On the weakest information on failures to solve mutual exclusion and consensus in asynchronous crash-prone read/write systems

Carole Delporte-Gallet, Hugues Fauconnier, Michel Raynal

On the Weakest Information on Failures to Solve Mutual Exclusion and Consensus in Asynchronous Crash-prone Read/Write Systems

Carole Delporte-Gallet[†] Hugues Fauconnier[†]
Michel Raynal[◇,⋆]

[†]IRIF, Université de Paris, Paris, France
[◇]Univ Rennes, Inria, CNRS, IRISA, 35000 Rennes, France
[⋆]Department of Computing, Polytechnic University, Hong Kong

Abstract

Mutual exclusion and consensus are among the most important coordination problems encountered in asynchronous concurrent systems, whether processes communicate using read/write registers or message passing. Unfortunately, neither can be solved in crash-prone systems, as soon as even a single process may crash. Hence, an important question: *which is the weakest information on failures* that must be given to the processes so that these problems can be solved whatever the number of crashes. This approach to circumvent impossibility results is known under the name *failure detectors*.

Considering mutual exclusion and consensus in a crash-prone asynchronous system where the processes communicate through read/write registers, this article answers the previous question by presenting two failure detectors. The first, called Quasi-Perfect (QP) allows mutual exclusion to be solved in the presence of any number of process crashes. The second, called Ω^*, allows consensus to be solved in the general model where not all but an a priori unknown subset of processes participate in consensus. In addition to algorithms solving each of the previous problems with the help of the associated failure detector, the article shows that QP and Ω^* provides the weakest information on failures needed to solve mutex exclusion and participant-restricted consensus respectively.

Keywords: Asynchronous system, Concurrency, Consensus, Failure detector, Mutual exclusion, Participating process, Process crash, Read/write register, Weakest information on failures.

1 Introduction

Mutual exclusion and consensus. Mutual exclusion is the oldest (1965) and most famous synchronization problem [14]. It consists in ensuring that some parts of code (called critical sections) can be accessed by a single process at a time. It is the concurrency-related problem that received the most attention (e.g., see the historical book [4], and synchronization-oriented textbooks such as [29, 32]).

Consensus was introduced later (1980) in the context of synchronous systems prone to Byzantine failures [22, 26]. It consists in allowing the processes to agree on the same value. As a simple example, when, to cope with process crashes, one has to duplicate a state machine on a set of server processes with one copy per server, in order to prevent the copies from diverging the ones from the others, the servers have to agree on a single order to apply the state machine operations (or commands) on their individual copies [21, 29, 31]. In short, while resources are physical objects managed with mutual exclusion, state machines are digital objects managed with consensus [27].

Unfortunately mutual exclusion cannot be solved in asynchronous systems where processes can crash, be their communication medium a read/write shared memory or message-passing. This can be easily shown by a simple scenario in which a process crashes while it is inside the critical section. As the critical section is never released, due to asynchrony no alive process can know if the process inside the critical section is crashed or is only very slow. This uncertainty implies that no other process can enter the critical section, and consequently the system can block forever. A second bad news lies in the fact a similar impossibility result is also true for consensus. Namely, there is no deterministic algorithm solving consensus in asynchronous systems be the communication medium message passing [16] or a read/write shared memory [24], even if a single process may crash and the processes have to agree on a single bit.

From impossibility results to failure detectors. Several approaches have been proposed and investigated to circumvent the previous impossibilities. One of them, which is system-oriented, consists in enriching the system with failure-related objects providing each process individually with information on failures. This is the *failure detector*-based approach introduced in [6]. More precisely, a failure detector provides each process with one or several read-only local variables, containing information on failures. According to the type and the quality of this information, different failure detector classes can be defined. As a simple example, the class of *perfect* failure detectors (denoted P) provides each process p_i with a read-only set SUSPECTED_i that (a) never contains the identity of a process p_j before p_j crashes, and (b) eventually contains the identities of all the processes that crashed. It is easy to see that a perfect failure detector allows a process not to remain blocked forever because another process crashed.

A fundamental notion associated with failure detectors is the notion of *weakest failure detector* for a given problem. Intuitively, *weakest failure detector* means that, a failure detector D is the weakest failure detector to solve a problem $\mathcal{P}$, if (1) there is a D-based algorithm solving $\mathcal{P}$, and (2) any other failure detector that allows $\mathcal{P}$ to be solved, provides the non-crashed processes with enough information on failures that allows them to build D.

Content of the article. Considering systems in which the processes communicate through atomic read/write registers, and any number of processes may crash, this article is made up of two parts: one devoted to mutual exclusion (Sections 3-6), and one devoted to consensus (Sections 7-10). Both parts have the same structure (numbered 1, 2, 3, and 4 in the article).

1. First the definition of the problem: mutual exclusion in Section 3 and restricted-participant consensus in Section 7 in the previous crash model.
2. Then the statement of a failure detector that allows to solve the problem: QP for mutual exclusion in Section 4 and Ω^* for participant-restricted consensus in Section 8.
3. Then the presentation of a QP-based algorithm in Section 5 and of an Ω^*-based participant-restricted consensus algorithm in Section 9.
4. Finally Section 6 and Section 10 present proofs showing that each of the previous failure detectors is the weakest that allows the corresponding problem to be solved (which means that it captures the weakest information on failures needed to solve the problem).

These technical sections are preceded by Section 2 which presents the fault-prone computing model, and followed by Section 11 which concludes the article[1].

[1]This article unifies, extends, and makes simpler two conference articles, one devoted to mutual exclusion [12], the other one devoted to consensus [13].

2 Basic Computing Model: Read/Write, Asynchrony and Crash failures

Process and communication model. The system is made up of a set Π of n sequential asynchronous processes denoted p_1, ..., p_n. Without loss of generality we consider that the integer i is the identity of p_i. Asynchronous means that the processes progress independently the ones from the others.

The processes communicate by accessing a shared memory made up of atomic read/write registers. From a notational point of view, atomic read/write registers are denoted with uppercase letters. Differently, local variables of each process are denoted with lowercase letters.

