RZ 3361 (# 93407) 20/08/2001
Computer Science 27 pages

Research Report

(Im)Possibilities of Predicate Detection in Crash-Affected Systems

Felix C. Gärtner[1] and Stefan Pleisch[2]

[1]Department of Computer Science
Darmstadt University of Technology
D-64283 Darmstadt
Germany
felix@informatik.tu-darmstadt.de

[2]IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
spl@zurich.ibm.com

(Im)Possibilities of Predicate Detection in Crash-Affected Systems

Felix C. Gärtner

Department of Computer Science, Darmstadt University of Technology, D-64283 Darmstadt, Germany

Stefan Pleisch

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

In an asynchronous system, where processes can crash, perfect predicate detection for general predicates is difficult to achieve. A general predicate thereby is of the form $\alpha \wedge \beta$, where α and β refer to a normal process variable and to the operational state of that process, respectively. Indeed, the accuracy of predicate detection largely depends on the quality of failure detection. In this paper, we investigate the predicate detection semantics that are achievable for general predicates using either failure detector classes $\Box\Diamond\mathcal{P}$, $\Diamond\mathcal{P}$, or $\mathcal{P}$. For this purpose, we introduce weaker variants of the predicate detection problem, which we call *stabilizing* and *infinitely often accurate*. We show that perfect predicate detection is impossible using the aforementioned failure detectors. Rather, $\Diamond\mathcal{P}$ and $\mathcal{P}$ only allow stabilizing predicate detection. Consequently, we explore alternative approaches to perfect predicate detection: introducing a stronger failure detector, called *ordered perfect*, or restricting the general nature of predicates.

1 Introduction

Testing and monitoring distributed programs involves the basic task of detecting whether a predicate holds during the execution of the system. For example, a software engineer might want to detect the predicate "variable x has changed to value 2" to find out at what point in the execution x takes on a bad value. Predicate detection in distributed settings is a well-understood problem and many techniques together with their detection semantics have been proposed [7]. However, most of the techniques have been proposed under the assumption that no faults occur in the system. Hence, most of the methods proposed in the literature are not robust in the sense that they offer no guarantees if faults such as message losses or process crashes occur in the system.

In an asynchronous system where processes can crash, a general predicate detection mechanism should also detect these crash events. Chandra and Toueg [3] proposed to encapsulate the functionality of failure detection into a separate module and specify it using axiomatic properties. Such a *failure detector* can be used to locally maintain information about the operational state of the processes. Based on the quality of failure detection, different classes of failure detectors can be defined. Most relevant to this paper are the classes of perfect, eventually perfect, and infinitely often accurate failure detectors (denoted $\mathcal{P}$, $\Diamond\mathcal{P}$, and $\Box\Diamond\mathcal{P}$, respectively) [3, 11]. Chandra and Toueg assume a *query model* for their failure detectors, i.e., the application must query the failure detector to receive information about the operational states of processes. In contrast, infinitely often accurate failure detectors [11] assume an *interrupt model*. In this model, the application is notified when the failure detector changes its perception of the operational state of a process. In other words, the failure detector sends a failure detector event if it suspects a process it has previously not suspected, or if it does not suspect a process any more. In this paper, we use the interrupt model but show that our results are valid in both models for failure detector classes $\Diamond\mathcal{P}$ and $\mathcal{P}$.

Standard predicate detection techniques aim at monitoring conditions which are composed of predicates on the local state space of processes [7]. For instance, a predicate $x_i = 1 \wedge y_i = 2$ evaluates on the variables x and y in the local state of process p_i. In this paper, we consider the detection of predicates that are boolean combinations of predicates on the local state and predicates on the operational state of a process. This allows us to evaluate predicates of the form $x_i = 1 \wedge \mathit{crashed}_i$, where $\mathit{crashed}_i$ is a predicate that is true iff (if and only if) application process p_i has crashed. Ideally, a predicate detection algorithm never erroneously detects such a predicate and does not miss any occurrence of the predicate in the underlying computation. However, the quality of the underlying failure detector severely limits the quality of predicate detection. We show that *perfect* predicate detection is generally impossible with failure detectors of type $\Box\Diamond\mathcal{P}$ and $\Diamond\mathcal{P}$. Rather surprisingly, the impossibility still holds for $\mathcal{P}$. We investigate weaker variants of predicate detection which we call *stabilizing* and *infinitely often accurate*. Briefly spoken, a predicate detection algorithm is stabilizing if it eventually stops making false detections and it is infinitely often accurate if it has infinitely many phases where it does not issue false detections. We also investigate two conditions under which perfect predicate detection is solvable. The first is the existence of a novel type of failure detector which we call *ordered perfect* (denoted $\widehat{\mathcal{P}}$) and which is strictly stronger that $\mathcal{P}$. The second condition imposes restrictions on the generality of predicates.

Apart from clarifying the relation between predicate detection and failure detection, this work wishes to stress the connection between "stabilizing failure detectors" [3] and *self-stabilization* [8] (which has only partly been done by other authors [15, 2]) and, hence, argue that self-stabilization is a concept of eminent practical importance. Furthermore, our results manifest a drawback of the approach that uses abstract failure detectors to solve problems in distributed computing, namely, that for every new problem it is necessary to adapt the failure detector properties, a highly non-trivial task.

Related work. While predicate detection in fault-free environments has been intensely studied [7], solving the task in faulty environments is not yet very well understood. To our knowledge, Shah and Toueg [17] were the first to investigate this by adapting the snapshot algorithm of Chandy and Lamport [4] with a simple timeout mechanism. Chandra and Toueg [3] later argued to define the functionality of failure detection in an abstract way and proposed a rich set of failure detector classes. However, these classes were meant to help solve the *consensus* problem and not the problem of predicate detection. Garg and Mitchell [10] investigate the predicate detection problem again and define an *infinitely often accurate* failure detector, i.e., a failure detector which is implementable in asynchronous systems [11], but they restrict the scope of the predicates to *set-decreasing* and *conjunctive* predicates. A predicate is set-decreasing whenever it holds for a set of processes, it also holds for a subset of these processes. For example, "no token" is a set-decreasing predicate. The set-decreasing property is used to ensure liveness of the predicate detection mechanism by ignoring those processes which are currently suspected. Conjunctive predicates can be expressed as the conjunction of local process predicates and channel predicates. Furthermore, channels are assumed to be *send-monotonic*, i.e., a false predicate does not become true by only sending messages. To our knowledge, our work is the first to investigate the relationship between predicate detection and failure detection in the general case.

While it is not clear whether Garg and Mitchell [10] or Shah and Toueg [17] consider predicates which contain references to the operational state of processes, Gärtner and Kloppenburg [12] explicitly allow these types of predicates but restrict themselves to environments where only infinitely often accurate failure detectors are available. Other authors have investigated the use of perfect failure detectors to detect special predicates, e.g., distributed deadlocks [20] or distributed termination [18].

The observation that perfect failure detectors do not allow to solve all problems which are solvable in synchronous systems has been previously made by Charron-Bost, Guerraoui, and Schiper [6] by exhibiting a problem that is solvable in synchronous systems but is not solvable in asynchronous systems augmented with perfect failure detectors (the *strongly dependent decision* problem). While Charron-Bost et al. [6] argue that this result has practical consequences with respect to the efficiency of atomic commitment, our result shows that there exists a practically relevant problem, namely predicate detection, which suffers from the deficiencies of perfect failure detectors.

Paper organization. After introducing the system assumptions in Section 2, we define three different semantics for the predicate detection problem in Section 3. In Section 4, we consider systems where crash failures occur, augmented with infinitely often accurate (Section 4.2), with eventually perfect (Section 4.3), and perfect failure detectors (Section 4.4). Our focus is on a system with one application process and one observer. We then generalize to multiple application processes and observers in Section 5. Finally, Section 6 concludes the paper and states potential future work.

2 System Assumptions

2.1 System Model

We consider a system with n *application processes* $p_1, ..., p_n$ and m *monitor processes* $b_1, \ldots, b_m$ (i.e., observers) whose task is to monitor the execution on the application processes. We denote the set of application processes by Π and the set of monitor processes by Ψ. If it is clear from the context we will refer to application processes simply as "processes" and to monitor processes as "monitors". Π forms the *application system*, whereas $\Pi \cup \Psi$ is called the *observation system*.

Processes communicate by message passing via FIFO channels in a fully connected network.

Communication is reliable, i.e., no messages are lost, duplicated, or altered. Our system is *asynchronous*, i.e., no boundaries on communication delays nor on relative processor speeds exist.

We use a discrete global clock to simplify the presentation of our model [3]. However, no process has access to this clock, it is merely a fictional device. For simplicity we take the range $\mathcal{T}$ of the clock to be the set $\mathbb{N}$ of natural numbers.

Application processes can fail by crashing. Once a process has crashed, it does not recover any more during the execution. A *failure pattern* F is a mapping from $\mathcal{T}$ to the powerset of Π, where $F(t)$ specifies the set of application processes which have crashed up until time t. We define $\mathit{crashed}(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $\mathit{correct}(F) = \Pi \setminus \mathit{crashed}(F)$. If $p \in \mathit{crashed}(F)$ we call it *faulty*. Consequently, a non-faulty process is called *correct*. For simplicity, we assume that monitor processes do not fail but we place no restriction on the number of faulty application processes.

