Pi), \mathcal{E}^t_{g.o.})$.

Theorem 1. *Let Σ be a cc-restricted problem specification. Then, if A is an algorithm using a failure detector $\mathcal{D}$ and $A' =$ trans(A) is the transformation of A using trans$(\mathcal{D})$, it holds that:*

$$\forall t \text{ with } 0 \leq t \leq n : \ \mathcal{S}^t_{c.s.}(A) \subseteq \Sigma \ \Leftrightarrow \ \mathcal{S}^t_{g.o.}(A') \subseteq trans(\Sigma)$$

Proof. Because of the lack of space, we will only give an intuition of the proof here and postpone the formal proof into the appendix. We divide up the proof into two parts. Let $\mathcal{S}_{c.s.} := \mathcal{S}^t_{c.s.}(A)$ and $\mathcal{S}_{g.o.} := \mathcal{S}^t_{g.o.}(A')$ and assume that $A' = trans(A)$.

"⇒": Assume that $\mathcal{S}_{c.s.} \subseteq \Sigma$. By constructing for a given $(H, \mathcal{F})$ in $\mathcal{S}_{g.o.}$ a tuple $(H', \mathcal{F}')$ in $\mathcal{S}_{c.s.}$ with $(H, \mathcal{F}) \in trans((H', \mathcal{F}'))$, we can show that $\mathcal{S}_{g.o.} \subseteq trans(\mathcal{S}_{c.s.})$ (Proposition 1 in the appendix). In this construction, we remove the added communication layers from H and use the properties of our two send-primitves to prove the reliability of the links in H'. We ensure "No Loss" with the relaying algorithm and "No Creation" with the three way handshake algorithm. As we know from the definition of *trans*, that $trans(\mathcal{S}_{c.s.}) \subseteq trans(\Sigma)$, we can conclude that $\mathcal{S}_{g.o.} \subseteq trans(\Sigma)$.

```
Algorithm Relay
1:  upon event ⟨init⟩ do
2:      relayed_i := ∅;
3:      delivered_i := ∅;
4:
5:  upon event ⟨relay-send(p_i, m, p_j)⟩ do
6:      for k := 1 to n do
7:          trigger ⟨send(p_i, [m, p_j], p_k)⟩;
8:      relayed_i := relayed_i ∪ {[m, p_j]};
9:
10: upon event ⟨deliver(p_j, [m, p_k], p_i)⟩ do
11:     if (k = i) and (m ∉ delivered_i) then
12:         trigger ⟨relay-deliver(sender(m), m, p_i)⟩;
13:         delivered_i := delivered_i ∪ {m};
14:     elseif (k ≠ i) and ([m, p_k] ∉ relayed_i) then
15:         for l := 1 to n do
16:             trigger ⟨send(p_i, [m, p_k], p_l)⟩;
17:         relayed_i := relayed_i ∪ {[m, p_k]};
```

Figure 4: The Relaying Algorithm for Process p_i.

"⇐": Assume that $\mathcal{S}_{g.o.} \subseteq \mathit{trans}(\Sigma)$. We construct a history H' for all histories H in $\mathcal{S}_{c.s.}$, such that H' is in $\mathcal{S}_{g.o.} \subseteq \mathit{trans}(\Sigma)$. Together with the fact, that Σ is cc restricted, we can use this to prove that $\mathcal{S}_{c.s.} \subseteq \Sigma$ (Proposition 2 in the appendix).

□

Weakest Failure Detectors A failure detector [4] is a weakest failure detector for a problem specification, if it is necessary and sufficient. Sufficient means, that there exists an algorithm using this failure detector that satisfies the problem specification, whereas necessary means, that every other sufficient failure detector is reducible to it. In the appendix we prove quite straighforwardly that our transformations preserve the weakest failure detector property at least according to the class of transformed failure detectors.

Theorem 2. *If $\mathcal{D}$ is the weakest failure detector for Σ, then $\mathit{trans}(\mathcal{D})$ is the weakest transformed failure detector for $\mathit{trans}(\Sigma)$.*

5 Summary

We have given transformations for algorithms, failure detectors, and problem specifications, so crash-stop resilient algorithms can be automatically enhanced to tolerate the more severe general omission failures, highly applicable in practical settings running security problems.

References

[1] Gildas Avoine, Felix C. Gärtner, Rachid Guerraoui, and Marko Vukolic. Gracefully degrading fair exchange with security modules. In *The 5th European Dependable Computing Conference (EDCC)*, pages 55–71, 2005.