From 0 to $(n-1)$ processes can commit crash failures. A process commits a crash when it halts prematurely. Before halting (if it ever halts), a process executes correctly its algorithm. After it crashed, a process does no longer access the shared memory. Given an execution, a process that crashes in this execution is said to be *faulty*. Otherwise, it is *correct*.

Failure detectors: notation and reduction. A failure detector is a device that provides each process p_i with read-only local variables giving (possibly incorrect) information on failures. Different failure detectors can be defined according to the quality on the information they deliver to the processes. As a simple example the *perfect* failure detector was defined in the introduction. A formal introduction to the concept of a failure detector can be found in [6].

A failure pattern is a function $\mathcal{F}$ that gives, for each time instant τ, the processes that crashed up to time τ. The set of correct processes in an execution is consequently $\mathcal{C} = \Pi \setminus \big(\cup_{\tau > 0} \mathcal{F}(\tau) \big)$. The time used in the definition of both a failure pattern $\mathcal{F}$ and a failure detector, is a conceptual time perceived by an omniscient external observer. This time remains always unknown to the processes.

A failure detector D is *weaker than* a failure detector D' (denoted $D \preceq D'$) if there is a reduction algorithm from D' to D, i.e, an algorithm based on D' whose outputs satisfy the properties of D. If D is weaker than D', any problem that can be solved with D can be solved with D'. If $D \preceq D'$ but $D' \not\preceq D$, D is *strictly weaker than* D' ($D \prec D'$).

3 Mutual Exclusion 1: Definition for the Crash Model

Mutual exclusion is the oldest (and one of the most important) concurrency-related problem. Formalized by Dijkstra in the mid-sixties [14], it consists in building what is called a lock (or mutex) object, defined by two operations, denoted $\mathsf{entry}()$ and $\mathsf{exit}()$. (Recent textbooks including mutual exclusion and variants of it are [28, 32].)

The invocation of these operations by a process p_i always follows the following pattern: "$\mathsf{entry}()$; *critical section*; $\mathsf{exit}()$", where "critical section" is any sequence of code. A process that is not in the critical section and has no pending $\mathsf{entry}()$ or $\mathsf{exit}()$ invocation, is said to be in the *remainder section*. A mutual exclusion object (also called a lock) is defined by the following properties.

- Mutual exclusion: No two processes are simultaneously in their critical section.
- Deadlock-freedom: If a correct process p_i has a pending $\mathsf{entry}()$ operation and no process is in the critical section, eventually some process p_j (possibly $p_j \neq p_i$) returns from its $\mathsf{entry}()$ operation.
- Wait-free exit: If a correct process invokes $\mathsf{exit}()$, it returns from its invocation.

In the failure model considered in this paper, it is possible that a process crashes while it is in the critical section. It is assumed that such a crash implicitly releases of the critical section. (Without such an assumption, the crash of a process inside a critical section would prevent any other process from entering the critical section.) Hence, from a conceptual point view, no crashed process is inside the critical section. It is the role of the failure detector to inform the other processes of it.

While some mutual exclusion algorithms consider the deadlock-freedom progress condition, other mutual exclusion algorithms consider the stronger starvation-freedom progress condition, namely, if a correct process invokes the operation entry() and does not crash, it eventually returns from its invocation (i.e., it enters the critical section).

4 Mutual Exclusion 2: The Quasi-Perfect Failure Detector

4.1 Definition.

The failure detector QP was introduced in [11] in the context of *fair synchronization*. More precisely, this article shows that QP is the weakest failure detector to solve the *fair synchronization* problem as defined in [33]. QP provides each process p_i with two sets of processes, denoted TRUSTED_i and CRASHED_i. Both these sets are initially empty, and are then subsets of Π. Let $\text{INIT}_i = \Pi \setminus (\text{TRUSTED}_i \cup \text{CRASHED}_i)$, hence initially $\text{INIT}_i = \Pi$.

Informal presentation. Intuitively, the aim of the set TRUSTED_i is to eventually contain the set of correct processes, while the aim of the set $\text{INIT}_i \cup \text{CRASHED}_i$ is to eventually contain the set of faulty processes.

The set INIT_i can only decrease, while the set CRASHED_i can only increase. Moreover, QP is such that, at any time, we have $\text{TRUSTED}_i \cap \text{CRASHED}_i = \emptyset$.

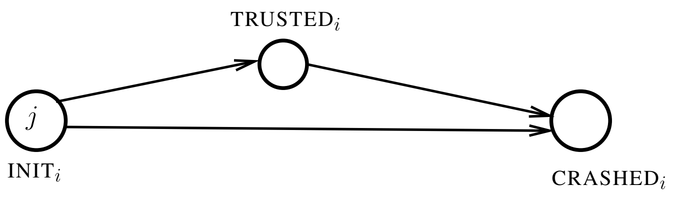

Figure 1: Possible moves of the process identity j in QP at process p_i

Let us consider a process p_i, while p_j is any process. As previously indicated, initially $p_j \in \text{INIT}_i$. If p_j is correct, or before it crashes (if it is faulty), p_j can move from INIT_i to TRUSTED_i. Moreover, if it is correct, p_j will go and remain forever in TRUSTED_i. If it crashes, p_j either remains forever in INIT_i, or goes directly from INIT_i to CRASHED_i, or goes from TRUSTED_i to CRASHED_i. Figure 1 describes this behavior (at each process p_i, initially INIT_i contains all process identities; then a process identity can move along the arrows.)

By convention, if a process p_i crashes at time τ, its failure detector outputs after time τ are both the set $\text{TRUSTED}_i(\tau)$ and the set $\text{CRASHED}_i(\tau)$.

Formal definition. Formally, the behavior of QP is defined by the following properties, for every process p_i. To simplify the reading of notations, we write $k \in \text{SET}_i$ instead of $p_k \in \text{SET}_i$, where SET_i is TRUSTED_i, CRASHED_i, $\mathcal{F}(\tau)$, or $\mathcal{C}$.