2.2 Failure Detectors

Each monitor process has access to a local failure detector module that provides (possibly incorrect) information about failures that occur on application processes [3]. A *failure detector history* H is a mapping from $\Psi \times \mathcal{T}$ to the powerset of Π. The value of $H(b_j, t)$ denotes the value of the failure detector module on monitor b_j at time t. If $p_i \in H(b_j, t)$ we say that process p_i is *suspected* by monitor b_j at time t.

A *failure detector* $\mathcal{D}$ maps a failure pattern to a set of failure detector histories. Intuitively, $\mathcal{D}(F)$ is the set of histories which the failure detector could have produced in runs with failure pattern F. Failure detectors are defined in terms of a *completeness* and an *accuracy* property. These properties restrict the set of failure detector histories that are possible given a specific failure pattern. The completeness property requires that a failure detector eventually suspects processes that have crashed, while the accuracy property limits the number of mistakes a failure detector can make. We recall the definitions of accuracy and completeness which are relevant to this paper [3, 11]:

- (strong completeness) Eventually every application process that crashes is permanently suspected by every monitor process. Formally:

$$\forall F.\forall H \in \mathcal{D}(F).\exists t \in \mathcal{T}.\forall p \in \mathit{crashed}(F).\forall b \in \Psi.\forall t' \geq t.p \in H(b, t')$$

- (strong accuracy) No application process is suspected before it crashes. Formally:

$$\forall F.\forall H \in \mathcal{D}(F).\forall t \in \mathcal{T}.\forall p \in \Pi \setminus F(t).\forall b \in \Psi.p \notin H(b, t)$$

- (eventual strong accuracy) There is a time after which correct application processes are not suspected by any monitor.

$$\forall F.\forall H \in \mathcal{D}(F).\exists t \in \mathcal{T}.\forall t' \geq t.\forall p \in \Pi \setminus F(t').\forall b \in \Psi.p \notin H(b, t')$$

- (infinitely often accuracy) Correct application processes are not permanently suspected by any monitor.

$$\forall F.\forall H \in \mathcal{D}(F).\forall b \in \Psi.\forall p \in \mathit{correct}(F).\forall t \in \mathcal{T}.p \in H(b, t) \Rightarrow \exists t' > t.p \notin H(b, t')$$

The failure detectors we consider in this paper all satisfy strong completeness. A *perfect failure detector* additionally satisfies strong accuracy. An *eventually perfect failure detector* satisfies eventual strong accuracy instead of strong accuracy. Finally, an *infinitely often accurate failure detector* satisfies strong completeness and infinitely often accuracy.

Failure detectors are grouped into classes that represent the set of failure detectors satisfying the given properties. We denote the class of all perfect failure detectors by $\mathcal{P}$, the class of all eventually perfect failure detectors by $\diamond\mathcal{P}$, and the class of all infinitely often accurate failure detectors by $\square\diamond\mathcal{P}$.

Failure detectors are defined as passive modules, that can be queried at any time by an application [3]. This definition corresponds to a *query model* for failure detectors. $\mathcal{P}$ and $\diamond\mathcal{P}$ are defined in this model. On the other hand, $\square\diamond\mathcal{P}$ [11] makes no sense in the query model because, intuitively, an application could always query the failure detector module in periods when it is inaccurate. Consequently, infinitely often accurate failure detectors implicitly assume an *interrupt model*, where every change in the perception of the operational state of a process is notified to the application.

To enable the use of the same algorithms for predicate detection for $\mathcal{P}$, $\diamond\mathcal{P}$, and $\square\diamond\mathcal{P}$, we assume the interrupt model in this paper. However, we show later that the interrupt model and the query model are equivalent for failure detector classes $\mathcal{P}$ and $\diamond\mathcal{P}$. Hence, our results for $\mathcal{P}$ and $\diamond\mathcal{P}$ are also valid in the query model.

2.3 Algorithms and Runs

An algorithm A consists of a set of deterministic automata, one for each process. Following Chandra and Toueg [3], we define an *execution* or *run R of A using failure detector $\mathcal{D}$* as a tuple $R = (F, H_{\mathcal{D}}, I, S, T)$, where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a failure detector history of $\mathcal{D}$ for failure pattern F, I is a set of initial states of the application processes, S is an infinite sequence of steps of the algorithm, and T is a list of increasing time values indicating when the steps in S occurred. Runs must satisfy the usual fairness and well-formedness requirements [3]: (1) no process executes a step after crashing, (2) correct processes take an infinite number of steps, and (3) every message that was sent is eventually received.

We give our algorithms in an event-based notation and thus assume that a local FIFO event queue is part of the local state of every process. Within an execution step, a process takes an event from the queue, performs a state transition according to the event, and then may send a message or add a new event to the queue. Message arrivals are treated as events too, i.e., when a message arrives, an appropriate event is added to the queue. It is "received" by the process when this event is processed.

In contrast to Chandra and Toueg [3], we assume interrupt-style failure detectors, i.e., whenever the value of $H_{\mathcal{D}}$ changes, a *failure detector event* is added to the local queue of the monitor process. This event contains the ID of the monitored process, whose perceived operational state has changed, and the description of the state change (i.e., whether it is still suspected or not). Consequently, the interrupt model allows processes to make a special step, in which no event is taken from the queue but a failure detector event is added.

2.4 Interrupt-Style *vs.* Query-Style Failure Detectors

In this section we compare the query and the interrupt model for failure detectors and show that they are equivalent if perfect and eventually perfect failure detectors are considered.

We define a *property* of the system as a set of executions. A system *satisfies* a property iff every execution which is possible by the system is an element of the property. If a property P is considered as a problem specification, we say that an algorithm A *solves* problem P iff A satisfies P. More precisely, we say that *algorithm A solves problem P using failure detector $\mathcal{D}$* iff all executions of A using failure detector $\mathcal{D}$ are an element of P. Let $\mathcal{C}$ be a class of failure detectors. We say that *A solves P using $\mathcal{C}$* iff for all $\mathcal{D} \in \mathcal{C}$, A solves P using $\mathcal{D}$. Finally, we say that *problem P can be solved using $\mathcal{C}$* iff for all failure detectors $\mathcal{D} \in \mathcal{C}$ exists an algorithm A such that A solves P using $\mathcal{D}$.

Intuitively, the interrupt model is at least as strong as the query model, as it makes the additional assumption that a process is notified of all perception changes of the failure detector (see Section 2.3). Consequently, correct algorithms in the query model can be easily adapted to the interrupt model.

Theorem 1 *If problem P can be solved using failure detector class $\mathcal{C}$ in the query model, then P can be solved using $\mathcal{C}$ in the interrupt model.*

PROOF SKETCH: An algorithm A' which solves P in the query model can be transformed into an algorithm A for the interrupt model as follows: Whenever a failure detection event is processed from the local event queue, A manipulates a local suspicion list. The algorithm of A' can be incorporated into A by changing the commands which query the failure detector to instead query the current value of the suspicion list.

The transition from the interrupt model to the query model is more difficult. Interestingly, we can show that both models allow to solve the same classes of problems if perfect or eventually perfect failure detectors are used.

To see this, consider an algorithm A which solves a problem P in the interrupt model. The algorithm is notified about *every* state change of the failure detector module. Now consider the same algorithm running in the query model. In the query model, the failure detector is passive and does not issue events. The closest we can come to the interrupt model is to add a converter task to A which queries the failure detector as often as possible and adds failure detection events to the local queue whenever it perceives a state change of the failure detector. In the case of perfect failure detectors, the transition obviously works fine: A perfect failure detector changes its state at most once per application process. Whenever this occurs, A will be notified of this state change as soon as the converter task is scheduled again. While this may happen "later" as in the interrupt model, there is no difference in A's perception of the failure detector. A perfect failure detector in the query model "looks the same" in the interrupt model (see Figure 1).

The same arguments also apply to eventually perfect failure detectors. The only difference is that due to unfortunate scheduling of the converter task, false suspicions may go unnoticed. Indeed, assume that failure detector module $\mathcal{D}_j$ on monitor b_j suspects process p_i, but later does not suspect p_i any more. If in the query model the application running on b_j has never queried the failure detector in the meantime, it is not aware of the (erroneous) suspicion of p_i. Hence, while the detection latency may increase, the number of false detections may actually decrease.

Theorem 2 *Let $\mathcal{C}$ be either the class $\diamond\mathcal{P}$ of eventually perfect or the class $\mathcal{P}$ of perfect failure detectors. Then the following holds: If problem P can be solved using failure detector class $\mathcal{C}$ in the interrupt model, then P can be solved using $\mathcal{C}$ in the query model.*

Notes on proof style. Proofs are written in a structured style similar to proof trees of interactive theorem proving environments. This approach is advocated by Lamport who promises that this style "makes it much harder to prove things that are not true" [14]. The proof is a sequence of numbered proof steps at different levels. Every proof step has a proof which may be refined at lower levels by additional proof steps. For example, proof step $\langle 1 \rangle 2$ is the second proof step on level 1. Proofs may also be read in a structured way, for example, by reading only the top level proof steps and going into sublevels only when necessary.

