Automatic Transformations from Crash-Stop to Permanent Omission

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

From Crash-Stop to Permanent Omission: Automatic Transformation and Weakest Failure Detectors

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, pl. Jussieu, 75251 Paris, Cedex 05, France
Case Postale 7014, Paris, France - Fax: +33-14-4276849
Phone: +33-14-4276845
{cd, hf, tielmann}@liafa.jussieu.fr

[2] University of Mannheim
Laboratory for Dependable Distributed Systems
A5, 6, 68159 Mannheim, Germany
Postal Box D-68131, Mannheim, Germany - Fax: +49-621-1813577
{freiling, lucia}@rumms.uni-mannheim.de

Abstract

This paper studies the impact of omission failures on asynchronous distributed systems with crash-stop failures. We provide two different transformations for algorithms, failure detectors, and problem specifications, one of which is weakest failure detector preserving.

We prove that our transformation of failure detector Ω [**?**] is the weakest failure detector for Consensus in environments with crash-stop and permanent omission failures and a majority of correct processes.

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 [**?**], [**?**], [**?**].

Keywords: Fault-Tolerance, Weakest Failure Detectors, Transformations, Asynchronous Systems, Crash-Stop, Permanent Omission

1 Introduction

Message omission failures, which have been introduced in [**?**] and been refined in [**?**] put the blame of a message loss to a specific process instead of an unreliable message channel. Beyond the theoretical interest, omission models are also interesting for practical problems like security problems: Assume that some kind of trusted smartcards are disposed on untrusted processors. Then, it is relatively

*Contact Author

easy to restrict the power of a malicious adversary to only be able to remove messages of the trusted smartcards or to stop the smart cards themselves. Omission models have lead to the development of reductions from security problems in the Byzantine failure model [?], such as electronic commerce and voting [?], fair-exchange [?], and secure multiparty computation [?] to well-known distributed problems in the general omission model, such as consensus [?], where both process crashes and message omissions may take place.

The message omission and crash failures are considered here in asynchronous systems. Due to classical impossibility results concerning problems as consensus [?] in asynchronous systems, following the failure detector approach [?], we augment the system with oracles that give information about failures.

The extension of failure detectors to more severe failure models than crash failures is not so clear [?], because in these models failures may depend on the scheduling and on the algorithm. As it is easy to transform the general omission model into a model with only permanent omissions using standard techniques like the piggybacking of messages, we consider only permanent omissions and crashes. In this model, precise and simple definitions for failure detectors can easily be deduced from the ones in the crash model.

To provide the permanent omission model with the benefits of a well-understood system model like the crash-stop model, we give automatic transformations for problem specifications, failure detectors, and algorithms such that algorithms designed to tolerate only crash-stop failures can be executed in permanent omission environments and use transformed failure detectors to solve transformed problems. Specifically, we give two transformations. At first, one that works in every environment, but that transforms uniform problems into problems with only limited uniformity, and second one that works only with a majority of correct processes, but transforms uniform crash-stop problems into their uniform permanent omission counterpart. An interesting point is the fact that the transformation of the specification gives for most of the classical problems the standard specification in the message omission and crash failure model. Then, for example, from an algorithmic solution A of the consensus problem with a failure detector $\mathcal{D}$ in the crash-stop model, we get automatically $A' = \mathit{trans}(A)$, an algorithmic solution of the consensus problem using $\mathcal{D}' = \mathit{trans}(\mathcal{D})$ in the message omission and crash failure model.

Moreover, our first transformation preserves also the "weaker than" relation [?] 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 failure detector for the transformed problem. We can use this to show that our transformation of failure detector Ω is the weakest failure detector for (uniform) Consensus in an environment with permanent omission failures and a majority of correct processes. It is interesting to note that this transformed version of Ω can be implemented in partially synchronous models using some weak timing assumptions [?].