- $\forall\, i, \forall\, \tau$: $\text{TRUSTED}_i(\tau) \cap \text{CRASHED}_i(\tau) = \emptyset$.
- $\forall\, i$: $j \in \text{TRUSTED}_i(\tau) \Rightarrow$
  $\big(\forall\, \tau' \geq \tau\text{: } j \in \text{TRUSTED}_i(\tau') \cup \text{CRASHED}_i(\tau')\big) \wedge$
  $\big(\forall\, k \in \mathcal{C} :\ \exists \tau'\text{: } j \in \text{TRUSTED}_k(\tau') \cup \text{CRASHED}_k(\tau')\big)$.
  (A trusted process has to be eventually observed by all correct processes.)
- $j \in \text{CRASHED}_i(\tau) \Rightarrow j \in \mathcal{F}(\tau)$. ($\text{CRASHED}_i$ contains only crashed processes.)
- $j \in \text{CRASHED}_i(\tau) \Rightarrow \big(\forall\, \tau' \geq \tau\text{: } j \in \text{CRASHED}_i(\tau')\big)$. (crashes are stable.)

Moreover, if p_i is correct:

- $j \in \mathcal{F}(\tau) \Rightarrow \big(\exists \tau' \geq \tau\text{: } j \notin \text{TRUSTED}_i(\tau')\big)$. (Eventually, no faulty process $\in \text{TRUSTED}_i$.)

- $j \in \mathcal{C} \Rightarrow \big(\exists \tau\colon j \in \text{TRUSTED}_i(\tau)\big)$. (Eventually, every correct process $\in$ TRUSTED_i.)

It is easy to see that a correct process p_j never appears in the set CRASHED_i. As the perfect failure detector P, the failure detector QP is realistic [7], i.e., it can be implemented with appropriately defined underlying synchrony assumptions.

4.2 QP with respect to P and $\Diamond P$

Both the perfect failure detector P and the eventually perfect failure detector $\Diamond P$ were introduced in [6]. Each output at each process p_i is a set of processes, denoted SUSPECTED_i. The sets output by the perfect failure detector P satisfy the following properties: Strong Accuracy: no process belongs to SUSPECTED_i before it crashes, and Strong Completeness: eventually every process that crashes belongs forever to SUSPECTED_i.

The sets SUSPECTED_i output by $\Diamond P$ satisfy a weakened version of P, namely, its accuracy and completeness properties are required to hold eventually only. The power of a perfect failure detector is investigated in [19], and the following theorem is proved in [11].

Theorem 1. *$\Diamond P \prec QP \prec P$.*

Remark. Let us notice that P and QP are very close. Considering P, let $\text{OTHERS}_i = \Pi \setminus \text{SUSPECTED}_i$. Initially $\text{OTHERS}_i = \Pi$, and then this set can only decrease, while SUSPECTED_i can only increase. When considering QP, a local set INIT_i contains initially all the processes, and can then only decrease. But a faulty process that does not participate in the computation can remain forever in this set. This is a main difference between P and QP, which makes QP weaker than P. It follows that, from a practical point of view, despite not being the weakest, P is a good candidate to solve crash-prone mutual exclusion.

5 Mutual Exclusion 3: a QP-based Algorithm

5.1 Enriching Lamport's Bakery Mutual Exclusion Algorithm

Lamport's Bakery Mutual Exclusion Algorithm. This section shows how read/write-based Lamport's Bakery algorithm [20] can be enriched with access to the failure detector QP to build a lock object which copes with asynchrony and any number of process crashes. Lamport's basic algorithm is pretty simple and elegant. It appears in Figure 2 from which the boxes at line 1, line 6, and line 7 are suppressed. When a process p_i wants to enter the critical section it first computes a label (its ticket number) at line 3. As this computation is asynchronous, p_i informs the other processes it is computing its ticket value by raising its flag $FLAG[i]$ (lines 2 and 4). The order on the labels is the classical lexicographical ordering on pairs of integers.

When the "doorway" statements are executed (lines 1-4), p_i waits until it is the process with the smallest labels among all the non-crashed processes (waiting room implemented by lines 5-8). (Pedagogical presentations and proofs of Lamport's Bakery algorithm can be found in [28, 32].) This algorithm satisfies the starvation-freedom property.

Exploiting the power provided by the failure detector QP. The enrichment appears in the three boxes. First, when p_i invokes entry(), it waits until its local failure detector module trusts it (line 1). This ensures that, if p_i is correct, it will appear in all the sets TRUSTED_j, and, if it crashes, it will eventually appear in all the sets CRASHED_j (before crashing it was then in INIT_j or TRUSTED_j). The two other boxes (lines 6-7), allows a process not to be blocked forever by a crashed process.

It follows from these observations that the proof is essentially the same that the one of Lamport's Bakery algorithm. Moreover as in the Bakery algorithm, due to the fact that crashes are stable (once

```
init: $\forall j \in \{1,\dots,n\} : FLAG[j] \in \{\texttt{down}, \texttt{up}\}$, initialized to down;
      $\forall j \in \{1,\dots,n\} : LABEL[j] \in \mathbb{N}$, initialized to 0.

operation entry() is
(1)   $\boxed{\mathsf{wait}\ (i \in \text{TRUSTED}_i);}$
(2)   $FLAG[i] \leftarrow \texttt{up}$;
(3)   $LABEL[i] \leftarrow \max(LABEL[1], \dots, LABEL[n]) + 1$;
(4)   $FLAG[i] \leftarrow \texttt{down}$;
(5)   for all $k \neq i$ do
(6)       $\mathsf{wait}\big((FLAG[k] = \texttt{down}) \vee \boxed{k \in \text{CRASHED}_i}\big)$;
(7)       $\mathsf{wait}\big((LABEL[k] = 0) \vee (LABEL[i], i) < (LABEL[k], k)) \vee \boxed{k \in \text{CRASHED}_i}\big)$;
(8)   end for.

operation exit() is
(9)   $LABEL[i] \leftarrow 0$.
```

Figure 2: QP-based version of Lamport's bakery mutual exclusion algorithm

crashed, a process remains crashed forever), this fault-tolerant version satisfies the starvation-freedom property (which is stronger than deadlock-freedom) of the basic bakery algorithm, and we have the following theorem.

