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

Carole Delporte-Gallet, Hugues Fauconnier, Felix Freiling, Lucia Draque Penso, Andreas Tielmann

To cite this version:

Carole Delporte-Gallet, Hugues Fauconnier, Felix Freiling, Lucia Draque Penso, Andreas Tielmann. Transforming Asynchronous Systems with Crash-Stop Failures and Failure Detectors to the General Omission model. 2007. hal-00130766v2

HAL Id: hal-00130766
https://hal.archives-ouvertes.fr/hal-00130766v2
Submitted on 9 Mar 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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.
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 electronic 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].
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 \(T\) 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 \(H_A[i]/_{\ell}\) 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 $A(Π)$ is a vector of local histories $⟨H_A[1], H_A[2], \ldots, H_A[n]⟩$.

**Failures and Failure Patterns** A failure pattern $F$ is a function that maps each value $t$ from $T$ 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 $C : T \rightarrow 2^{Π}$ 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:

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

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 $F = (C, O_S, O_R)$.

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

$$\text{crashed}(F, t) := \{p \mid p \in C(t)\}$$

$$\text{crash-correct}(F, t) := \{p \mid p \notin C(t)\}$$

$$\text{send-omissive}(F, t) := \{p_s \mid \exists p_d : (p_s, p_d) \in O_S(t)\}$$

$$\text{receive-omissive}(F, t) := \{p_d \mid \exists p_s : (p_s, p_d) \in O_R(t)\}.$$ 

$$\text{omissive}(F, t) := \text{send-omissive}(F, t) \cup \text{receive-omissive}(F, t)$$

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:

\[
directly-\text{reachable}(\mathcal{F}, t) := \{(p, q) \mid (q, p) \notin O_S(t) \land (p, q) \notin O_R(t) \\
\quad \land q \in \text{crash-correct}(\mathcal{F}, t) \rightarrow p \in \text{crash-correct}(\mathcal{F}, t)\}
\]

\[
\text{reachable}(\mathcal{F}, t) := \{(p_d, p_s) \mid (p_d, p_s) \in \text{directly-reachable}(\mathcal{F}, t) \\
\quad \lor \exists r \in \Pi : ((p_d, r) \in \text{reachable}(\mathcal{F}, t) \land (r, p_s) \in \text{reachable}(\mathcal{F}, t))\}
\]

\[
in-\text{connected}(\mathcal{F}, t) := \{p_d \mid \exists c \in \text{correct}(\mathcal{F}) : (p_d, c) \in \text{reachable}(\mathcal{F}, t)\}
\]

\[
out-\text{connected}(\mathcal{F}, t) := \{p_s \mid \exists c \in \text{correct}(\mathcal{F}) : (c, p_s) \in \text{reachable}(\mathcal{F}, t)\}
\]

\[
\text{connected}(\mathcal{F}, t) := \text{in-\text{connected}(\mathcal{F}, t) \cap out-\text{connected}(\mathcal{F}, t)}
\]

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

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

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

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

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

We define an environment \(\mathcal{E}\) to be a set of possible failure patterns. \(\mathcal{E}_{c,s}\) denotes the
set of all failure patterns where only crash-stop faults occur and at most \(t\) processes crash.
\(\mathcal{E}_{g.o}\) 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, \(\mathcal{E}_{c,s} \subseteq \mathcal{E}_{g.o}\)).

### Failure Detectors

A failure detector provides (possibly incorrect) information about
the failure pattern that occurs in an execution [5]. A failure detector history \(\text{FDH}\) with
range \(\mathcal{R}\) is a function from \(\Pi \times \mathcal{T}\) to \(\mathcal{R}\). \(\text{FDH}(p, t)\) 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 \(\mathcal{R}_D\). \(\mathcal{D}(\mathcal{F})\) 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 \( p_j \), \((p_i, p_j) \notin O_S(F), (p_j, p_i) \notin O_R(F)\), 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 \( \Pi \) be a set of processes and \( A \) be an algorithm. We define \( \mathcal{H}(A(\Pi), E) \) to be the set of all tuples \((H_A, F)\) such that \( H_A \) is a history of \( A(\Pi) \) and \( F \in E \). A system \( S(A(\Pi), E) \) of \( A(\Pi) \) is a subset of \( \mathcal{H}(A(\Pi), E) \). A problem specification \( \Sigma \) is a set of tuples of histories and failure patterns and a system \( S \) satisfies a problem specification \( \Sigma \), if \( S \subseteq \Sigma \). Take Consensus as an example: It is specified by making statements about some variables \textit{propose} and \textit{decide} in the states of a history (e.g., the value of \textit{decide} has eventually to be equal at all (crash-)correct processes).