The problem of automatically increasing the fault tolerance of problems in environments with crash-stop failures has been extensively studied before (e.g., [?], [?], [?], or [?]). The results of [?], [?], and [?] assume in contrast to ours synchronous systems and no failure detectors. In [?], several transformations from crash-stop to send omission, to general omission, and to Byzantine faults are proposed. In [?], round-based algorithms with broadcast primitives are transformed into crash-stop-, general omission-, and Byzantine-tolerant algorithms. Asynchronous systems are considered in [?], 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 [?] 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 [**?**] such that we can model omission failures, failure patterns, and failure detectors. Another definition for a system model with crash-recovery failures, omission failures, and failure detectors is given in [**?**].

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. In the same way, it is the first time that the problem of the weakest failure detector for the Consensus problem for message omission and crash failures is solved.

We organize this paper as follows. In Section **??**, we define our formal system model, in Section **??**, we define our general problem and algorithm transformations, in Section **??** we state our theorems, and finally, in Section **??**, we summarize and discuss our results. The detailed proofs can be found in the optional appendix.

2 Model

The asynchronous distributed system is assumed to consist of n distinct fully-connected processes $\Pi = \{p_1, \ldots, p_n\}$. 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 system model used here is derived from that in [**?**]. 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.

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* $C : \mathcal{T} \to 2^{\Pi}$ 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. 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. The permanent omissions are modeled via a send- and a receive-omission failure pattern: $O_S : \mathcal{T} \to 2^{\Pi\times\Pi}$ and $O_R : \mathcal{T} \to 2^{\Pi\times\Pi}$. If $(p_s, p_d) \in O_S(t)$, then process p_s has a permanent send-omission to process p_d after time t. If $(p_s, p_d) \in O_R(t)$, then process p_d has a permanent receive-omission to process p_s after time t. All the failure patterns defined so far can be put together to a single failure pattern $\mathcal{F} = (C, O_S, O_R)$.

With such a failure pattern, we define a process to be correct, if it experiences no failure at all. A process p is crash-correct ($p \in$ *cr.-correct*$(\mathcal{F})$) in $\mathcal{F}$, if it does not crash. An in-connected process is a process that is crash-correct and receives all messages from a correct process (possibly indirect) and an out-connected process is a process where a correct process receives all messages from it (also possibly indirect). If a process p is in-connected and out-connected in a failure pattern $\mathcal{F}$, then we say that p is *connected* in $\mathcal{F}$ ($p \in$ *connected*$(\mathcal{F})$). This implies, that *crash-correct*$(\mathcal{F}) \subseteq$ *connected*$(\mathcal{F}) \subseteq$ *correct*$(\mathcal{F})$. Furthermore, we say that $p \in$ *connected*$(\mathcal{F}, t)$, if p is connected in $\mathcal{F}$ at time t (and similar for *cr.-correct*$(\mathcal{F}, t)$). We specify the point in time when a process p_i becomes disconnected/crashed in a failure pattern $\mathcal{F}$ with $t_{not(connected,\mathcal{F},p_i)}$ and $t_{not(cr.-correct,\mathcal{F},p_i)}$ respectively. A more formal definition can be found in the appendix.

We say that a failure pattern $\mathcal{F}'$ is an omission equivalent extension of another failure pattern $\mathcal{F}$ ($\mathcal{F} \leq_{om} \mathcal{F}'$), if the set of crash-correct processes in $\mathcal{F}$ is at all times equal to the set of connected processes in $\mathcal{F}'$ and there are no omission failures in $\mathcal{F}$. 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 connected (clearly, $\mathcal{E}^t_{c.s.} \subseteq \mathcal{E}^t_{g.o.}$).

Failure Detectors A failure detector provides (possibly incorrect) information about a failure pattern [**?**]. 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 $\mathcal{R}$. $\mathcal{D}(\mathcal{F})$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern $\mathcal{F}$.

Histories A *local history of a local algorithm module* $A(p_i)$, denoted $H[i]$, 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 assume that there is a function that assings every event to a certain point in time and define $H[i]/_t$ to be the maximal prefix of $H[i]$ where all events have occurred before time t. A *history* H of $A(\Pi)$ is a vector of local histories $\langle H[1], H[2], \ldots, H[n] \rangle$.