Theorem 2. *The fault-tolerant QP-*based mutual exclusion *algorithm described in Fig.* 2 *satisfies the mutual exclusion, starvation-freedom, and wait-free exit properties.*

6 Mutual Exclusion 4: Extracting QP from Crash-Tolerant Mutual Exclusion Algorithms

This section shows that QP is optimal, i.e., it provides the processes with the weakest information on failures that allows the mutual exclusion problem to be solved in the read/write crash-prone model. To that end, it describes an algorithm that builds ("extracts" in the failure detector parlance) a failure detector satisfying the properties defining QP from ANY algorithm that solves deadlock-free mutual exclusion in presence of process crashes. See Figure 3.[2]

$$\text{Any crash-prone mutex algorithm } A \longrightarrow \boxed{\text{Extraction algorithm}} \longrightarrow QP \text{ failure detector}$$

Figure 3: Extracting QP from any crash-prone mutual exclusion algorithm

6.1 Extraction algorithm.

Let A be any algorithm that solves the deadlock-free mutual exclusion problem in the presence of any number of process crashes. An algorithm E that "extracts" QP from A is presented in Figure 4. "Extract" means that, at any time, the algorithm E outputs the sets TRUSTED_i and CRASHED_i at every process p_i, and these sets satisfy the properties defining the failure detector QP. To this end, the processes share a set of n locks objects and cooperate by executing n concurrent tasks.

[2]For the interested reader, extraction algorithms for other weakest failure detectors are described in [3, 5, 8, 9].

A set of n lock objects and a set of n tasks. As just said, in the extraction algorithm E, the processes access n deadlock-free lock objects denoted $MT[1..n]$. The set of n tasks is denoted $T[1..n]$. Let us consider an index $i \in \{1, ..., n\}$. As shown in Fig. 4, there are two possible behaviors for a process that executes a task.

- The code of process p_i for the task $T[i]$ is defined by the lines 2-4, protected by the lock $MT[i]$
- The code of a process p_j such that $p_j \neq p_i$ for the task $T[i]$ is empty (line 8), protected by the lock $MT[i]$. Hence, this code is a pure synchronization code.

The important point here is that, when p_i and p_j execute the task $T[i]$, the Boolean $SUCC[i]$ (set to `true` by p_i at line 2 and read by p_j at line 6) impose the following synchronization pattern: the invocation of $MT[i]$.entry() by p_j cannot terminate before p_i invokes $MT[i]$.exit().

```
init: $\forall j \in \{1, \ldots, n\}$ $SUCC[j] \in \{$true, false$\}$ initialized false;
      TRUSTED$_i \leftarrow \emptyset$; CRASHED$_i \leftarrow \emptyset$ % output of the failure detector.

run in parallel the tasks $T[1], \ldots, T[n]$ defined as follows:
task $t_i$ is
(1)  $MT[i]$.entry();
(2)  $SUCC[i] \leftarrow$ true;
(3)  TRUSTED$_i \leftarrow$ TRUSTED$_i \cup \{i\}$;
(4)  wait(false); % wait forever
(5)  $MT[i]$.exit().

task $t_k$, $1 \leq k \leq n \wedge k \neq i$, is
(6)  wait($SUCC[k]$);
(7)  TRUSTED$_i \leftarrow$ TRUSTED$_i \cup \{k\}$;
(8)  $MT[k]$.entry(); no-op; $MT[k]$.exit();
(9)  $\langle$TRUSTED$_i$, CRASHED$_i$ $\rangle \leftarrow \langle$TRUSTED$_i \setminus \{k\}$, CRASHED$_i \cup \{k\}\rangle$.
```

Figure 4: Extracting QP from a crash-prone mutual exclusion algorithm A (code for process p_i)

Mutex-based cooperation between the processes executing task $T[i]$. The n tasks are executed in parallel. As far as the task $T[i]$ is concerned, the extraction algorithm E works as follows.

- Process p_i first invokes $MT[i]$.entry() (line 1), while (due to the Boolean $SUCC[i]$, which is set to true at line 2 only by p_i) any other process p_j must wait at line 6 before invoking $MT[i]$.entry(), i.e., the invocation of $MT[i]$.exit() by p_j at line 8 can succeed only after p_i has executed lines 2-4. It then follows from the wait statement on line 4 executed by p_i that, if p_i does not crash, p_j will never invoke of $MT[i]$.entry(), and consequently, will remain blocked at line 6 as far as the task $T[i]$ is concerned.
- If p_i is correct, it never executes $MT[i]$.exit(). Then, due to the mutual exclusion realized by $MT[i]$, no other process p_j will be able to complete its invocation of $MT[i]$.entry() (line 8) and consequently, i will never be removed from the set TRUSTED$_j$ at line 9.
- If p_i crashes before executing line 2, $SUCC[i]$ is never set to `true`, and p_i remains in the set INIT$_j$ $=\Pi \setminus$ (TRUSTED$_j \cup$ CRASHED$_j$) at each p_j.
- If p_i crashes after $SUCC[i]$ has been assigned to the value `true` at line 2, it follows from the deadlock-freedom property of $MT[i]$ that at least one of the other processes p_j executes $MT[i]$.entry() and the corresponding "no-op" internal operation (line 8). As the number of processes $p_j \neq p_i$ is bounded, and each of them invokes $MT[i]$.entry() and $MT[i]$.exit() (line 8), it follows from the deadlock-freedom property of $MT[i]$ that each process p_j (that does not crash) executes the line 8. Hence, p_j removes i from TRUSTED$_j$ and adds it to CRASHED$_j$ (line 9).

6.2 Optimality of the Failure Detector QP

Theorem 3. *The algorithm described in Fig.* 4 *extracts a failure detector satisfying the properties of QP from any algorithm solving deadlock-free mutual exclusion in any read/write n-process system where any number of processes may crash.*

Proof The three following properties are direct consequences of the text of the extraction algorithm. They are left to the reader.

- $\forall\, \tau$: $\text{TRUSTED}_i(\tau) \cap \text{CRASHED}_i(\tau) = \emptyset$.
- $j \in \text{TRUSTED}_i(\tau) \Rightarrow \big(\forall\, \tau' \geq \tau\text{: } j \in \text{TRUSTED}_i(\tau') \cup \text{CRASHED}_i(\tau')\big)$.
- $j \in \text{CRASHED}_i(\tau) \Rightarrow \big(\forall\, \tau' \geq \tau\text{: } j \in \text{CRASHED}_i(\tau')\big)$.