PROOF SKETCH: We start off with an algorithm A solving P in the interrupt model. An algorithm A' for the query model is constructed as follows: The converter task (see Figure 2) is added to A which repeatedly queries the failure detector module and inserts events into the local queue if the output

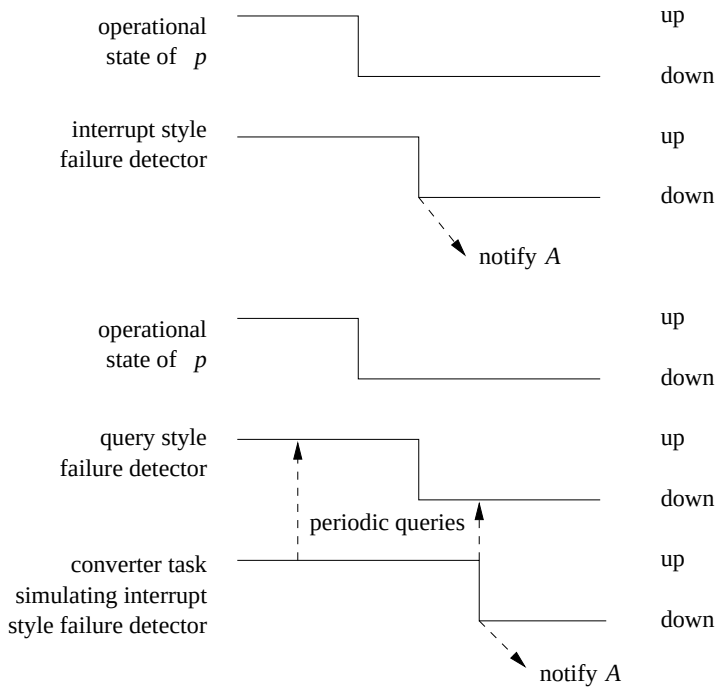

Figure 1: Transition from the interrupt model to the query model of failure detectors.

At monitor b, run algorithm A concurrently to the following task:

```
1   wasSuspected := false
2   loop forever
3     suspected := ⟨query failure detector for p_i⟩
4     if ⟨suspected ≠ wasSuspected⟩ then
5       if ⟨¬wasSuspected⟩ then
6         ⟨trigger event "D suspects process p_i"⟩
7         wasSuspected := true
8       else
9         ⟨trigger event "D rehabilitates process p_i"⟩
10        wasSuspected := false
11      endif
12    endif
13    wait Δt
14  end loop
```

Figure 2: *Converter task* which generates interrupt-based output from query-based perfect and eventually perfect failure detectors concerning the operational state of a process p_i.

(i.e., suspected or not suspected) of the failure detector has changed since the last query. We need to show that A' solves P in the query model if either perfect or eventually perfect failure detectors are used.

The idea of the proof is to take an arbitrary run R' of A' in the query model and show that there is a run R of A in the interrupt model which from the point of view of the algorithm is indistinguishable (i.e., it has the same failure pattern and the same sequence of steps). Since A solves P in the interrupt model, $R \in P$. But because the algorithm is the same and R' is indistinguishable from R, $R' \in P$. Hence, every run R' of A' satisfies P and so A' solves P.

1 $\langle 1 \rangle 1$. There exists an algorithm A that solves P using $\mathcal{C}$ in the interrupt model.

PROOF: From antecedent of the theorem and the definition of "can be solved". □

2 $\langle 1 \rangle 2$. Construct A' by adding the task of Figure 2 to A. A' solves P using $\mathcal{C}$ in the query model.

PROOF SKETCH: Algorithm A was built for the interrupt model and now receives input from the concurrent task. Algorithm A and the task together form the algorithm A' running in the query model. We have to prove that any run of A' satisfies P.

The proof proceeds in three proof steps. In proof step $\langle 2 \rangle 1$, we take an arbitrary run R' of A' in the query model and construct a run R in the interrupt model that is "indistinguishable" to A. The only difference between R and R' is the failure detector history. It is also shown that the failure detector history in R belongs to the same failure detector class as the history in R'.

In proof step $\langle 2 \rangle 2$ we assert that the newly constructed run R satisfies P which follows from proof step $\langle 1 \rangle 1$. This is used by proof step $\langle 2 \rangle 3$ to assert that R' satisfies P because R and R' are indistinguishable to A. Since R' satisfies P and we have not restricted R', A' satisfies P.

2.1 $\langle 2 \rangle 1$. Take an arbitrary run $R' = (F', H'_{\mathcal{D}}, I', S', T')$ of A' in the query model. Then there exists a run $R = (F, H_{\mathcal{D}}, I, S, T)$ of A in the interrupt model such that:

1. $F = F'$
2. $H_{\mathcal{D}} \in \mathcal{D}(F')$
3. $S = S'$ without steps of the concurrent task.
4. $T = T'$ without the time references from steps of the concurrent task.

PROOF SKETCH: We only prove this proof step for the case $\mathcal{C} = \Diamond\mathcal{P}$ since the proof for perfect failure detectors is a special case.

Among the steps taken in R', only a subset of steps contains an access to the failure detector for information about the operational state of process p. This access is performed by the converter task and results[1] in the addition of a new event into the local event queue of A. Consider an arbitrary such step s'. The output of the failure detector has changed time $\Delta t'$ before the converter task has queried the failure detector. In the interrupt model, changes in the output of failure detectors are immediately added to the local event queue. Hence, from $H'_{\mathcal{D}}$ we construct $H_{\mathcal{D}}$ by "delaying" this state change in $H'_{\mathcal{D}}$ for $\Delta t'$. Applying a delay to all state changes in $H'_{\mathcal{D}}$ results in $H_{\mathcal{D}}$, which causes failure detection events for A in the interrupt model at exactly those points in time as A would experience them through the converter task in the query model. Proof step $\langle 3 \rangle 1$ shows how to calculate $\Delta t'$.

The following proof steps show that the resulting failure detector history $H_{\mathcal{D}}$ belongs to the same failure detector class as $H'_{\mathcal{D}}$. This should be clear from the following observation: The transformation may omit state changes in $H'_{\mathcal{D}}$ which are never witnessed by A in the query model. Reducing false suspicions, however, is usually desirable and does not violate the definition of $\Diamond\mathcal{P}$.

2.1.1 $\langle 3 \rangle 1$. Consider the sequence of steps S' of R' and construct a sequence of steps S'_p in which only steps are taken into account where the converter task issues failure detection events (i.e., A' queries the failure detector for process p and receives "new" information about p).

Let s'_i be an arbitrary step in S'_p executed at time t'_i. Then the value of $H'_{\mathcal{D}}$ regarding p changed at some time $t'_i - \Delta t'_i$.

PROOF: Follows from the code of the converter task. □

2.1.2 $\langle 3 \rangle 2$. Define $H_{\mathcal{D}}$ as $H'_{\mathcal{D}}$ in which the changes are delayed by $\Delta t'_i$ (see Figure 3). Now $H_{\mathcal{D}} \in \mathcal{D}(F)$.

PROOF: By definition, correct processes take infinitely many steps. This implies that the converter task queries the failure detector infinitely often. Hence, strong completeness is achieved in $H_{\mathcal{D}}$. The proof step follows from $\langle 3 \rangle 1$ and the definition of the class $\Diamond\mathcal{P}$. □

[1] If the output of the failure detector has changed since the last query.

2.1.3 $\langle 3 \rangle 3$. R is a run of A in the interrupt model.
PROOF: Follows from the fact that steps of the concurrent task are excluded in S and T, and proof step $\langle 3 \rangle 2$. □

2.1.4 $\langle 3 \rangle 4$. Q.E.D.
PROOF: Since p was an arbitrary faulty process, step $\langle 3 \rangle 1$ can be done for all faulty processes.

2.2 $\langle 2 \rangle 2$. R satisfies P.
PROOF: From proof step $\langle 2 \rangle 1$ and the fact that A solves P using $\mathcal{C}$ (proof step $\langle 1 \rangle 1$). □

2.3 $\langle 2 \rangle 3$. R' satisfies P.
PROOF: Proof step $\langle 2 \rangle 1$ proves that with respect to A, R' is indistinguishable to R (i.e., A receives input events at exactly the same points in time). So A must have the same behavior in R' as it has in R. The proof step then follows from proof step $\langle 2 \rangle 2$. □

2.4 $\langle 2 \rangle 4$. Q.E.D.
PROOF: Follows from the fact that we have not restricted R', i.e., the statement holds for every run of A' in the query model. □

3 $\langle 1 \rangle 3$. Q.E.D.
PROOF: Directly from proof step $\langle 1 \rangle 2$ and the definition of "can be solved". □

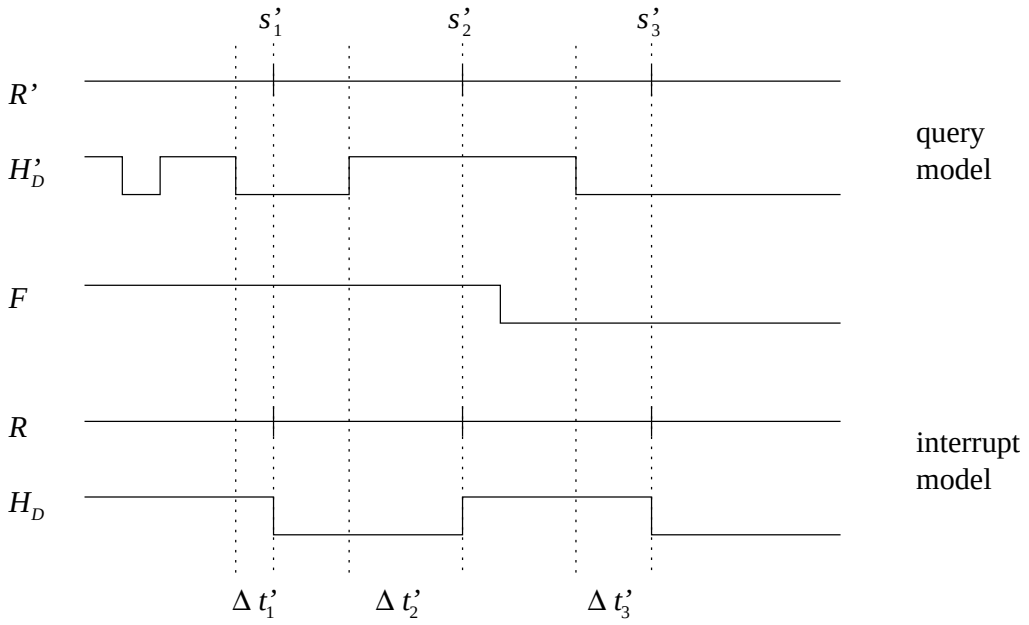

Figure 3: For eventually perfect failure detectors the failure detector history $H'_{\mathcal{D}}$ in the query model can be transformed into an indistinguishable failure detector history $H_{\mathcal{D}}$ in the interrupt model.