Reliable Links A reliable link does not create, duplicate, or lose messages. Specifically, if there is no permanent omission between two processes and the recipient executes infinitely many receive actions, then it will eventually receive every message. 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 $(H, \mathcal{F})$ such that H is a history of $A(\Pi)$, $\mathcal{F} \in \mathcal{E}$, and H and $\mathcal{F}$ are compatible, that is crashed processes do not take any steps after the time of their crash, there are no receive-events after a permanent omission, etc.. 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 (see Table **??**): 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).

3 From Crash-Stop to Permanent Omission

We will give here two transformations: one general transformation for all environments, where we provide only restricted guarantees for disconnected processes, and one for environments where less than half of the processes may not be connected, where we are able to provide for all processes the same guarantees as for the crash-stop case.

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 some additional states caused by differences in these variables (in particular, $H \leq_{sa} H$ for all H). We say that a problem specification Σ is closed under stuttering and augmentation, if $(H, \mathcal{F}) \in \Sigma$ and $H \leq_{sa} H'$ implies that $(H', \mathcal{F})$ is also in Σ. Most problems satisfy this natural closure property (e.g., Consensus).

3.1 The General Transformation

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 we demand only something from processes as long as they are connected. More formally:

$$\begin{aligned} (H', \mathcal{F}') \in \mathit{trans}((H, \mathcal{F})) \quad &\Leftrightarrow \quad \forall p_i \in \Pi : \; H[i]/_{t_{not(cr.-correct, \mathcal{F}, p_i)}} \leq_{sa} H'[i]/_{t_{not(connected, \mathcal{F}', p_i)}} \\ \mathit{trans}(\Sigma) \quad &:= \quad \{(H', \mathcal{F}') \mid (H', \mathcal{F}') \in \mathit{trans}((H, \mathcal{F})) \wedge (H, \mathcal{F}) \in \Sigma\} \end{aligned}$$

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 connected processes, because only histories with the same states (disregarding the augmentation variables) are allowed in the transformation at this processes (see Table **??**). 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.

A transformation of uniform Consensus leads to a problem specification where the uniform agreement is only demanded for processes before their time of disconnection. This means that it is allowed that after a partitioning of the network, the processes in the different network partitions come to different decision values. Another transformation, in which uniform Consensus remains truly uniform is given in Section **??**.

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 $\mathcal{D}$:

$$\mathit{trans}(\mathcal{D})(\mathcal{F}) := \bigcup_{\mathcal{F}'} \{\mathcal{D}(\mathcal{F}') \mid \mathcal{F}' \leq_{om} \mathcal{F}\}$$

Consider failure detector Ω [**?**]. Ω outputs only failure detector histories that eventually provide the same crash-correct leader at all crash-correct processes. Then, *trans*(Ω) outputs these failure detector histories if and only if they provide a *connected* common leader at all *connected* 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. We transform a given algorithm A into another algorithm $A' = \mathit{trans}(A)$ in two steps:

- 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 **??**. 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 idea of the relay algorithm is to relay every message to all other processes, such that they relay it again and all connected processes can communicate with each other, despite the fact that they are not directly-reachable. It is similar to other algorithms in the literature like [**?**]. Its detailed description is in Figure **??**.

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 $\mathit{trans}(\mathcal{D})(p_i)$. The relaying layer overlays the network with the best possible communication graph and the 3wh-layer on top of it cuts the unidirectional edges from this graph.

Algorithm *3wh*

1: **procedure** *3wh-send*(m, p_j)
2: *relay-send*$([1, m], p_j)$;
3:
4: **procedure** *3wh-receive*(m)
5: *relay-receive*$([l, m'])$;
6: **if** $(l = 1)$ **then** *relay-send*$([2, m'], \mathit{sender}([l, m']))$; $m := \perp$;
7: **elseif** $(l = 2)$ **then** *relay-send*$([3, m'], \mathit{sender}([l, m']))$; $m := \perp$;
8: **elseif** $(l = 3)$ **then** $m := m'$;
9: **elseif** $[l, m'] = \perp$ **then** $m := \perp$;

Figure 1: The Three Way Handshake Algorithm for Process p_i.