The rest of the proof addresses the other properties defining QP.

If $j \in \text{TRUSTED}_i(\tau)$, p_j was added to TRUSTED_i before time τ (at line 3 if $i = j$, and at line 7 if $i \neq j$). Hence, $SUCC[j]$ was previously set to `true` (line 2). Thus, when a correct process p_k ($k \neq j$) executes task t_j, it finds $SUCC[j]$ equal to `true`, and consequently adds j to TRUSTED_k at line 7. Hence, we have

- $j \in \text{TRUSTED}_i(\tau) \Rightarrow \big(\forall k \in \mathcal{C} : \exists \tau'\text{: } j \in \text{TRUSTED}_k(\tau') \cup \text{CRASHED}_k(\tau')\big)$.

If $j \in \text{CRASHED}_i(\tau)$, at some time $\tau' \leq \tau$, p_i inserted p_j in $\text{CRASHED}_i(\tau')$ (line 9). Hence, p_i previously executed lines 6 and 7 of task t_j. At line 6, p_i found $SUCC[j] = \texttt{true}$. Process p_j is the only process that can set $SUCC[j]$ to `true`, and it does it at line 2, after having completed $MT[j]$.entry(). After that, p_j remains in its critical section controlled by $MT[j]$. As due to the mutual exclusion property of $MT[j]$, the processes p_i and p_j cannot be together in the critical section, it follows that p_j crashed before p_i enters in critical section protected by $MT[j]$. Hence, it follows that

- $i \in \mathcal{C}, j \in \text{CRASHED}_i(\tau) \Rightarrow j \in \mathcal{F}(\tau)$.

Let us now assume that p_i is correct, and p_j crashed before time τ. We show that $\big(\exists \tau' \geq \tau\text{: } j \notin \text{TRUSTED}_i(\tau')\big)$. There are two cases, either (1) $\forall \tau' : j \notin \text{TRUSTED}_i(\tau')$, and the property holds, or (2) $\exists\, \tau' : j \in \text{TRUSTED}_i(\tau')$ and p_i inserts p_j in TRUSTED_i (line 7).

Assume that $j \in \mathcal{F}(\tau)$ and $\exists\, \tau' : j \in \text{TRUSTED}_i(\tau')$. As p_i inserted p_j in TRUSTED_i (line 7), it previously found $SUCC[j]$ equal to `true` (line 6). As p_j is the only process that can assign $SUCC[j]$ to `true`, it does it at line 2 in the critical section controlled by $MT[j]$ after having executed $MT[j]$.entry(). Any correct process that finds $SUCC[j]$ equal to `true`, invokes $MT[j]$.entry(). As the number of processes is bounded, by the deadlock freedom property of $MT[j]$, p_i executes line 8 and terminates it. It then moves j from TRUSTED_i to CRASHED_i, and never inserts it again in TRUSTED_i. Hence, we have

- $i \in \mathcal{C},\ j \in \mathcal{F}(\tau) \Rightarrow \big(\exists \tau' \geq \tau\text{: } j \notin \text{TRUSTED}_i(\tau')\big)$.

Let us finally consider the case where both p_i and p_j are correct. In the extraction algorithm, p_j executes $MT[j]$.entry() of task t_j. Until p_j sets $SUCC[j]$ to `true`, all the other processes that execute task t_j remain blocked at line 6 and do not execute $MT[j]$.entry(). Due to the deadlock-freedom property of $MT[j]$, p_j will enter in critical section and set $SUCC[j]$ to `true`. Then, p_j is added to $\text{TRUSTED}_i(\tau)$ at line 3 if $i = j$, or at line 7 if $i \neq j$. Hence, it follows that

- $i \in \mathcal{C}, j \in \mathcal{C} \Rightarrow \big(\exists \tau\text{: } j \in \text{TRUSTED}_i(\tau)\big)$. □

From two observations to a theorem.

- Observation O1. Theorem 3 shows that QP can be obtained from any crash-tolerant deadlock-free mutual exclusion algorithm. This means that, if a failure detector D allows us to solve deadlock-free mutual exclusion, QP can be built from D. (This means that D contains at least as much information on failures than the one provided by QP.)
- Observation O2. On another side, the algorithm described in Fig. 2 shows that, in a read/write n-process system where any number of processes may crash, starvation-free (hence deadlock-free) mutual exclusion can be solved from QP.

As starvation-freedom is a stronger progress condition than deadlock-freedom, the following theorem is a consequence of the previous observations.

Theorem 4. *QP is the weakest failure detector for both deadlock-free and starvation-free mutual exclusion in read/write n-process systems where any number of processes may crash.*

Based on totally different and more involved formalism, a failure detector, denoted Γ^1, suited to starvation-free mutual exclusion is described in [2], where it is proved to be optimal. Hence we have the following corollary.

Corollary 1. *Γ^1 and QP are equivalent for starvation-free mutual exclusion.*

The reader interested in the weakest failure detector for mutual exclusion in asynchronous crash-prone message-passing systems will consult [10].

7 Participant-Restricted Consensus 1: Definition

In the consensus problem each process p_i invokes once the $\mathsf{propose}()$. This operation takes a value as input parameter (called input or proposed value) and returns a result (called *decided value*). The consensus problem is defined by the following properties, where it is assumed that all the processes that do not crash invoke the operation $\mathsf{propose}()$.

- Validity: If a process decides a value v, this value was proposed by some process.
- Agreement: No two processes decide different values.
- Termination: If a process invokes $\mathsf{propose}()$ and does not crash, it decides.

When process p_i invokes $\mathsf{propose}(v)$ we say "p_i proposes v". When this invocation terminates and returns value w, we say "p_i decides w".

Participant-restricted consensus. A process p_i participates in a consensus instance if it invokes $\mathsf{propose}()$ (from an operational point of view, this corresponds to the first shared memory access invoked by $\mathsf{propose}()$).

Participant-restricted consensus is consensus in which not all the correct processes are required to participate. Hence, a non-participating process can be correct or faulty. Moreover the subset of processes that participate is not known in advance.