Unfortunately, Theorem 2 cannot be extended to failure detector class $\Box\Diamond\mathcal{P}$. To understand the intuitive reason for this, consider the following scenario: A monitor b wants to observe the operational state of an application process p. The desired detection property is that if p is correct and b has noticed that p is down, it must eventually refute this notice and state that p is up again. Detecting this in the interrupt model using any $\mathcal{D} \in \Box\Diamond\mathcal{P}$ is rather simple as the failure detector will not permanently suspect p (this is guaranteed by the infinitely often accuracy property) and the change of state will be noticed. However, in the query model an algorithm may always query the failure detector exactly within those periods where $\mathcal{D}$ is inaccurate and suspects p again.

Theorem 3 *There exists a problem P that can be solved using failure detector class $\Box\Diamond\mathcal{P}$ in the interrupt model, but cannot be solved using $\Box\Diamond\mathcal{P}$ in the query model.*

In the following, when we refer to failure detectors, we always assume the interrupt model, unless explicitly stated otherwise. Since it is equivalent to the query model if perfect or eventually perfect failure detectors are used, the results obtained for these types of failure detectors also hold in the query model. If infinitely often accurate failure detectors are considered, the interrupt model is in a sense "stronger" than the query model. This however means that any impossibility result obtained for these types of failure detectors in the interrupt model directly hold in the query model too. But even for infinitely often accurate failure detectors, algorithms developed in the query model will also work in the interrupt model.

2.5 Comparing Failure Detectors

Intuitively, a failure detector $\mathcal{D}_1$ is *stronger* than a failure detector $\mathcal{D}_2$ (denoted $\mathcal{D}_1 \succeq \mathcal{D}_2$) if there exists a distributed algorithm that can be used to emulate $\mathcal{D}_2$ using $\mathcal{D}_1$ [3]. If $\mathcal{D}_1 \succeq \mathcal{D}_2$ and $\mathcal{D}_2 \succeq \mathcal{D}_1$ we write $\mathcal{D}_1 \cong \mathcal{D}_2$ and say that $\mathcal{D}_1$ and $\mathcal{D}_2$ are *equivalent*. If $\mathcal{D}_1 \succeq \mathcal{D}_2$ but not $\mathcal{D}_2 \succeq \mathcal{D}_1$ we say that $\mathcal{D}_1$ *is strictly stronger than* $\mathcal{D}_2$ and write $\mathcal{D}_1 \succ \mathcal{D}_2$. The relation $\succeq$ can be defined for failure detector classes in an analogous way. From the literature [3, 11] we know that the following relations hold: $\mathcal{P} \succ \Diamond\mathcal{P} \succ \Box\Diamond\mathcal{P}$.

3 Predicate Detection

Every application process p_i has a local state s_i consisting of an assignment of values to all of its variables. A *global state* $G = \{s_1, \ldots, s_n\}$ is a set containing exactly one local state s_i from every application process p_i. State changes are assumed to be atomic events local to some process. For an observer of the application system, a computation should be observed as a sequence of global states $G_1, G_2, \ldots$, where G_{i+1} results from G_i by executing a local event on some process. Global state G_1 denotes the initial global state of this system.

Given some global predicate φ over the global state G of $p_1, \ldots, p_n$, we would like to have an algorithm which answers the question of whether or not φ holds in a given computation. Whenever an event occurs at some application process p_k, a *control message* about this event is sent from p_k to all the monitors (the normal computation messages are called *application messages*) (see Figure 4). One generally assumes that the execution of the event and the sending of the control message execute as one atomic action.

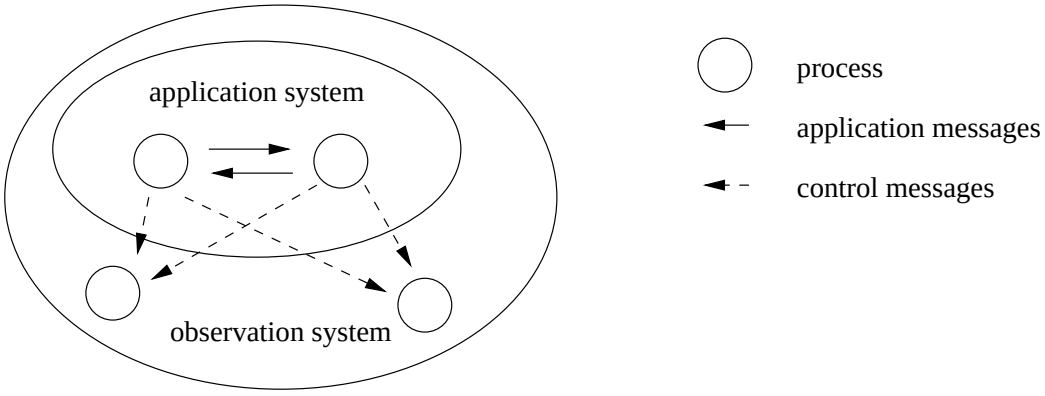

Figure 4: Observation and application system. Application processes exchange application messages, whereas control messages are sent from application to monitor processes.

Every property (i.e., every set of executions) can be written as the intersection of a *safety property*

and a *liveness property* [1, 13]. Informally, a safety property states that "something bad never happens", i.e., it rules out a set of execution prefixes which are not allowed to occur. Mutual exclusion is an example of a safety property since it rules out all executions which end with a state where two processes are in their critical sections at the same time. On the other hand, a liveness property states that "something good will eventually happen", i.e., it requires that the suffix of every execution be of a certain form. Termination is the standard example of a liveness property because it mandates that every execution contains a termination state. We will use safety and liveness properties to define different classes of predicate detection semantics in the next section.

3.1 Three Different Detection Semantics

We use the symbols $\Box$ ("always") and $\Diamond$ ("eventually") here in the following way: Let S be a safety property. We denote by $\Diamond S$ the property in which S eventually holds, i.e., the set in which every execution in S can be prefixed by an arbitrary but finite sequence of states. We denote by $\Box\Diamond S$ the property where S holds infinitely often, i.e., the set consisting of all traces which can be constructed from (infinitely) interleaving finite sequences from S and $\Diamond S$. Note that $S \subseteq \Diamond S \subseteq \Box\Diamond S$.

A detection algorithm for a global predicate φ should notify us by triggering a detection event on the monitor processes once φ holds in the computation. Formally, we seek an algorithm which satisfies the following two properties:

S (safety) If a monitor triggers a detection event then φ has held in the computation, and

L (liveness) once φ holds in the computation, a monitor will eventually trigger a detection event.

We assume that the algorithm triggers a positive signal on detection but we also allow the algorithm to revoke its detection by issuing a "previous detection was wrong" signal to the application. If this occurs, we say that the algorithm *undetects* the predicate. Note that we detect whether the predicate φ *held*, i.e., we detect a stable predicate even if φ is unstable. A predicate is *stable*, if once it holds it holds forever.

Definition 1 (detection semantics) *Let S and L denote the safety and liveness properties of predicate detection. We define the three detection semantics Sem_1, Sem_2, and Sem_3 as follows (where $+$ denotes set intersection):*

1. $Sem_1 = L + S$ *(perfect)*
2. $Sem_2 = L + \Diamond S$ *(stabilizing)*
3. $Sem_3 = L + \Box\Diamond S$ *(infinitely often accurate)*

To illustrate the use of these detection semantics, assume that φ is a debugging condition, i.e., a "bad state" which should not occur. On detection of such a state, the software developer usually wants to stop the application system and analyze the execution which caused the bad state. In this context, we would ideally like a detection algorithm *Alg* for predicate φ to satisfy Sem_1, i.e., *Alg* will make no mistakes and not miss any occurrence of φ. We call this *perfect* predicate detection. However, this is sometimes impossible to achieve. In particular, if φ contains conditions about the operational state of processes, the detection algorithm might mistakenly detect φ, i.e., violate S. In these cases *Alg* should at least satisfy Sem_2, i.e., *Alg* may (erroneously) detect the predicate and later undetect it again. We call this *stabilizing* predicate detection because it is guaranteed that the algorithm will eventually stop making wrong detections.

Note that from the user's point of view there is no immediate way to distinguish between a correct and an incorrect detection (this may only be achieved through other means, e.g., through halting the

system and inspecting it). Stabilizing predicate detection may, however, still be useful, e.g., in situations where false detections merely effect the *efficiency* of an application (not its correctness) or in situations where achieving Sem_1 is provably impossible. Revisiting our debugging example the developer may want to detect a predicate φ in his distributed application in order to identify an invalid state of the application. Detection semantics Sem_2 are sufficient in most cases, as the developer can manually verify whether the predicate detection algorithm has been accurate. If it has not been, the predicate detection is continued.