[2] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings in the 10th International Workshop on Distributed Algorithms (WDAG96)*, pages 105–122, 1996.

[3] Rida A. Bazzi and Gil Neiger. Simulating crash failures with many faulty processors (extended abstract). In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 166–184, London, UK, 1992. Springer-Verlag.

[4] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In Maurice Herlihy, editor, *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92)*, pages 147–158, Vancouver, BC, Canada, 1992. ACM Press.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[6] Soma Chaudhuri. Agreement is harder than consensus: set consensus problems in totally asynchronous systems. In *Proceedings of Principles of Distributed Computing 1990*, 1990.

[7] C. Delporte-Gallet, R. Guerraoui H. Fauconnier, and B. Pochon. The perfectly-synchronised round-based model of distributed computing (to appear). *Information & Computation*, 2007.

[8] Carole Delporte-Gallet, Hugues Fauconnier, and Felix C. Freiling. Revisiting failure detection and consensus in omission failure environments. In *ICTAC*, pages 394–408, 2005.

[9] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Brief announcement: Failure detectors in omission failure environments. In *Symposium on Principles of Distributed Computing*, page 286, 1997.

[10] Milan Fort, Felix Freiling, Lucia Draque Penso, Zinaida Benenson, and Dogan Kesdogan. Trustedpals: Secure multiparty computation implemented with smartcards. In *ESORICS '06: 11th European Symposium On Research In Computer Security*, pages 34–48, Hamburg, Germany, 2006. Springer-Verlag.

[11] Felix Freiling, Maurice Herlihy, and Lucia Draque Penso. Optimal randomized omission-tolerant uniform consensus in message passing systems. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, December 2005.

[12] Vassos Hadzilacos. Ph.d. thesis, Harvard University, 1984. Technical report TR11-84.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. 4(3):382–401, July 1982.

[14] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[15] Philippe Parvedy and Michel Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 302–310, New York, NY, USA, 2004. ACM Press.

[16] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Softw. Eng.*, 12(3):477–482, 1986.

A Formal Proof

Theorem 2. *If $\mathcal{D}$ is the weakest failure detector for Σ, then trans($\mathcal{D}$) is the weakest transformed failure detector for trans(Σ).*

Proof. If $\mathcal{D}$ is the weakest failure detector for Σ, then *trans*($\mathcal{D}$) is sufficient for *trans*(Σ) (Theorem 1). Assume a transformed failure detector $\mathcal{D}'' = \mathit{trans}(\mathcal{D}')$ is sufficient for *trans*(Σ). Then, we know, that $\mathcal{D}'$ is sufficient for Σ (Theorem 1)) and moreover, $\mathcal{D}'$ is reducible to $\mathcal{D}$ (since $\mathcal{D}$ is the weakest failure detector for Σ). If the reduction algorithm is $T_{\mathcal{D}' \to \mathcal{D}}$, then $\mathit{trans}(T_{\mathcal{D}' \to \mathcal{D}})$ using *trans*($\mathcal{D}'$) emulates the outputs of *trans*($\mathcal{D}$). Therefore, $\mathcal{D}'' = \mathit{trans}(\mathcal{D}')$ is reducible to *trans*($\mathcal{D}$). □

Proposition 1. $\mathcal{S}_{g.o.} \subseteq \mathit{trans}(\mathcal{S}_{c.s.})$

Proof. The proposition is equivalent to

$$(H, \mathcal{F}) \in \mathcal{S}_{g.o.} \Rightarrow (H, \mathcal{F}) \in \mathit{trans}(\mathcal{S}_{c.s.})$$

From the definition of *trans* follows:

$$(H, \mathcal{F}) \in \mathcal{S}_{g.o.} \quad \Rightarrow \quad \exists (H', \mathcal{F}') \in \mathcal{S}_{c.s.} : \forall p_i \in \Pi : \; H'[i]/_{t_{not(cc,\mathcal{F}',p_i)}} \leq_{sa} H[i]/_{t_{not(cc+c,\mathcal{F},p_i)}} (1)$$

We will in the following construct a new history H' and a failure pattern $\mathcal{F}'$ from H and $\mathcal{F}$ which satisfy equation (1):

(a) At first, we undo step 2 of the transformation and remove the variables, additional states, and events of the relaying algorithm from H. This means, that every time a relay-send or relay-receive event in H occurs, this event is substituted by an send/receive event of the underlying communication channel. We let the inserted events take place at the time when the relay events have been completed (since a process may take several steps to accomplish the relaying task). We call the intermediate history we get after this H_1.