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

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 \((t_{\text{not}(cc+c,F,p_i)}\) 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:

\[(H', F') \in \text{trans}((H, F)) \iff \forall p_i \in \Pi : H[i] / t_{\text{not}(\text{cc}, F, p_i)} \leq_{sa} H'[i] / t_{\text{not}(\text{cc+c}, F', p_i)}\]

This implies that for all \(p_i\):

\[t_{\text{not}(\text{cc}, F, p_i)} \leq t_{\text{not}(\text{cc+c}, F', p_i)}\].

\[
\text{trans}(\Sigma) = \{(H', F') \mid (H', F') \in \text{trans}((H, F)) \land (H, F) \in \Sigma\}
\]

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.

<table>
<thead>
<tr>
<th>Consensus</th>
<th>(\text{trans}(\text{Consensus}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validity: The decided value of every cc process must have been proposed.</td>
<td>The decided value of every cc+c process must have been proposed.</td>
</tr>
<tr>
<td>Agreement: No two cc processes decide differently.</td>
<td>No two cc+c processes decide differently.</td>
</tr>
<tr>
<td>Termination: Every cc process eventually decides.</td>
<td>Every cc+c process eventually decides.</td>
</tr>
</tbody>
</table>

Table 1: Transformation of non-uniform Consensus

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

\[
\text{trans}(D)(F) := \bigcup_{F'} \{D(F') \mid \forall t : \text{cc}(F', t) = \text{cc+c}(F, t)\}
\]

If we take an \(\Omega\) 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 \(A' = \text{trans}(A)\) in two steps:

8
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* 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 $D(p_i)$, we translate it to a query on $\text{trans}(D)(p_i)$.

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

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 \( \Sigma \) is cc-restricted. Then, if and only if there is an algorithm \( A \) that satisfies \( \Sigma \) using a failure detector \( D \) in an environment with at most \( t \) crash-stop failures \((0 \leq t \leq n)\) and no omission failures, then \( \text{trans}(A) \) satisfies \( \text{trans}(\Sigma) \) using \( \text{trans}(D) \) 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 \( S^t_{c.s.}(A) := \mathcal{H}(A(\Pi), E^t_{c.s.}) \) and \( S^t_{g.o.}(A') := \mathcal{H}(A'(\Pi), E^t_{g.o.}) \).

Theorem 1. Let \( \Sigma \) be a cc-restricted problem specification. Then, if \( A \) is an algorithm using a failure detector \( D \) and \( A' = \text{trans}(A) \) is the transformation of \( A \) using \( \text{trans}(D) \), it holds that:
\[
\forall t \text{ with } 0 \leq t \leq n : S^t_{c.s.}(A) \subseteq \Sigma \iff S^t_{g.o.}(A') \subseteq \text{trans}(\Sigma)
\]

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 \( S_{c.s.} := S^t_{c.s.}(A) \) and \( S_{g.o.} := S^t_{g.o.}(A') \) and assume that \( A' = \text{trans}(A) \).

\( \Rightarrow \): Assume that \( S_{c.s.} \subseteq \Sigma \). By constructing for a given \( (H, F) \) in \( S_{g.o.} \), a tuple \( (H', F') \) in \( S_{c.s.} \), with \( (H, F) \in \text{trans}((H', F')) \), we can show that \( S_{g.o.} \subseteq \text{trans}(S_{c.s.}) \) (Proposition 1 in the appendix). In this construction, we remove the added communication layers from \( H \) and use the properties of our two send-primitives 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 \( \text{trans} \), that \( \text{trans}(S_{c.s.}) \subseteq \text{trans}(\Sigma) \), we can conclude that \( S_{g.o.} \subseteq \text{trans}(\Sigma) \).
Algorithm Relay

1: upon event \( \langle \text{init} \rangle \) do
2: \( \text{relayed}_i := \emptyset \);
3: \( \text{delivered}_i := \emptyset \);
4:
5: upon event \( \langle \text{relay-send}(p_i, m, p_j) \rangle \) do
6: for \( k := 1 \) to \( n \) do
7: \( \text{trigger} \langle \text{send}(p_i, [m, p_j], p_k) \rangle \);
8: \( \text{relayed}_i := \text{relayed}_i \cup \{ [m, p_j] \} \);
9:
10: upon event \( \langle \text{deliver}(p_j, [m, p_k], p_i) \rangle \) do
11: if \( (k = i) \) and \( (m \not\in \text{delivered}_i) \) then
12: \( \text{trigger} \langle \text{relay-deliver}(\text{sender}(m), m, p_i) \rangle \);
13: \( \text{delivered}_i := \text{delivered}_i \cup \{ m \} \);
14: elseif \( (k \neq i) \) and \( ([m, p_k] \not\in \text{relayed}_i) \) then
15: for \( l := 1 \) to \( n \) do
16: \( \text{trigger} \langle \text{send}(p_i, [m, p_k], p_l) \rangle \);
17: \( \text{relayed}_i := \text{relayed}_i \cup \{ [m, p_k] \} \);

Figure 4: The Relaying Algorithm for Process \( p_i \).

\[ \leftrightharpoons \]: Assume that \( S_{g.o.} \subseteq \text{trans}(\Sigma) \). We construct a history \( H' \) for all histories \( H \) in \( S_{c.s.} \), such that \( H' \) is in \( S_{g.o.} \subseteq \text{trans}(\Sigma) \). Together with the fact, that \( \Sigma \) is cc restricted, we can use this to prove that \( S_{c.s.} \subseteq \Sigma \) (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 straightforwardly that our transformations preserve the weakest failure detector property at least according to the class of transformed failure detectors.