But even Sem_2 is sometimes impossible to satisfy, i.e., *Alg* may make *infinitely* many mistakes about φ holding. In this case we would prefer that *Alg* behaves according to Sem_3, i.e., *Alg* continuously switches between phases where possible detections are accurate and phases where mistakes regarding φ are made. We call this *infinitely often accurate* predicate detection. This means that if φ never holds then every detection event will be followed by an undetection event. Semantics Sem_3 offer close to no guarantees and, hence, can be considered as a "best effort" specification. But at least it is better than ignoring the safety part of the specification overall (i.e., we rule out trivial predicate detection which always issues a detection event even in cases where φ never holds). Note that $Sem_1 \subseteq Sem_2 \subseteq Sem_3$.

The detection algorithms we study in the following sections all have a common architecture (see Figure 5). In particular they rely on the existence of a "low level" failure detection service which satisfies certain properties, e.g., those of $\diamond\mathcal{P}$ or $\mathcal{P}$. The goal of the algorithms is to provide the best possible detection semantics,

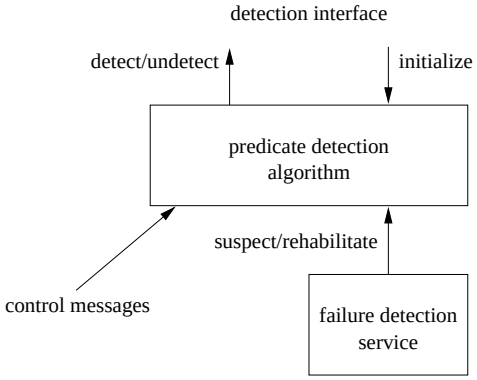

Figure 5: Architecture of the predicate detection algorithms at monitor m_i.

3.2 Global Predicates in Faulty Systems

In systems with crash faults, it is a natural desire to detect a class of predicates which makes no sense in fault-free systems because we now additionally want to detect predicates involving the operational state of processes. For example, we may want to detect the fact that "process p_i crashed while holding a lock." To express this information within a global predicate we assume that the operational state of a process is explicitly modeled by a boolean flag *crashed* which is part of the global state. More specifically, $crashed_i$ is true iff p_i has crashed. Using this flag, we formalize the above predicate as $lock_i = true \wedge crashed_i$.

With respect to the truth value of a predicate on the global state, the *crashed* variables are treated just like other variables local to the processes. Let α denote a local predicate which only references local variables of a process, e.g., $\alpha \equiv x_i = 1$, and let β denote a predicate which only contains references to the operational state of a process, e.g., $\beta \equiv crashed_i$. To detect α we can use a standard mechanism for predicate detection in fault free systems. Conversely, to detect β we can use

a (reliable) failure detection algorithm. However, global predicates can be constructed from boolean combinations of α and β. If we have disjunctions of α and β, e.g., $x_i = 1 \vee \mathit{crashed}_i$, it is sufficient to run existing detection algorithms for α and β independently and issue a detection event as soon as one of the algorithms issues such an event. However, global predicates that are a conjunction of α and β are more difficult to detect and are the focus of this paper.

The following examples illustrate the types of predicates we address. Let ec_i denote a local variable of p_i which stores the sequence number of events ("event count") on p_i.

- $ec_i \geq 5 \wedge \mathit{crashed}_i$, i.e., "$p_i$ crashed after event 5"
- $ec_i = 5 \wedge \mathit{crashed}_i$, i.e., "$p_i$ crashed immediately after event 5"
- $ec_i < 5 \wedge \mathit{crashed}_i$, i.e., "$p_i$ crashed before reaching event 5"
- $ec_i = 5 \wedge \neg \mathit{crashed}_i$, i.e., "$p_i$ executed event 5".

Although we have not restricted the class of predicates, it should be noted that only predicates are detectable that do not explicitly depend on global time. For instance, the predicate $\varphi \equiv$ "p_i executed event e_i more than 10 seconds ago" is impossible to detect in an asynchronous system model where no notion of global time exists. In analogy to Charron-Bost et al. [6], we call predicates that do not refer to real time *time free*.

4 Predicate Detection in Faulty Systems

We now consider an asynchronous system where crash faults can happen and study what types of detection semantics (i.e., Sem_1, Sem_2, or Sem_3) are achievable using different classes of failure detectors. For simplicity, we will restrict our attention to the case where $n = m = 1$, i.e., a system with two processes only, namely an application process and a monitor process. We discuss the case where $n, m > 1$ in Section 5.

If faults can happen, the predicate detection algorithm must cater for the fact that a failure detector issues a suspicion or rehabilitation of a process. A process is *rehabilitated* if it has been erroneously suspected and the failure detector revokes its suspicion. Figure 6 shows a generic detection algorithm for this case. A boolean flag h ("history") is used to record the type of the most recent event which was triggered at the interface.

4.1 Plausible Failure Detector

Generally, the failure detector module accessed by the monitor issues suspicion and rehabilitation events for a process p_i. We define the following two properties concerning these events:

- (alternation) Suspicion and rehabilitation events for p_i alternate, i.e., the failure detector module never issues two suspicion events without issuing a rehabilitation event inbetween, and vice versa.
- (plausibility) Reception of a control message from p_i is only possible in phases where p_i is not suspected.

We argue later that these properties are fundamental to the predicate detection problem. Based on these properties, we define a plausible failure detector as follows:

Definition 2 (plausible failure detector) *A* plausible failure detector *is a failure detector which satisfies alternation and plausibility.*

On every monitor b:

variables:
$G : (s_1, \ldots, s_n)$ **init** $(I_1, \ldots, I_n)$
crashed$[1..n]$ **array of** $\{$*true*,*false*$\}$ **init** (*false*, ..., *false*)
$\varphi : G \times$ *crashed* $\rightarrow \{$*true*, *false*$\}$ **init** ⟨by application⟩
$h : \{$*true*, *false*$\}$ **init** *false*

1 **algorithm:**
2 **upon** ⟨a message (i, e) arrives⟩ **do**
3 ⟨update $G[i]$ according to e⟩
4 **if** $\varphi(G,$ *crashed*$) \wedge \neg h$ **then**
5 $h :=$ *true*
6 ⟨trigger detection event⟩
7 **elsif** $\neg\varphi(G,$ *crashed*$) \wedge h$ **then**
8 $h :=$ *false*
9 ⟨trigger undetection event⟩
10 **end**
11 **upon** ⟨p_i is suspected
12 or rehabilitated by failure detector⟩ **do**
13 ⟨update *crashed*$[i]$ accordingly⟩
14 **if** $\varphi(G,$ *crashed*$) \wedge \neg h$ **then**
15 $h :=$ *true*
16 ⟨trigger detection event⟩
17 **elsif** $\neg\varphi(G,$ *crashed*$) \wedge h$ **then**
18 $h :=$ *false*
19 ⟨trigger undetection event⟩
20 **end**

Figure 6: Generic algorithm for predicate detection in faulty environments.

On every monitor b:

variables:
suspects **set of** ⟨processes⟩ **init** $\emptyset$
wasSuspected : $\{true, false\}$ **init** *false*

```
1   algorithm:
2     upon ⟨D rehabilitates p_i
3           or a control message from p_i arrives⟩ do
4       wasSuspected := false
5       if p_i ∈ suspects then
6         wasSuspected := true
7         suspects := suspects \ {p_i}
8         ⟨trigger event "rehabilitation of p_i"⟩
9       endif
10      if ⟨control message was received in line 2⟩ then
11        ⟨deliver control message ⟩
12        if wasSuspected then
13          suspects := suspects ∪ {p_i}
14          ⟨trigger event "suspicion of p_i"⟩
15        endif
16      endif
17    upon ⟨D suspects p_i⟩ do
18      if p_i ∉ suspects then
19        suspects := suspects ∪ {p_i}
20        ⟨trigger event "suspicion of p_i"⟩
21      endif
```

Figure 7: Implementation of a wrapper that makes any failure detector $\mathcal{D}$ in $\Box\Diamond\mathcal{P}$ or $\Diamond\mathcal{P}$ plausible in asynchronous systems. Note that line 2 and 17 refer to events generated by $\mathcal{D}$ while in lines 8, 11, 14, and 20 events at the interface of the plausible failure detector are triggered which are then processed at lines 2 and 11 in the algorithm of Figure 6.

Figure 7 gives the pseudocode that converts failure detectors in $\Box\Diamond\mathcal{P}$ or $\Diamond\mathcal{P}$ into their plausible equivalent. For this purpose, the failure detector also takes into account the control messages of the processes. Indeed, if a control message from process p_i is received while the failure detector suspects p_i, then p_i is rehabilitated. However, this requires that the delivery of control messages is included into the wrapper which constitutes the plausible failure detector (see Figure 8).

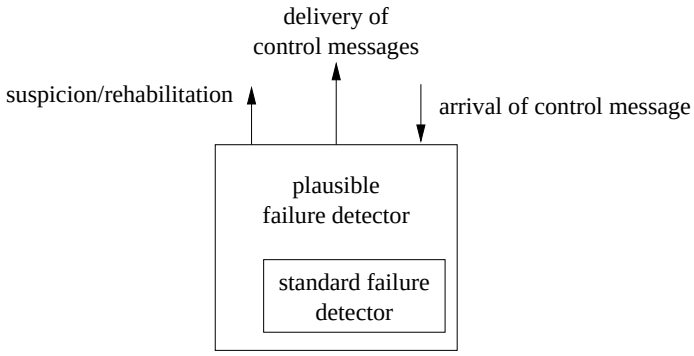

Figure 8: Interface of a plausible failure detector.