8 Participant-Restricted Consensus 2: the Failure Detector Ω^*

8.1 The eventual leader failure detector Ω

The *eventual leader* failure detector, denoted Ω, was introduced in [5], where it is shown to be the weakest failure detector to solve consensus in asynchronous message-passing systems in which a majority of processes do not crash. This failure detector provides each process p_i with a read-only local variable LEADER_i, which always contains a process identity, and is such that, after an unknown but finite period, the variables LEADER_i of all the correct processes contain the same identity and this identity is the identity of a correct process (this property is called *eventual leadership*)[3].

An Ω-based consensus algorithm for asynchronous read/write systems in which any number of processes may crash is presented in [23], where it is shown that Ω is the weakest failure detector to solve consensus in asynchronous read/write systems in which any number of processes may crash.

Be the communication medium read/write registers or message-passing, the Ω-based consensus algorithms implicitly assume that all the processes participate in the consensus. This is because the process that is eventually elected as the common leader by Ω can be *any* correct process. If this process does not participate, consensus cannot be solved. It follows that Ω is not the weakest failure detector to solve consensus if some correct processes do not participate.

8.2 The Eventual Leader Failure Detector Ω^*

The failure detector Ω^* was introduced in [18, 30]. It is used in [18] to boost liveness properties of concurrent objects, and in [30] to solve k-set agreement (a generalization of consensus, which corresponds to the case $k = 1$).

The failure detector $\Omega^*(X)$. Given any set X of processes, $\Omega^*(X)$ provides each process p_i with a read-only local variable $\text{LEADER}_i(X)$ such that the following properties are satisfied.

- Validity: At any time, any local variable $\text{LEADER}_i(X)$ contains the identity of a process of X.
- Restricted eventual leadership: There is an unknown but finite time after which the local variables $\text{LEADER}_i(X)$ of the correct processes of X contain the same process identity, which is the identity of a correct process of X.

Hence, given any non-empty set of processes X, there is an arbitrary period during which the processes of X have arbitrary leaders, but this anarchy period is finite. When this period terminates the correct processes of X agree on the same leader, which is one of them. Let us remark that when $X = \Pi$ (the whole set of processes), $\Omega^*(X)$ boils down to Ω.

The failure detector Ω^*. This failure detector considers all the non-empty subsets $X \subseteq \Pi$. It provides each process p_i with $2^n - 1$ failure detectors $\Omega^*(X)$, one for each non-empty subset X of Π.

The following theorem follows directly from the definition of $\Diamond P$ (while Ω, Ω^*, and $\Diamond P$ belongs to the family of eventual failure detectors, $\Diamond P$ is the only of them that, after some finite time, behaves as the perfect failure detector P –which was defined in the Introduction–).

Theorem 5. $\Omega \prec \Omega^* \prec \Diamond P \prec P$.

9 Participant-Restricted Consensus 3: an Ω^*-based Algorithm

This section presents an Ω^*-based consensus algorithm suited to the participant-restricted process model.

[3]An algorithm implementing Ω in asynchronous crash-prone read/write systems enriched with very weak synchrony assumptions is described in [15].

9.1 A high level communication object: adopt/commit/abort

The adopt/commit/abort object is a fault-tolerant object introduced in [17]. It is a one-shot object that provides the processes with a single operation denoted $\mathsf{adopt\_commit}()$. This operation takes a value as input parameter (we then say that the invoking process *proposes* that value) and returns a pair $\langle d, v\rangle$, where d is a control tag and v a value (we then say that the invoking process *decides* the pair $\langle d, v\rangle$). (It is assumed that no process invokes $\mathsf{adopt\_commit}(v)$ with $v = \bot$.)

Let ACA be an adopt/commit/abort object. Its behavior is defined by the following properties.

- Termination: An invocation of $ACA.\mathsf{adopt\_commit}()$ by a correct process terminates.
- Validity: This property is made up of two parts:
  - Output domain: If a process decides $\langle d, v\rangle$ then $d \in \{\texttt{commit}, \texttt{adopt}, \texttt{abort}\}$ and v is a value that was proposed to ACA.
  - Obligation. If all the processes that invoke $ACA.\mathsf{adopt\_commit}()$ propose the same value v, then the only pair that can be decided is $\langle \texttt{commit}, v\rangle$.
- Quasi-agreement. If a process decides $\langle \texttt{commit}, v\rangle$ then any other deciding process decides $\langle d, v\rangle$ with $d \in \{\texttt{commit}, \texttt{adopt}\}$.

Intuitively, an adopt/commit/abort object is a kind of very weak consensus object. It is easy to see (quasi-agreement property) that, if a process decides $\langle \texttt{abort}, -\rangle$, no process decides $\langle \texttt{commit}, -\rangle$. Differently, if a process decides $\langle \texttt{adopt}, -\rangle$, it cannot conclude which control tags were decided by other processes, it can only conclude that they belong either to the set $\{\texttt{adopt}, \texttt{commit}\}$, or (exclusive) to the $\{\texttt{adopt}, \texttt{abort}\}$.

This very weak agreement object can be implemented in any asynchronous read/write systems in which any number of processes may crash [17, 28]. It consists in two consecutive "for" loops, which access $2n + 1$ atomic read/write registers (two arrays and a shared variable). Variants of the adopt/commit/abort object have been used in asynchronous message-passing systems in with a majority of processes do not crash [1, 25, 29].

9.2 Description of the algorithm

Global and local data structures. A consensus object is realized from the following shared objects.

- DEC is a multi-writer multi-reader atomic register initialized to the default value $\bot$. Its aim is to contain the value decided by the consensus object.
- $ACA[1..)$ is an unbounded array of adopt/commit/abort objects. $ACA[r]$ is the object used by a process when it executes its rth round. As we have seen, an invocation $ACA[r].\mathsf{commit\_adopt}()$ returns a pair res such that $res.tag \in \{\texttt{commit}, \texttt{adopt}, \texttt{abort}\}$ and $res.val$ contains a proposed value.
- $PART[1..n]$ is an array of single-writer multi-reader atomic registers initialized to $[\texttt{out}, \cdots, \texttt{out}]$. Only p_i can write (only once) into $PART[i]$, namely it writes $\texttt{in}$ when it starts participating.