(b) Then, we undo step 1 and remove the variables, additional states, and events of the three way handshake algorithm from H_1 (in the same way as above). We call this intermediate history H_2.

(c) After that, we construct $\mathcal{F}'$, such that $\forall t : cc(\mathcal{F}', t) = cc{+}c(\mathcal{F}, t) \wedge \mathit{omissive}(\mathcal{F}') = \emptyset$. To build H' from H_2, we substitute every query on a failure detector *trans*($\mathcal{D}$) in H_2 with a query on $\mathcal{D}$ in H' and remove all states and events for every process p_i that occur after time $t_{not(cc,\mathcal{F}',p_i)}$.

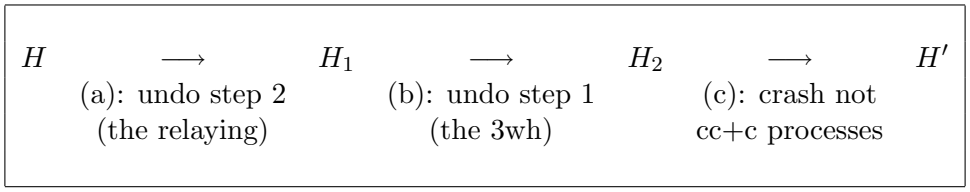

Figure 5: Construction of H'

The schedule of the construction is illustrated in Figure 5. From the construction of H' and $\mathcal{F}'$ it is clear, that $\forall p_i \in \Pi : \; H'[i]/_{t_{not(cc,\mathcal{F}',p_i)}} \leq_{sa} H[i]/_{t_{not(cc+c,\mathcal{F},p_i)}}$. It remains to show, that $(H', \mathcal{F}') \in \mathcal{S}_{c.s.}$. This means, that at most t processes crash in $\mathcal{F}'$ (Lemma 1), H' is a history of $A(\Pi)$ using $\mathcal{D}$ (Lemma 2), and all links in H' are reliable according to $\mathcal{F}'$ (Lemma 3).

□

Lemma 1. *At most t processes crash in $\mathcal{F}'$.*

Proof. Follows immediately from (c). □

Lemma 2. *H' is a history of $A(\Pi)$ using $\mathcal{D}$.*

Proof. All events and states are from $A(\Pi)$, because all additional events and states have been removed. If algorithm A makes use of a failure detector $\mathcal{D}$, then $trans(\mathcal{D})(\mathcal{F}) = \mathcal{D}(\mathcal{F}')$ (Since $\forall t : \; cc(\mathcal{F}', t) = cc{+}c(\mathcal{F}, t)$). □

Lemma 3. *All links in H' are reliable according to $\mathcal{F}'$.*

Proof. We have to show the three properties of reliable links, namely: No Creation (Lemma 5), No Duplication (Lemma 6), and No Loss (Lemma 7). □

To prove lemma 5, we first need to show the auxiliary lemma 4:

Lemma 4. *Let t_s be the time a send event from $A(p_i)$ to $A(p_j)$ in H_2 occurs, t_r be the time of the corresponding receive event in H_2, and $t_j := t_{not(cc+c,\mathcal{F},p_j)}$ and $t_i := t_{not(cc+c,\mathcal{F},p_i)}$. Then:*

$$t_s \geq t_i \Rightarrow t_r \geq t_j$$

Proof. The above lemma is equivalent to: $t_r < t_j$ implies $t_s < t_i$. At first, we observe that $t_s < t_r$ and $p_i \notin C(t_s)$ (because the receive event is executed at time t_s). Assume $t_r < t_j$. Since $A(p_j)$ receives the message, we can conclude:

$$t_{not(reachable,\mathcal{F},p_j,p_i)} \quad > \quad t_r > t_s \tag{2}$$

Since the in H_2 removed 3wh-algorithm has only allowed to 3wh-deliver messages after having received a $[3, m]$ message (lines 9-10 in Figure 3), which is only sent from a process after having on his part received a $[2, m]$ message (lines 7-8), we are sure that after the 3wh-send event, $A(p_i)$ was able to receive the $[2, m]$ message from $A(p_j)$ and therefore:

$$t_{not(reachable,\mathcal{F},p_i,p_j)} \quad > \quad t_s \tag{3}$$

From the definition of connected follows:

$$\exists c \in \mathit{correct}(\mathcal{F}), t_{not(reachable,\mathcal{F},c,p_j)} \geq t_j > t_r > t_s \tag{4}$$
$$\exists c' \in \mathit{correct}(\mathcal{F}), t_{not(reachable,\mathcal{F},p_j,c')} \geq t_j > t_r > t_s \tag{5}$$