Algorithm *Relay*

1: **procedure** *init*
2: $relayed_i := \emptyset$; $delivered_i := \emptyset$;
3:
4: **procedure** *relay-send*(m, p_j)
5: **for** $k := 1$ **to** n **do**
6: *send*$([m, p_j], p_k)$;
7: $relayed_i := relayed_i \cup \{[m, p_j]\}$;
8:
9: **procedure** *relay-receive*(m)
10: *receive*$([m', p_k])$;
11: **if** $([m', p_k] = \perp)$ **then** $m := \perp$;
12: **elseif** $(k = i)$ **and** $(m' \notin delivered_i)$ **then**
13: $m := m'$; $delivered_i := delivered_i \cup \{m'\}$;
14: **elseif** $(k \neq i)$ **and** $([m', p_k] \notin relayed_i)$ **then**
15: **for** $l := 1$ **to** n **do**
16: *send*$([m', p_k], p_l)$;
17: $relayed_i := relayed_i \cup \{[m', p_k]\}$; $m := \perp$;

Figure 2: The Relaying Algorithm for Process p_i.

3.2 The Transformation for $n > 2t$

If only less than a majority of the processes are disconnected ($n > 2t$), then we only need to adapt the problem specification to the failure patterns of the new environment. We indicate this adaption of a problem specification with the index *g.o.* and specify it in the following way: $\Sigma_{g.o.} := \{(H, \mathcal{F}) \mid \exists (H, \mathcal{F}') \in \Sigma \wedge \mathcal{F}' \leq_{om} \mathcal{F}\}$. If we adapt Consensus to omission failures, then we get $\text{Consensus}_{g.o.}$ as in Table **??**.

The failure detector specifications can be transformed as in Section **??**. The algorithm transformation $trans_2$ works similar as in the previous section, but we add an additional *two-way-handshake (2wh) layer* between the relaying layer and the 3wh layer. The algorithm is described in Figure **??** and is similar to an algorithm in [**?**]. The idea of the algorithm is to broadcast every message to all other processes and to block until $t + 1$ processes have acknowledged the message. In this way, disconnected processes block forever (since they receive less than $t+1$ acknowledgements) and connected processes can continue.

4 Results

The proofs of the following theorems can be found in the appendix. In our first theorem, we show that a transformed algorithm solves a transformed problem with a transformed failure detector if it solves the untransformed problem. 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.

	Consensus	*trans*(Consenus)	$\text{Consensus}_{g.o.}$
Validity:	The decided value of every *cr.-correct* process must have been proposed.	The decided value of every *connected* process must have been proposed.	The decided value of every *connected* process must have been proposed.
Non-Uniform Agreement:	No two *cr.-correct* processes decide differently.	No two *connected* processes decide differently.	No two *connected* processes decide differently.
Uniform Agreement:	No two processes decide differently.	No two processes decide differently *before their disconnection.*	No two processes decide differently.
Termination:	Every *cr.-correct* process eventually decides.	Every *connected* process eventually decides.	Every *connected* process eventually decides.

Table 1: Transformations of the Consensus Problem

Theorem 1. *Let Σ be a problem specification closed under stuttering and augmentation. 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:*

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

Our second theorem shows, that with a majority of connected processes ($n > 2t$), $trans_2$ can be used to solve the adaption of a problem to the general omission model.

Theorem 2. *If A is an algorithm using a failure detector $\mathcal{D}$ and $A' = trans_2(A)$ is the transformation of A using $trans_2(\mathcal{D})$ and Σ is closed under stuttering and augmentation, then it holds that:*

$$\forall t \text{ with } t < n/2 : \ (\mathcal{H}(A(\Pi), \mathcal{E}^t_{c.s.}) \subseteq \Sigma \ \Rightarrow \ (\mathcal{H}(A'(\Pi), \mathcal{E}^t_{g.o.}) \subseteq \Sigma_{g.o.}$$

Weakest Failure Detectors A failure detector [**?**] 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. We show in the following theorem, that *trans* preserves the weakest failure detector property for non-uniform[1] failure detectors.