Each process p_i manages three local variables: est_i that contains its current estimate of the decision value, r_i that contains its current round number, and the set $part_i$ which contains the processes that p_i currently knows as participants.

The algorithm implementing the operation propose()**.** This very simple algorithm is described in Fig. 5. The processes execute asynchronous rounds until they decide. Let us remind that only the processes participating in the consensus execute rounds and the participating processes are not necessarily executing the same round at the same time. A process decides a value v when it invokes return(v) at line 13.

```
operation propose($v_i$) is
(1)  $PART[i] \leftarrow$ in; $est_i \leftarrow v_i$; $r_i \leftarrow 0$;
(2)  while ($DEC = \perp$) do
(3)      $part_i \leftarrow \{k \mid PART[k] =$ in$\}$.
(4)      if (LEADER$(part_i) = i$) then
(5)          $r_i \leftarrow r_i + 1$;
(6)          $res_i \leftarrow ACA[r_i]$.commit_adopt$(est_i)$;
(7)          if $res_i = \langle$commit$, -\rangle$
(8)              then $DEC \leftarrow res_i.val$
(9)              else $est_i \leftarrow res_i.val$
(10)         end if
(11)     end if
(12) end while;
(13) return($DEC$)
end operation.
```

Figure 5: From adopt/commit/abort objects and Ω^* to consensus

A process p_i first posts the fact it starts participating, and deposits the value v_i it proposes into its local estimate est_i (line 1). Then, it loops until it reads $DEC \neq \perp$ (line 2). If $DEC \neq \perp$, it decides the value of DEC (line 13).

If $DEC = \perp$, a process p_i first builds a local view $part_i$ of the set of the participating processes (line 3). Then, it checks whether it is a leader with respect to the process set $part_i$ (line 4). If it is not, it re-enter the "while" loop. In the other case, p_i tries to impose its current value of est_i as the decision value. To that end it progresses to the next round (line 5) and proposes the value of est_i to the adopt/commit/object object associated with its current round r_i, namely $ACA[r_i]$ (line 6). This invocation returns a pair made up of a tag and a value. If the returned pair is $\langle$commit$, v\rangle$ (line 7), p_i writes v into DEC (line 8) and decides (line 13). If it does not obtain the tag commit, p_i adopts the value v it has just obtained from $ACA[r_i]$ as its new estimate value est_i before re-entering the " while" loop.

9.3 Proof of the algorithm

Theorem 6. *The algorithm described in Figure* 5 *is a wait-free implementation of a consensus object in the asynchronous read/write model enriched with* Ω^* *and where any subset of processes participate.*

Proof The consensus validity property states that a decided value is a proposed value. The proof of this property follows from the following observations. Initially, the estimates est_i of the participating processes contain proposed values (line 1). Then, the input parameter of any invocation of commit_adopt() is such an estimate, and thanks to the "output domain" validity property of the objects $ACA[1]$, $ACA[2]$, etc., the value returned $res_i.val$ is also an estimate of a proposed value (line 6). Hence, only proposed values can be assigned to DEC (line 8). It follows that a decided value is a value proposed by one of the participating process.

The consensus agreement property states that no two processes decide different values. Let us consider the first round r during which a process assigns a value to the atomic register DEC (line 8).

Let p_i be a process that issues such an assignment. It follows from lines 6–7 that p_i has obtained $res_i = \langle \texttt{commit}, v\rangle$ from $ACA[r]$. Moreover, it follows from the quasi-agreement property of this object that any other process p_j that returns from ACA.adopt_commit() during the same (asynchronous) round, obtains $res_j = \langle \texttt{commit}, v\rangle$ or $res_j = \langle \texttt{adopt}, v\rangle$. If p_j obtains $\langle \texttt{commit}, v\rangle$, it writes v into the atomic register DEC, and if it obtains $\langle \texttt{adopt}, v\rangle$, it writes v into est_i. It follows that, any process that executes round r (and does not crash) either decides before entering round $r+1$ or starts round $r+1$ with $est_j = v$. This means that, from round $r+1$, the only value that a process can propose to $ACA[r+1]$ is v. It follows that, from then on, only v can be written into the atomic register DEC, which concludes the proof of the consensus agreement property.

The consensus termination property states that any invocation of propose() by a correct process terminates. Let us first observe that there is a finite time τ_0 after which only the correct processes execute the algorithm and all of them compute forever the same set of participating processes $part$. Let us also observe that, due to the termination property of the adopt/commit/abort objects, whatever r, any invocation of $ACA[r]$.adopt_commit() issued by a correct process terminates. Consequently, the proof of the consensus termination property amounts to show that a value is eventually written in DEC. We prove this by contradiction.

Let us assume that no value is ever written in DEC. It follows from the eventual leadership property of Ω^* that there is a finite time $\tau_1 \geq \tau_0$ after which all the invocations of $\text{LEADER}(part)$ forever return the same identity $\ell \in part$ such that p_ℓ is a correct process. It follows that, after some finite time p_ℓ is the only process that repeatedly executes lines 5–11. As (by assumption) it never writes into DEC, p_ℓ executes an infinite number of rounds. It follows that eventually there is a round number r executed only by p_ℓ. It then follows from the obligation property of $ACA[r]$ that this invocation returns the pair $\langle \texttt{commit}, est_\ell\rangle$. Consequently, p_ℓ executes line 8 and writes est into DEC, which contradicts the initial assumption and concludes the proof of the termination property. $\square_{Theorem\ 6}$

10 Participant-Restricted Consensus 4: Optimality of Ω^*

Theorem 7. *Ω^* is the weakest failure detector to implement participant-restricted consensus in an asynchronous read/write system in which any number of processes may crash.*

Proof The fact that Ω^* allows participant-restricted consensus to be solved follows from the existence of the algorithm described in Fig. 5.