If we put all paths together, we have:

$$\text{with (2) and (4)} \quad : \quad \exists c \in \mathit{correct}(\mathcal{F}), t_{not(reachable,\mathcal{F},c,p_i)} > t_s \tag{6}$$
$$\text{with (3) and (5)} \quad : \quad \exists c' \in \mathit{correct}(\mathcal{F}), t_{not(reachable,\mathcal{F},p_i,c')} > t_s \tag{7}$$

Equation (6) and (7) imply $t_{not(connected,\mathcal{F},p_i)} > t_s$ and together with $p_i \notin C(t_s)$, we conclude that $t_i > t_s$. □

Lemma 5. *(No Creation in H'.) For all messages m, if p_j receives m from p_i in H', then p_i sends m to p_j in H'.*

Proof. We know, that there is no creation in H. In our construction, send events of the same layer can only decrease in the local history of crashed processes in step (c) (after the time of their crash). But since Lemma 4 shows that messages that are sent from a process that is already disconnected in $\mathcal{F}$ (and therefore crashed in $\mathcal{F}'$) can only be received by processes that are already disconnected too, the corresponding receive events also get lost in H'. □

Lemma 6. *(No Duplication in H'.) For all messages m: p_j receives m from p_i at most once.*

Proof. In the 3wh-algorithm, no message is delivered more than once and in the relay-algorithm, every message received is remembered in a variable $delivered_i$ (lines 11-13 in Figure 4). □

Lemma 7. *(No Loss in H' according to $\mathcal{F}'$.) For all messages m, if p_i sends m to p_j and p_j executes receive actions infinitely often, then p_j receives m from p_i.*

Proof. In the removed relaying algorithm, after every relay-send event, the message m is relayed by $A(p_i)$ to all other processes (lines 6-7 in Figure 4). If a cc+c process (in $\mathcal{F}$) receives such a relayed message, it checks in lines 11-12 whether it is the recipient and has

not yet delivered it (and relay-delivers m in this case). Otherwise, it propagates m further to all other processes (lines 14-16).

Since p_i is at the time of the in step (a) in H_1 inserted send-event out-connected in $\mathcal{F}$ (otherwise, p_i would have already crashed in $\mathcal{F}'$), there is a path of directly-reachable cc+c processes to a (totally) correct process in $\mathcal{F}$. A correct process will receive m and relay it (possibly indirectly) to $A(p_j)$, since p_j is in-connected in $\mathcal{F}$ (because it takes infinitely many steps in $(H', \mathcal{F}')$). □

The following lemma is used for the proof of proposition 2.

Lemma 8. *Let $(H'_1, \mathcal{F}'_1)$ be in trans$((H_1, \mathcal{F}_1))$ and $(H'_2, \mathcal{F}'_2)$ be in trans$((H_2, \mathcal{F}_2))$. Then, for all $p_i \in \Pi$:*

$$H'_1[i]/_{t_{not(cc+c,\mathcal{F}'_1,p_i)}} = H'_2[i]/_{t_{not(cc+c,\mathcal{F}'_2,p_i)}} \quad \Rightarrow \quad H_1[i]/_{t_{not(cc,\mathcal{F}_1,p_i)}} = H_2[i]/_{t_{not(cc,\mathcal{F}_\in,p_i)}}$$

Proof. Follows from the definition of *trans* and the fact, that $t_{not(cc,\mathcal{F},p_i)} \leq t_{not(cc+c,\mathcal{F}_{tr},p_i)}$. □

Proposition 2. $\mathcal{S}_{c.s.} \subseteq \Sigma$

Proof. Assume $(H, \mathcal{F}) \in \mathcal{S}_{c.s.}$. We then build an new history H' from H and simulate all links according to the specification of the three-way-handshake and the relay algorithm such that $(H', \mathcal{F}) \in$ *trans*$((H, \mathcal{F}))$ and $(H', \mathcal{F}) \in \mathcal{S}_{g.o.} \subseteq$ *trans*(Σ) ($\mathcal{F} \in \mathcal{E}^t_{c.s.}$ implies that $\mathcal{F} \in \mathcal{E}^t_{g.o}$). This means, that there exists a $(H'', \mathcal{F}'') \in \Sigma$, with $(H', \mathcal{F}) \in$ *trans*$((H'', \mathcal{F}''))$. Together with Lemma 8 and the fact, that Σ is cc restricted, we can conclude that $(H, \mathcal{F}) \in \Sigma$.

□