Theorem 3. *For all $t \in \mathcal{T}$: If a non-uniform failure detector $\mathcal{D}$ is a weakest failure detector for Σ in $\mathcal{E}^t_{c.s.}$ and Σ is closed under stuttering and augmentation, then $trans(\mathcal{D})$ is a weakest failure detector for $trans(\Sigma)$ in $\mathcal{E}^t_{g.o.}$.*

With Theorem **??**, **??**, and **??** we are able to show, the following theorem:

Theorem 4. *$trans(\Omega)$ is a weakest failure detector for uniform $\text{Consensus}_{g.o.}$ with a majority of correct processes.*

[1] A non-uniform failure detector outputs always the same set of histories for two failure patterns $\mathcal{F}$ and $\mathcal{F}'$ in which $correct(\mathcal{F}) = correct(\mathcal{F}')$.

```
Algorithm 2wh
1:  procedure init
2:      received_i := ∅; Ack_i := 0;
3:
4:  procedure 2wh-send(m, p_j)
5:      relay-send([m, p_j, ONE], p_k) to all other p_k; Ack_i := 1;
6:      while (Ack_i ≤ t) do
7:          relay-receive([m', p_k, num]));
8:          if (num = TWO) and (m' = m) and (k = j) then inc(Ack_i);
9:          elseif (num = ONE) then add [m', p_k, num] to received_i;
10:
11: procedure 2wh-receive(m))
12:     m := ⊥; relay-receive(m');
13:     if (m' ≠ ⊥) then add m' to received_i;
14:     if ([m'', p_k, ONE] ∈ received_i) for any m'', p_k then
15:         relay-send([m'', p_k, TWO], sender([m'', p_k, ONE]));
16:         if (k = i) then m := m'';
```

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

Proof. Since we know, that Ω is a weakest failure detector for non-uniform Consensus [**?**] and Ω is clearly non-uniform, together with Theorem **??**, *trans*(Ω) is a weakest failure detector for non-uniform *trans*(Consensus). Since non-uniform *trans*(Consensus) is strictly weaker than uniform Consensus$_{g.o.}$, *trans*(Ω) is especially necessary for uniform Consensus$_{g.o.}$. To show that *trans*(Ω) is sufficient for uniform Consensus$_{g.o.}$, we can simply use Theorem **??**, since we know that Ω is sufficient for uniform Consensus with a majority of correct processes. □

5 Conclusion

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. Furthermore, we have shown that *trans*(Ω) is the weakest failure detector for Consensus in an environment with permanent omission failures where less than half of the processes may crash. Additionally, we have proven that our transformation preserves the weakest failure detector property for all non-uniform failure detectors. As an open problem, we think that it would be interesting to replace the requirement of a correct majority in our second transformation with a failure detector Σ [**?**] that will also be sufficient. Apart from that, it may be possible to give more specific transformations that are less general, but also less communication expensive than our transformation.

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] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing, 2004.

[10] 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.

[11] Assia Doudou, Benoît Garbinato, Rachid Guerraoui, and André Schiper. Muteness failure detectors: Specification and implementation. In Jan Hlavicka, Erik Maehle, and András Pataricza, editors, *EDCC*, volume 1667 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 1999.

[12] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

A Model Details

We describe here more in detail some notions we only skeched in the previous part of the paper. At first, we define some predicates processes might fulfill depending on the failure pattern and the time t.

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

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:

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

Note that every connected process is necessarily crash-correct, since it is reachable from a correct process. We define for every predicate φ:

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

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 $t_{not(\varphi,\mathcal{F},p,q,\ldots)} = \infty$.

B Formal Proofs

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

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

Proof. We divide up the proof into two parts. Let $\mathcal{S}_{c.s.} := (\mathcal{H}(A(\Pi), \mathcal{E}^t_{c.s.})$ and $\mathcal{S}_{g.o.} := (\mathcal{H}(A'(\Pi), \mathcal{E}^t_{g.o.})$ and assume that $A' = \mathit{trans}(A)$.