As shown in the following lemma, for most failure detector classes a plausible failure detector still belongs to the same class of failure detectors as its non-plausible equivalent.

Lemma 1 *Let F be either failure detector class $\Box\Diamond\mathcal{P}$ or $\Diamond\mathcal{P}$ and let $\bar{F}$ denote the set of failure detectors from F which are plausible. Then $F \cong \bar{F}$.*

PROOF SKETCH: A failure detector is rendered plausible by wrapping the failure detector and the delivery module for control messages into a separate module. If a control message arrives after a suspicion event has been generated, a rehabilitation event is passed to the detection algorithm before delivering the control message. Clearly, the wrapper can be implemented in asynchronous systems.
ASSUME: Let F denote either $\Box\Diamond\mathcal{P}$ or $\Diamond\mathcal{P}$ and let $\mathcal{D}$ be a failure detector from F.
PROVE: The algorithm in Figure 7 issues events that satisfy the requirements of $\bar{F}$.

1 $\langle 1\rangle 1$. The algorithm satisfies strong completeness.
1.1 $\langle 2\rangle 1$. ASSUME: p_i crashes.
PROVE: An event "suspicion of p_i" is eventually triggered.
1.1.1 $\langle 3\rangle 1$. Eventually $\mathcal{D}$ suspects p_i.
PROOF: From the fact that $\mathcal{D}$ satisfies strong completeness. □
1.1.2 $\langle 3\rangle 2$. CASE: $p_i \notin$ *suspects*
PROOF: Directly from the algorithm (lines 17–21). □
1.1.3 $\langle 3\rangle 3$. CASE: $p_i \in$ *suspects*
PROOF: Since *suspects* $= \emptyset$ initially and the only place where p_i can be added to *suspects* is line 13 or 19 of the algorithm it follows, that the algorithm must previously have issued a suspicion event in line 14 or 20. □
1.1.4 $\langle 3\rangle 4$. Q.E.D.
PROOF: Steps $\langle 3\rangle 2$ and $\langle 3\rangle 3$ cover all cases. □
1.2 $\langle 2\rangle 2$. Q.E.D.
PROOF: From step $\langle 2\rangle 1$. □
2 $\langle 1\rangle 2$. The algorithm satisfies the accuracy requirement of F.
2.1 $\langle 2\rangle 1$. CASE: $F = \Box\Diamond\mathcal{P}$
PROOF SKETCH: We must show that a non-crashed process is not suspected permanently.

2.1.1 $\langle 3 \rangle 1$. ASSUME: p_i is correct and there is a time t where p_i is suspected in line 14 or 20 but never rehabilitated in line 8 after t.
PROVE: false
PROOF: From the infinitely often accuracy property of $\mathcal{D}$ and since p_i is correct, eventually p_i is rehabilitated by $\mathcal{D}$ in line 2. Since p_i has been previously suspected, it is an element of *suspects* and hence the algorithm issues a rehabilitation event in line 8, a contradiction. □

2.1.2 $\langle 3 \rangle 2$. Q.E.D.
PROOF: Indirectly from step $\langle 3 \rangle 1$. □

2.2 $\langle 2 \rangle 2$. CASE: $F = \Diamond \mathcal{P}$
PROOF SKETCH: We must show that there is a time after which a correct process p_i is not wrongly suspected in line 14 or 20.

2.2.1 $\langle 3 \rangle 1$. There is a time t after which $\mathcal{D}$ does not suspect a correct process anymore, i.e., after t, process p_i has been rehabilitated in line 2 and is never suspected in line 17.
PROOF: From the eventual strong accuracy of $\mathcal{D}$. □

2.2.2 $\langle 3 \rangle 2$. There are at most finitely many control messages in transit from p_i to the monitor at time t.
PROOF: From the fact that p_i has only run for a finite time. □

2.2.3 $\langle 3 \rangle 3$. Consider the arrival of the first control message m after time t. Following this arrival lines 8 and 14 are *not* executed.
PROOF: From step $\langle 3 \rangle 1$ follows that $p_i \notin$ *suspects* when m arrives. Hence, line 8 is not executed. From the use of the *wasSuspected* flag follows that the condition in line 12 is false and so line 14 is not executed either. □

2.2.4 $\langle 3 \rangle 4$. Q.E.D.
PROOF: From step $\langle 3 \rangle 2$ and repeatedly applying step $\langle 3 \rangle 3$ we can show that after t, p_i is not suspected anymore in line 14. □

2.3 $\langle 2 \rangle 3$. Q.E.D.
PROOF: Steps $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$ cover all cases. □

3 $\langle 1 \rangle 3$. The algorithm satisfies alternation.
PROOF SKETCH: We prove this claim only for the case where a suspicion is followed by a rehabilitation. The converse is analogous.

3.1 $\langle 2 \rangle 1$. ASSUME: e is a suspicion event of the algorithm and e' is the following event issued by the algorithm.
PROVE: e' is a rehabilitation event.
PROOF: Since e is a suspicion event, it was generated in line 14 or 20 of the algorithm. This means that $p_i \in$ *suspects* and another suspicion event is only generated if the opposite is true. This can be only performed by executing line 7. This implies execution of line 8 and thus triggering a rehabilitation event. Hence, e' is a rehabilitation event. □

3.2 $\langle 2 \rangle 2$. Q.E.D.
PROOF: From step $\langle 2 \rangle 1$. □

4 $\langle 1 \rangle 4$. The algorithm satisfies plausibility.

4.1 $\langle 2 \rangle 1$. ASSUME: Line 11 is executed.
PROVE: If there has been at least one event issued, then the most recent event issued was a rehabilitation event.
PROOF: Assume there has been at least one event issued. If this was a suspicion event, then at line 2 $p_i \in$ *suspects*, otherwise $p_i \notin$ *suspects*. In the latter case we have shown the claim already. In the former case the algorithm issues a rehabilitation event in line 8 prior to delivering the control message. □

4.2 $\langle 2 \rangle 2$. Q.E.D.
PROOF: From step $\langle 2 \rangle 1$. □

5 $\langle 1 \rangle 5$. Q.E.D.
PROOF: From steps $\langle 1 \rangle 1$ to $\langle 1 \rangle 4$ and Definition 2. □

Transforming a failure detector $\mathcal{D}$ in $\mathcal{P}$ into a plausible failure detector using the algorithm in Figure 7 may result in a weaker failure detector. Indeed, assume a system with one process p and one monitor b, where p has sent a control message *msg* and then crashes. Before b receives the control message, it detects the crash of p. On reception of *msg*, the plausible version of $\mathcal{D}$ rehabilitates p in order to receive *msg*. Later, $\mathcal{D}$ eventually suspects p again. However, a failure detector in $\mathcal{P}$ is not allowed to make mistakes, i.e., can never rehabilitate processes. Hence, if $\mathcal{D}$ is made plausible with algorithm in Figure 7, it is no longer in class $\mathcal{P}$. This is the reason why Lemma 1—using this particular wrapper—does not hold for failure detectors in $\mathcal{P}$.

4.2 Using an Infinitely Often Accurate Failure Detector ($\Box\Diamond\mathcal{P}$)

Consider a purely asynchronous system in which crash faults can happen. Garg and Mitchell [11] have shown that a failure detector in $\Box\Diamond\mathcal{P}$ can be implemented in these systems, e.g., an *infinitely often accurate failure detector*. Additionally we assume that such a failure detector is plausible. In such cases predicate detection is only achievable with semantics Sem_3.

Theorem 4 *In asynchronous systems with crash failures and any failure detector in $\Box\Diamond\mathcal{P}$ it is (a) possible to satisfy detection semantics Sem_3 but it is (b) impossible to satisfy detection semantics Sem_2 and Sem_1 for general predicates without a failure detector strictly stronger than $\Box\Diamond\mathcal{P}$.*

PROOF SKETCH: We prove (a) using the standard algorithm in Figure 6. The reliable channel assumption ensures satisfaction of the liveness requirement of Sem_3 and the plausible infinitely often accuracy of failure detection ensures the safety requirement of Sem_3. Part (b) is proven indirectly: We assume that an algorithm satisfying Sem_2 exists and use it to construct a failure detector that allows to solve consensus, a contradiction to the impossibility result by Fischer, Lynch and Paterson [9].

ASSUME: The system model is deterministic (i.e., allows no randomization), asynchronous with crash failures and any failure detector in $\Box\Diamond\mathcal{P}$.

PROVE: It is (a) possible to satisfy detection semantics Sem_3 but it is (b) impossible to satisfy detection semantics Sem_2 and Sem_1 for general predicates.

1 $\langle 1 \rangle 1$. A plausible failure detector in $\Box\Diamond\mathcal{P}$ is available.
PROOF: Follows from Lemma 1. □

2 $\langle 1 \rangle 2$. General predicates are detectable with semantics Sem_3 in the given system model.

2.1 $\langle 2 \rangle 1$. The algorithm in Figure 6 satisfies the liveness requirement L of the predicate detection problem.
ASSUME: φ holds within the computation.
PROVE: The algorithm eventually triggers a detection event in line 6 or 16.