Theorem 2. If \( D \) is the weakest failure detector for \( \Sigma \), then \( \text{trans}(D) \) is the weakest transformed failure detector for \( \text{trans}(\Sigma) \).
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


A Formal Proof

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

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

Proposition 1. $S_{g.o.} \subseteq \text{trans}(S_{c.s.})$

Proof. The proposition is equivalent to

$\forall (H, F) \in S_{g.o.} \Rightarrow (H, F) \in \text{trans}(S_{c.s.})$

From the definition of $\text{trans}$ follows:

$\forall (H, F) \in S_{g.o.} \Rightarrow \exists (H', F') \in S_{c.s.} : \forall p_i \in \Pi : H'[i]/t_{not(\text{cc},F',p_i)} \leq sa H[i]/t_{not(\text{cc}+c,F,p_i)}(1)$

We will in the following construct a new history $H'$ and a failure pattern $F'$ from $H$ and $F$ 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 a 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 $F'$, such that $\forall t : cc(F', t) = cc+c(F, t) \land \text{omissive}(F') = \emptyset$.

To build $H'$ from $H_2$, we substitute every query on a failure detector $\text{trans}(D)$ in $H_2$ with a query on $D$ in $H'$ and remove all states and events for every process $p_i$ that occur after time $t_{not(\text{cc},F',p_i)}$. 

14
The schedule of the construction is illustrated in Figure 5. From the construction of $H'$ and $F'$ it is clear, that $\forall p_i \in \Pi : \frac{H'[i]}{t_{not}(cc,F',p_i)} \leq \frac{H[i]}{t_{not}(cc+c,F,p_i)}$. It remains to show, that $(H',F') \in S_{c.s.}$. This means, that at most $t$ processes crash in $F'$ (Lemma 1), $H'$ is a history of $A(\Pi)$ using $D$ (Lemma 2), and all links in $H'$ are reliable according to $F'$ (Lemma 3).

**Lemma 1.** At most $t$ processes crash in $F'$.

**Proof.** Follows immediately from (c).

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

**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 $D$, then $\text{trans}(D)(F) = D(F')$ (Since $\forall t : cc(F',t) = cc+c(F,t)$).

**Lemma 3.** All links in $H'$ are reliable according to $F'$.

**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 $t_j := t_{not}(cc+c,F,p_j)$ and $t_i := t_{not}(cc+c,F,p_i)$. 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:

$$t_{not}(reachable,F,p_j,p_i) \Rightarrow t_r > t_s$$

Figure 5: Construction of $H'$
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:

$$t_{not(reachable,F,p_i,p_j)} > t_s$$

From the definition of connected follows:

$$\exists c \in correct(F), t_{not(reachable,F,c,p_i)} \geq t_j > t_r > t_s$$

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

If we put all paths together, we have:

with (2) and (4) : $$\exists c \in correct(F), t_{not(reachable,F,c,p_i)} > t_s$$

with (3) and (5) : $$\exists c' \in correct(F), t_{not(reachable,F,p_j,c')} > t_s$$

Equation (6) and (7) imply $t_{not(connected,F,p_i)} > t_s$ and together with $p_i \not\in C(t_s)$, 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 $F$ (and therefore crashed in $F'$) 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).

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

\[Proof.\] 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 $F$) 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 \( p_i \in \Pi \):

\[
H'_1[i]/t_{not(cc+c, F'_1, pi)} = H'_2[i]/t_{not(cc+c, F'_2, pi)} \Rightarrow H_1[i]/t_{not(cc, F_1, pi)} = H_2[i]/t_{not(cc, F, pi)}
\]

**Proof.** Follows from the definition of trans and the fact, that \( t_{not(cc, F, pi)} \leq t_{not(cc+c, F_tr, pi)} \).

**Proposition 2.** \( S_c.s. \subseteq \Sigma \)

**Proof.** Assume \( (H, \mathcal{F}) \in S_c.s. \). 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 \( (H', \mathcal{F}) \in S_g.o. \subseteq trans(\Sigma) \) (\( \mathcal{F} \in E_{c.s.}^t \) implies that \( \mathcal{F} \in E_{g.o.}^t \)). 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 \( \Sigma \) is cc restricted, we can conclude that \( (H, \mathcal{F}) \in \Sigma \).