"⇒": 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 $(H, \mathcal{F}) \in \mathit{trans}((H', \mathcal{F}'))$, we can show that $\mathcal{S}_{g.o.} \subseteq \mathit{trans}(\mathcal{S}_{c.s.})$ (Proposition **??**). 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 $\mathit{trans}(\mathcal{S}_{c.s.}) \subseteq \mathit{trans}(\Sigma)$, we can conclude that $\mathcal{S}_{g.o.} \subseteq \mathit{trans}(\Sigma)$.

"⇐": Assume that $\mathcal{S}_{g.o.} \subseteq \mathit{trans}(\Sigma)$. We construct $(H', \mathcal{F}')$ for all $(H, \mathcal{F})$ in $\mathcal{S}_{c.s.}$, such that $(H', \mathcal{F}')$ is in $\mathcal{S}_{g.o.} \subseteq \mathit{trans}(\Sigma)$. We can use this to prove that $\mathcal{S}_{c.s.} \subseteq \Sigma$ (Proposition **??**).

□

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:

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

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

(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 $\mathcal{F}' \leq_{om} \mathcal{F}$. To build H' from H_2, we substitute every query on a failure detector $\mathit{trans}(\mathcal{D})$ 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 $t_{not(cr.-correct, \mathcal{F}', p_i)}$.

The schedule of the construction is illustrated in Figure **??**. From the construction of H' and $\mathcal{F}'$ it is clear, that $\forall p_i \in \Pi : \; H'[i]/_{t_{not(cr.-correct, \mathcal{F}', p_i)}} \leq_{sa} H[i]/_{t_{not(connected, \mathcal{F}, p_i)}}$. 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 **??**), H' is a history of $A(\Pi)$ using $\mathcal{D}$ (Lemma **??**), and all links in H' are reliable according to $\mathcal{F}'$ (Lemma **??**).

□

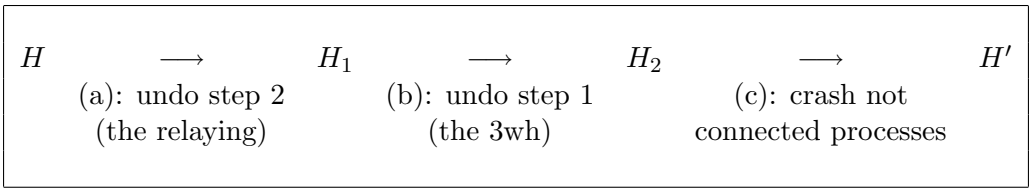

Figure 4: Construction of H'

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 $\mathcal{F}' \leq_{om} \mathcal{F}$). □

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 **??**), No Duplication (Lemma **??**), and No Loss (Lemma **??**). □

To prove lemma **??**, we first need to show the auxiliary lemma **??**:

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 $t_j := t_{not(connnected,\mathcal{F},p_j)}$ and $t_i := t_{not(connected,\mathcal{F},p_i)}$. 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 $t_s < t_r$. 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 (line 8 in Figure **??**), which is only sent from a process after having on his part received a $[2, m]$ message (line 7), 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:

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

If we put all paths together, we have:

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

Equation (**??**) and (**??**) imply $t_{not(connected,\mathcal{F},p_i)} = t_i > t_s$. □

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 **??** 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 12-13 in Figure **??**). □

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 5-6 in Figure **??**). If a connected process (in $\mathcal{F}$) receives such a relayed message, it checks in lines 12-13 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 connected 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 $(H', \mathcal{F}')$). □

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

Proof. Assume $(H, \mathcal{F}) \in \mathcal{S}_{c.s.}$. We then build a 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 $(H', \mathcal{F}) \in \mathcal{S}_{g.o.} \subseteq trans(\Sigma)$ ($\mathcal{F} \in \mathcal{E}^t_{c.s.}$ implies that $\mathcal{F} \in \mathcal{E}^t_{g.o}$). This means, that there exists a $(H'', \mathcal{F}'') \in \Sigma$, with $(H', \mathcal{F}) \in trans((H'', \mathcal{F}''))$.