2.1.1 $\langle 3 \rangle 1$. Without loss of generality φ has the form $\alpha \wedge \beta$ where α refers to the local state of p_i and β refers to the operational state of p_i. p_i sends control messages to the monitor about the state sequence resulting in state α.
PROOF: From the assumption that φ holds and the algorithm. □

2.1.2 $\langle 3 \rangle 2$. Eventually, these messages arrive at the monitor and the monitor constructs a sequence of global states resulting in a state where α holds.
PROOF: From step $\langle 3 \rangle 1$, the FIFO reliable channel assumption and the algorithm. □

2.1.3 $\langle 3 \rangle 3$. CASE: β has the form $crashed_i$.

2.1.3.1 $\langle 4 \rangle 1$. p_i crashed in a state where α held.
PROOF: From assumption that φ holds in the computation. □

2.1.3.2 $\langle 4 \rangle 2$. p_i is eventually suspected after the final control message from p_i arrived.
PROOF: From step $\langle 4 \rangle 1$, strong completeness and plausibility of the failure detector (step $\langle 1 \rangle 1$), and the algorithm. □

2.1.3.3 $\langle 4 \rangle 3$. Q.E.D.
PROOF: From steps $\langle 3 \rangle 2$ and $\langle 4 \rangle 2$ the algorithm will trigger a detection event in line 16. □

2.1.4 $\langle 3 \rangle 4$. CASE: β has the form $\neg \mathit{crashed}_i$.
PROOF: Since the failure detector satisfied plausibility, application events are only processed when p_i is supposed to be up. Since the algorithm has already constructed a global state in which α holds (step $\langle 3 \rangle 2$), the final control message to construct alpha arrives in a state where $\alpha \wedge \neg \mathit{crashed}_i \equiv \alpha \wedge \beta$ holds. Therefore, the algorithm triggers a detection event in line 6. □

2.1.5 $\langle 3 \rangle 5$. Q.E.D.
PROOF: Steps $\langle 3 \rangle 3$ and $\langle 3 \rangle 4$ cover all cases. □

2.2 $\langle 2 \rangle 2$. The algorithm in Figure 6 satisfies the safety requirement $\Box\Diamond S$ of the predicate detection problem.
PROOF SKETCH: We assume that φ never holds and t is a point in time of an execution. We must show that there is a time $t' \geq t$ such that the algorithm does not indicate that φ holds. Basically, there are three cases to consider: (1) The algorithm has never issued a detection event before t, (2) the most recent event issued by the algorithm was an undetection event, or (3) the most recent event was a detection event. Cases (1) and (2) are trivial. In the third case, obviously the algorithm has made a wrong failure detection which can only be due to an up process being suspected to be down (this follows from the plausibility of the failure detector). Since the failure detector is infinitely often accurate, it will eventually rehabilitate that process at some future point in time t' leading to an undetection.

2.2.1 $\langle 3 \rangle 1$. ASSUME: φ never holds in some execution and let t be a point in time of that execution.
PROVE: There is a point in time $t' \geq t$ such that the algorithm indicates that φ does not hold.

2.2.1.1 $\langle 4 \rangle 1$. CASE: The algorithm has never issued any detection or undetection event.
PROOF: Take $t' = t$. □

2.2.1.2 $\langle 4 \rangle 2$. CASE: The most recent event has been an undetection event.
PROOF: Take $t' = t$. □

2.2.1.3 $\langle 4 \rangle 3$. CASE: The most recent event has been a detection event.

2.2.1.3.1 $\langle 5 \rangle 1$. $\varphi(G, \mathit{crashed})$ and h both equal *true* at time t.
PROOF: From the algorithm. □

2.2.1.3.2 $\langle 5 \rangle 2$. The detection event resulted from a wrong suspicion of some process p_i issued by the failure detector in line 11.
PROOF: If the detection event was due to the rehabilitation of some process p_i in line 11 or due to the arrival of a control message in line 2, then φ must have held in the computation, a contradiction to the assumption of step $\langle 3 \rangle 1$. □

2.2.1.3.3 $\langle 5 \rangle 3$. There is a time $t' \geq t$ such that the failure detector rehabilitates p_i.
PROOF: From step $\langle 5 \rangle 2$ and the assumption that φ never holds, we know that p_i is a correct process. The step follows from this and the infinitely often accuracy property of the failure detector. □

2.2.1.3.4 $\langle 5 \rangle 4$. Q.E.D.
PROOF: Directly from step $\langle 5 \rangle 3$. □

2.2.1.4 $\langle 4 \rangle 4$. Q.E.D.
PROOF: Steps $\langle 4 \rangle 1$, $\langle 4 \rangle 2$ and $\langle 4 \rangle 3$ cover all cases. □

2.2.2 $\langle 3 \rangle 2$. Q.E.D.
PROOF: Directly from step $\langle 3 \rangle 1$. □

2.3 $\langle 2 \rangle 3$. Q.E.D.

PROOF: From steps $\langle 2\rangle 1$ and $\langle 2\rangle 2$ we have that the algorithm satisfies Sem_3. □

3 $\langle 1\rangle 3$. It is impossible to detect general predicates according to Sem_2 or Sem_1 in the given system model.

3.1 $\langle 2\rangle 1$. It is impossible according to Sem_2.

3.1.1 $\langle 3\rangle 1$. ASSUME: There exists an algorithm *Alg* which solves the predicate detection problem according to Sem_2 in the given system model.
PROVE: false

3.1.1.1 $\langle 4\rangle 1$. Consider the predicate $\varphi \equiv crashed_i$. Then *Alg* satisfies eventual strong accuracy and strong completeness for p_i.
PROOF: From the assumption and the definition of Sem_2. □

3.1.1.2 $\langle 4\rangle 2$. Consensus is solvable in the given system model.
PROOF: Use the algorithm in step $\langle 4\rangle 1$ to build an eventually perfect failure detector in the given system model and apply the algorithm of Chandra and Toueg [3]. □

3.1.1.3 $\langle 4\rangle 3$. Q.E.D.
PROOF: Step $\langle 4\rangle 2$ is a contradiction to the result of Fischer, Lynch and Paterson [9] (note that this result also holds in the interrupt model). □

3.1.2 $\langle 3\rangle 2$. Q.E.D.
PROOF: Follows indirectly from step $\langle 3\rangle 1$. □

3.2 $\langle 2\rangle 2$. It is impossible according to Sem_1.
PROOF: From step $\langle 2\rangle 1$ and the fact that $Sem_2 \supseteq Sem_1$. □

3.3 $\langle 2\rangle 3$. Q.E.D.
PROOF: From steps $\langle 2\rangle 1$ and $\langle 2\rangle 2$. □

4 $\langle 1\rangle 4$. Q.E.D.
PROOF: Step $\langle 1\rangle 2$ shows claim (a), step $\langle 1\rangle 3$ claim (b). □

4.3 Using an Eventually Perfect Failure Detector ($\Diamond\mathcal{P}$)

Defining stronger limitations on incorrect failure suspicions results in stronger predicate detection semantics. As $\mathcal{P}$ and $\Diamond\mathcal{P}$ are stronger than $\Box\Diamond\mathcal{P}$ [11], Theorem 4 (a) holds also for these failure detectors, i.e., Sem_3 can be achieved. Note that the failure detector must be made plausible before being used in the algorithm of Figure 6.[2]

Corollary 1 *In asynchronous systems with crash failures and any failure detector in $\Diamond\mathcal{P}$ it is possible to satisfy detection semantics Sem_3.*

PROOF SKETCH: Follows from Theorem 4, the fact that $\Diamond\mathcal{P} \succ \Box\Diamond\mathcal{P}$.
ASSUME: A failure detector in $\Diamond\mathcal{P}$ is available.
PROVE: There exists an algorithm that solves the predicate detection problem with Sem_3.

1 $\langle 1\rangle 1$. There exists an algorithm that solves the predicate detection problem using any failure detector in $\Box\Diamond\mathcal{P}$ with semantics Sem_3 in the given system model.
PROOF: From Theorem 4. □

2 $\langle 1\rangle 2$. A failure detector in $\Diamond\mathcal{P}$ satisfies the properties of $\Box\Diamond\mathcal{P}$.
PROOF: Obvious, since $\Diamond\mathcal{P}$ imposes stronger requirements on the behavior of failure detectors than $\Box\Diamond\mathcal{P}$. □

3 $\langle 1\rangle 3$. Q.E.D.
PROOF: From steps $\langle 1\rangle 1$ and $\langle 1\rangle 2$. □

[2] Also note that although making $\mathcal{P}$ plausible with the wrapper in Figure 7 weakens the failure detector to $\Diamond\mathcal{P}$, Theorem 4 (a) still holds.

However, although an eventually perfect failure detector is stronger, it still is not sufficient to detect predicates perfectly. Actually, even a perfect failure detector cannot achieve perfect predicate detection. The intuitive reason for this is depicted in Figure 9. Consider the case where a predicate φ is true iff p_1 crashes after event e_1, i.e., $\varphi \equiv ec_1 = 1 \wedge crashed_1$. After suspecting p_1 (see Figure 9 (b)) the monitor b_1 must eventually raise an exception to the application that the predicate held. However, b_1 can never be sure that the predicate detection is accurate, because a message from p_1 may arrive later informing it about an event e_2 (see Figure 9 (a)). Since the message can be delayed for an arbitrary amount of time, b_1 cannot distinguish between both scenarios.