The fact it is the weakest results from the following observation. Given an execution, let $part \subseteq \Pi$ be the set of processes that participate in the consensus (i.e., the set of processes that invoke the operation propose()). In such an execution, it follows from its definition that $\Omega^*(part)$ behaves exactly as Ω in a system of processes $part$. As Ω is the weakest failure detector to solve consensus in a model in which all the processes are assumed to participate, it follows that $\Omega^*(part)$ is the weakest when only processes in $part$ participate. $\square_{Theorem\ 7}$

11 Conclusion

This paper was on the minimal information on failures which allow to solve mutual exclusion and participant-restricted consensus in asynchronous systems where (i) processes communicate by accessing atomic read/write registers, and (ii) any number of processes may crash. It has presented two failure detectors, denoted QP and Ω^* that allow mutual exclusion and participant-restricted consensus to be solved. The paper has also shown that these failure detectors are optimal in the sense the information on failures provided by any other failure-detector solving the same problem includes the information provided by QP, resp. Ω^*.

On the pedagogical side, it is interesting to observe that Section 6 is simple enough to introduce students to the notion of an *extraction* algorithm used to prove optimality of a failure detector (i.e., to know which are the weakest information on failures needed to solve an otherwise impossible problem in asynchronous read/write crash-prone systems).

Acknowledgments

The authors want to warmly thank the referees for their constructive comments which helped improve both the content and the presentation of the article.

This work was partially supported by the French ANR project 16-CE40-0023-03 devoted to layered and modular structures in distributed computing.

References

[1] Ben-Or M., Another advantage of free choice: completely asynchronous agreement protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 27-30 (1983)

[2] Bhatt V. and Jayanti P., On the existence of weakest failure detectors for mutual exclusion and k-exclusion. *23rd Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 325-339 (2009)

[3] Bonnet F. and Raynal M., A simple proof of the necessity of the failure detector Σ to implement an atomic register in asynchronous message-passing systems. *Information Processing Letters*, 110(4): 153-157 (2010)

[4] Brinch Hansen P. (Editor), *The origin of concurrent programming*. Springer, 534 pages (2002)

[5] Chandra T., Hadzilacos V. and Toueg S. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722 (1996)

[6] Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)

[7] Delporte-Gallet C., Fauconnier H. and Guerraoui R., A Realistic look at failure detectors. *Proc. 43rd Annual IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN'02)*, IEEE Computer Press, PP. 345-353 (2002)

[8] Delporte-Gallet C., Fauconnier H. and Guerraoui R., Tight failure detection bounds on atomic object implementations. *Journal of the ACM*, 57(4), Article 22, 32 pages (2010)

[9] Delporte-Gallet C., Fauconnier H., Guerraoui R., Hadzilacos V., Kouznetsov P., and Toueg S., The weakest failure detectors to solve certain fundamental problems in distributed computing. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 338-346 (2004)

[10] Delporte-Gallet C., Fauconnier H., Guerraoui R., and Kouznetsov P., Mutual exclusion in asynchronous systems with failure detectors. *Journal od Parallel and Distributed Computing*, 65:492-505 (2005)

[11] Delporte-Gallet C., Fauconnier H., and Raynal M., Fair synchronization in the presence of process crashes and its weakest failure detector. *33rd IEEE International Symposium on Reliable Distributed Systems, (SRDS'14)*, IEEE Press, pp. 161-170 (2014)

[12] Delporte-Gallet C., Fauconnier H., and Raynal M., On the weakest failure detector for read/write-based mutual exclusion. *Proc. 33nd Int'l Conference on Advanced Information Networking and Applications (AINA'19)*. Springer Series "Advances in Intelligent Systems and Computing", Vol. AISC 926, pp. 272-285 (2019)

[13] Delporte-Gallet C., Fauconnier H., and Raynal M., Participant-restricted consensus in asynchronous crash-prone read/write systems and its weakest failure detector. *Proc. 15th International Conference on Parallel Computing Technologies (PaCT'19)*, Springer, LNCS 11657, pp. 419-430 (2019)

[14] Dijkstra E.W., Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569 (1965)

[15] Fernández A., Jiménez E., Raynal M., and Trédan G., A timing assumption and two t-resilient protocols for implementing an eventual leader service in asynchronous shared-memory systems. *Algorithmica*, 56(4):550-576 (2010)

[16] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of distributed consensus with one faulty process *Journal of the ACM*, 32(2):374-382 (1985)

[17] Gafni E., Round-by-round fault detectors: unifying synchrony and asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 143-152 (1998)

[18] Guerraoui R., Kapalka M,. and Kuznetsov P., The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6): 415-433 (2008)

[19] Hélary J.-M., Hurfin M., Mostéfaoui A., Raynal M., and Tronel F., Computing global functions in asynchronous distributed systems with perfect failure detectors. *IEEE Transactions on Parallel Distributed Systems*, 11(9):897-909 (2000)

[20] Lamport L., A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453-455, (1974)

[21] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565 (1978)

[22] Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)

[23] Lo W.-K. and Hadzilacos V., Using failure detectors to solve consensus in asynchronous shared memory systems. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer, LNCS 857, pp. 280-295 (1994)

[24] Loui M. and Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, JAI Press, 4:163-183 (1987)

[25] Mostéfaoui A. and Raynal M., Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach. *Proc. 13th Int'l Symposium on Distributed Computing (DISC'99)*, Springer LNCS 1693, pp. 49-63 (1999)

[26] Pease M., Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228-234 (1980)

[27] Rajsbaum S. and Raynal M., Mastering concurrent computing through sequential thinking. *Communications of the ACM*, 63(1):78-87 (2020)

[28] Raynal M., *Concurrent programming: algorithms, principles, and foundations*. Springer, 515 pages, ISBN 978-3-642-32027-9 (2013)

[29] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer, 492 pages, ISBN: 978-3-319-94140-0 (2018)

[30] Raynal M. and Travers C., In search of the Holy Grail: looking for the weakest failure detector for wait-free set agreement. *Proc. 10th Int'l Conference on Principles of Distributed Systems (OPODIS'11)*, Springer, LNCS 4305, pp. 3-19 (2006)

[31] Schneider F.B., Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299-319 (1990)

[32] Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006)

[33] Taubenfeld G., Fair synchronization. *Proc. 27rd Symposium on Distributed Computing (DISC'13)*, Springer LNCS 8205, pp. 179-193, (2013)