Since in both, $\mathcal{F}''$ and $\mathcal{F}$ occur only crash failures, $\mathcal{F}'' = \mathcal{F}$ and therefore for all p_i, $H''[i] \leq_{sa} H'[i]$. Together with the fact that Σ is closed under stuttering and augmentation, we can conclude that $(H', \mathcal{F}) \in \Sigma$. H' and H differ only in the augmentation variables that are not relevant for the fulfillment of $trans(\Sigma)$, therefore: $(H, \mathcal{F}) \in \Sigma$.

□

Theorem 2. *If A is an algorithm using a failure detector $\mathcal{D}$ and $A' = trans(A)$ is the transformation of A using $trans(\mathcal{D})$ and Σ is closed under stuttering and augmentation, then it holds that:*

$$\forall t \text{ with } t < n/2 : \ (\mathcal{H}(A(\Pi), \mathcal{E}^t_{c.s.}) \subseteq \Sigma \ \Rightarrow \ (\mathcal{H}(A'(\Pi), \mathcal{E}^t_{g.o.}) \subseteq \Sigma_{g.o.}$$

Proof. Let $\mathcal{S}_{c.s.} := (\mathcal{H}(A(\Pi), \mathcal{E}^t_{c.s.})$ and $\mathcal{S}_{g.o.} := (\mathcal{H}(A'(\Pi), \mathcal{E}^t_{g.o.})$ and assume that $A' = trans(A)$. It is sufficient to show, that

$$\forall (H, \mathcal{F}) \in \mathcal{S}_{g.o.}, \exists (H', \mathcal{F}') \in \mathcal{S}_{c.s.} : \ (H' \leq_{sa} H) \wedge (\mathcal{F}' \leq_{om} \mathcal{F}) \tag{8}$$

To show this, we construct $(H', \mathcal{F}') \in \mathcal{S}_{c.s.}$ for a given $(H, \mathcal{F}) \in \mathcal{S}_{g.o.}$ in the following way: We first remove the variables, events, and states of the relay-algorithm, then remove the same for the 2wh-algorithm, and then remove the 3wh-algorithm to get H'. $\mathcal{F}'$ is a failure pattern, such that $\mathcal{F}' \leq_{om} \mathcal{F}$. We need to show, that $(H', \mathcal{F}')$ fulfills the properties of equation **??**. From the construction it is clear, that $H' \leq_{sa} H$ and $\mathcal{F}' \leq_{om} \mathcal{F}$. 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 **??**), H' is a history of $A(\Pi)$ using $\mathcal{D}$ (Lemma **??**), all links are reliable in $(H', \mathcal{F}')$ (Lemma **??**), and H' and $\mathcal{F}'$ are compatible (Lemma **??**). □

Lemma 8. *At most t processes crash in $\mathcal{F}'$.*

Proof. Follows immediately from $\mathcal{F}' \leq_{om} \mathcal{F}$. □

Lemma 9. *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 $\mathcal{F}' \leq_{om} \mathcal{F}$). □

Lemma 10. *Connected processes take infinitely many steps.*

Proof. The only possibility for a process to block is in line 6 of the 2wh-algorithm in Figure **??**. Since $n > 2t$, even after the disconnection of all t possibly faulty processes, every connected process receives acknowledgements from $n - t > t$ connected processes and therefore never blocks in line 6. □

Lemma 11. *Every process 2wh-sends at most one message after its disconnection.*

Proof. If p_i is disconnected after some time, it either does not receive messages from connected processes or connected processes do not receive messages from it. If it does not receive messages from connected processes, then after a 2wh-send event, it receives at most t acknowledgements (from the disconnected ones) and therefore waits forever in line 6 of the 2wh-algorithm in Figure **??**. If the connected processes do not receive messages from it and p_i 2wh-sends a message, also at most t processes will receive the ONE-message and answer with a TWO-message. Therefore, process p_i will block forever in line 6. □

Lemma 12. *The state of a process in H' does not change after its disconnection.*

Proof. With the 3wh-algorithm, we can ensure that a process does not receive messages from connected processes (Lemma **??**). With Lemma **??**, no process sends more than one message after its disconnection (and this message is not sufficient for a 3wh). Therefore, this send event is not visible to other processes and the internal state of a disconnected process cannot be influenced after its disconnection. □

Lemma 13. *H' and $\mathcal{F}'$ are compatible.*

Proof. We show, that every connected process takes infinitely many steps (Lemma **??**), and that the state of a process after its disconnection does not change anymore in H' (Lemma **??**). □

Lemma 14. *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 **??**), No Duplication (Lemma **??**), and No Loss (Lemma **??**). □

Lemma 15. *No Creation in H'.*

Proof. There is no loss in H and the send events in the same layer never decrease. □

Lemma 16. *No Duplication in H'.*

Proof. In the relay- and the 3wh-handshake algorithm, there is no duplication (Lemma **??**). In the 2wh-algorithm, only one ONE message with the correct id is sent for every 2wh-send. □

Lemma 17. *No Loss in H'.*

Proof. We know from Lemma **??**, that there is no loss between connected processes without the 2wh-algorithm. With Lemma **??**, we know connected processes take infinitely many steps and make therefore infinitely many receive actions. It remains to show, that disconnected processes stop sending and receiving messages after their disconnection (Lemma **??**). □

Theorem 3. *For all $t \in \mathcal{T}$: If a non-uniform failure detector $\mathcal{D}$ is a weakest failure detector for Σ in $\mathcal{E}^t_{c.s.}$ and Σ is closed under stuttering and augmentation, then trans($\mathcal{D}$) is a weakest failure detector for trans(Σ) in $\mathcal{E}^t_{g.o.}$.*

Proof. If $\mathcal{D}$ is a weakest failure detector for Σ in $\mathcal{E}^t_{c.s.}$, then *trans*($\mathcal{D}$) is sufficient for *trans*(Σ) in $\mathcal{E}^t_{g.o.}$ (Theorem **??**). Assume a failure detector $\mathcal{D}'$ is sufficient for *trans*(Σ) in $\mathcal{E}^t_{g.o.}$. Then, $\mathcal{D}'$ is sufficient for *trans*(Σ) in $\mathcal{E}^t_{c.s.}$ (since $\mathcal{E}^t_{c.s.} \subseteq \mathcal{E}^t_{g.o.}$). Since $\mathcal{F} \in \mathcal{E}^t_{c.s.}$ plus $(H, \mathcal{F}) \in$ *trans*(Σ) implies that there is a $(H', \mathcal{F}') \in \Sigma$ such that for all p_i: $H'[i] \leq_{sa} H[i]$, we know that $(H, \mathcal{F})$ is also in Σ (from the definition of *trans*, $\mathcal{F} = \mathcal{F}'$, and Σ closed under stuttering and augmentation). Therefore, *trans*(Σ) is equal to Σ in $\mathcal{E}^t_{c.s.}$, $\mathcal{D}'$ is sufficient for Σ in $\mathcal{E}^t_{c.s.}$, and moreover, $\mathcal{D}'$ is reducible to $\mathcal{D}$ in $\mathcal{E}^t_{c.s.}$ (since $\mathcal{D}$ is a weakest failure detector for Σ in $\mathcal{E}^t_{c.s.}$). This means, that it is possible to emulate $\mathcal{D}$ using $\mathcal{D}'$ (i.e., a problem specification *Probl*($\mathcal{D}$) that is equivalent to $\mathcal{D}$). If the reduction algorithm is $T_{\mathcal{D}' \to \mathcal{D}}$, then *trans*($T_{\mathcal{D}' \to \mathcal{D}}$) using *trans*($\mathcal{D}'$) emulates *trans*($Probl(\mathcal{D})$) in $\mathcal{E}^t_{g.o.}$ (Theorem **??**) and since $\mathcal{D}$ is non-uniform, *trans*($Probl(\mathcal{D})$) is equivalent to *trans*($\mathcal{D}$). Therefore, $\mathcal{D}'$ is reducible to *trans*($\mathcal{D}$) in $\mathcal{E}^t_{g.o.}$. □