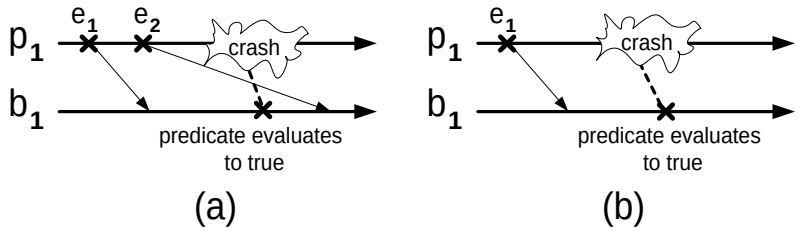

Figure 9: Predicate $\varphi \equiv ec_1 = 1 \wedge crashed_1$ is not detectable according to detection semantics Sem_1 with any failure detector in $\mathcal{P}$.

Theorem 5 *In asynchronous systems with crash failures and any failure detector not strictly stronger than $\mathcal{P}$ it is impossible to satisfy detection semantics Sem_1.*

ASSUME: Perfect failure detectors are available.
PROVE: There exists no algorithm that solves the predicate detection problem with semantics Sem_1 in the given system model.

1 $\langle 1 \rangle 1$. ASSUME: There exists an algorithm *Alg* which solves the predicate detection problem with semantics Sem_1 using perfect failure detectors.
PROVE: false

1.1 $\langle 2 \rangle 1$. Consider an execution σ_1 where an event $x := 1$ happens on a process p at time t, where p crashes after t without executing another event, and where *Alg* is used to detect $\varphi \equiv x = 1 \wedge crashed_p$. (This is visualized in Figure 9 (b) where e_1 denotes the event $x := 1$.) Then there is a point in time t' where *Alg* triggers a detection event.
PROOF: Obviously, φ holds in the execution as soon as p crashed. Since *Alg* satisfies Sem_1, it eventually triggers a detection event. The time t' at which this happens is the witness for the claim. □

1.2 $\langle 2 \rangle 2$. Consider an execution σ_2 which is the same as σ_1 except that p executes another event $x := 2$ after executing $x := 1$ and before crashing. (See Figure 9 (a) where e_2 denotes the additional event $x := 2$.) Furthermore, the control message is delayed in such a way that it is delivered to *Alg* after time t'. Then *Alg* never issues a detection event.
PROOF: Since φ never holds in σ_2 and *Alg* satisfies Sem_1, it never detects φ. □

1.3 $\langle 2 \rangle 3$. *Alg* issues a detection event in σ_2 at time t'.
PROOF: Follows from step $\langle 2 \rangle 1$ and the fact that σ_2 is indistinguishable from σ_1 up to time t' in an asynchronous system. □

1.4 $\langle 2 \rangle 4$. Q.E.D.
PROOF: Step $\langle 2 \rangle 3$ is a contradiction to step $\langle 2 \rangle 2$. □

2 $\langle 1 \rangle 2$. Q.E.D.
PROOF: Follows indirectly from step $\langle 1 \rangle 1$. □

Corollary 2 *In asynchronous systems with crash failures and any failure detector not strictly stronger than $\diamond\mathcal{P}$ it is impossible to satisfy detection semantics Sem_1.*

PROOF SKETCH: Follows from Theorem 5 and the fact that $\mathcal{P} \succ \diamond\mathcal{P}$.

On the other hand, detecting Sem_2 with $\diamond\mathcal{P}$ is achievable. However, the proof again relies on the fact that the given failure detector is plausible. Using a non-plausible failure detector may cause a miss of the occurrence of certain predicates and thus violate the liveness property. Assume, for instance, the predicate $\varphi \equiv x_i = 1 \wedge \neg crashed_i$, with x initially 0. Furthermore, assume that the failure detector $\mathcal{D}$ in $\diamond\mathcal{P}$ is not plausible and that it erroneously suspects process p_i. Although p_i sends the control messages about an event that sets x to 1 and back to 0 again, the monitor does not detect that φ has held.

Theorem 6 *In asynchronous systems with crash failures and any failure detector in $\diamond\mathcal{P}$ it is possible to satisfy detection semantics Sem_2.*

PROOF SKETCH: The proof is similar to the proof of Theorem 4, i.e., we use the algorithm in Figure 6 and show that it satisfies requirements of Sem_2. For simplicity, we only cover the case where the predicate φ which is to be detected has the form $\varphi \equiv \alpha \wedge \beta$ as described above.

1 $\langle 1 \rangle 1$. A plausible failure detector in $\diamond\mathcal{P}$ is available.
PROOF: Follows from Lemma 1. □

2 $\langle 1 \rangle 2$. The algorithm in Figure 6 satisfies L.

2.1 $\langle 2 \rangle 1$. ASSUME: φ holds in the computation.
PROVE: The algorithm issues a detection event.

2.1.1 $\langle 3 \rangle 1$. Eventually all control messages relating to α arrive at the monitor and are delivered in causal order.
PROOF: From the FIFO reliable channel assumption. □

2.1.2 $\langle 3 \rangle 2$. The monitor constructs a global state g where α holds.
PROOF: From step $\langle 3 \rangle 1$ and the algorithm. □

2.1.3 $\langle 3 \rangle 3$. CASE: $\beta \equiv \neg crashed_i$
PROOF: Since the failure detector is plausible (step $\langle 1 \rangle 1$), the final control message of step $\langle 3 \rangle 2$ will arrive while p_i is not suspected, so φ holds in g and the algorithm issues a detection event in line 6. □

2.1.4 $\langle 3 \rangle 4$. CASE: $\beta \equiv crashed_i$
Similar to the argument in Theorem 4 after constructing g and since φ holds in the computation, eventually the failure detector will suspect p_i because it satisfies plausibility and strong completeness. So eventually the algorithm will issue a detection event in line 16. □

2.1.5 $\langle 3 \rangle 5$. Q.E.D.
PROOF: Steps $\langle 3 \rangle 3$ and $\langle 3 \rangle 4$ cover all cases. □

2.2 $\langle 2 \rangle 2$. Q.E.D.
PROOF: From step $\langle 2 \rangle 1$ and the definition of L. □

3 $\langle 1 \rangle 3$. The algorithm in Figure 6 satisfies $\diamond S$.

3.1 $\langle 2 \rangle 1$. There is a time t after which the failure detector makes no wrong detections.
PROOF: From the fact that it satisfies eventual strong accuracy. □

3.2 $\langle 2 \rangle 2$. There is a time $t' \geq t$ after which the plausible version of the failure detector makes no wrong detections and no control messages are received from the suspected process in the future.
PROOF: From step $\langle 2 \rangle 1$ and step $\langle 1 \rangle 1$. □

3.3 $\langle 2 \rangle 3$. ASSUME: The algorithm issues a detection event after t'.
PROVE: φ held in the computation.

3.3.1 $\langle 3\rangle 1$. Control messages concerning α have been received.
PROOF: From assumption that a detection event is issued and the algorithm. □

3.3.2 $\langle 3\rangle 2$. Control messages concerning α have been sent.
PROOF: From step $\langle 3\rangle 1$ and the reliable channel assumption. □

3.3.3 $\langle 3\rangle 3$. α held at some point in the computation.
PROOF: From step $\langle 3\rangle 2$ and the FIFO ordering of control messages. □

3.3.4 $\langle 3\rangle 4$. CASE: $\beta \equiv \neg \mathit{crashed}_i$
PROOF: From step $\langle 3\rangle 3$, the case assumption, and the plausibility of the failure detector (step $\langle 1\rangle 1$) φ held in the computation. □

3.3.5 $\langle 3\rangle 5$. CASE: $\beta \equiv \mathit{crashed}_i$
PROOF: The plausibility requirement implies that the detection event can only have been issued in line 16. This means that the failure detector suspected p_i in line 11. Since this happens after time t', step $\langle 2\rangle 2$ implies that this suspicion is accurate and that all control messages have been received. Hence, φ held in the computation. □

3.3.6 $\langle 3\rangle 6$. Q.E.D.
PROOF: Steps $\langle 3\rangle 4$ and $\langle 3\rangle 5$ cover all cases. □

3.4 $\langle 2\rangle 4$. Q.E.D.
PROOF: From step $\langle 2\rangle 3$ and the definition of $\Diamond S$. □

4 $\langle 1\rangle 4$. Q.E.D.
PROOF: From steps $\langle 1\rangle 2$ and $\langle 1\rangle 3$ and the definition of Sem_2. □

4.4 Using a Perfect Failure Detector ($\mathcal{P}$)

Even a perfect failure detector is not sufficient to perfectly detect all possible predicates. Indeed, from the point of view of predicate detection for general predicates the strongest possible detection semantics are the same as for $\Diamond\mathcal{P}$. This has already been shown in Theorem 5, i.e., it is impossible to detect with Sem_1 using any failure detector in $\mathcal{P}$. However, since Sem_2 is achievable using $\Diamond\mathcal{P}$, it is also achievable with $\mathcal{P}$.

Corollary 3 *In asynchronous systems with crash failures and a perfect failure detector it is possible to satisfy detection semantics* Sem_2.

PROOF SKETCH: The proof follows from Theorem 6 and the fact that $\mathcal{P} \succ \Diamond\mathcal{P}$.

Interestingly, we can detect predicates of the form $\alpha \wedge \beta$ according to Sem_1 using $\mathcal{P}$ if α is stable. The stability of α ensures that the predicate still holds, although control messages may still arrive from events that occurred immediately before the crash of the process (see Figure 9).

4.5 Introducing Failure Detector Class $\widehat{\mathcal{P}}$

A perfect failure detector is not sufficient to achieve optimal detection semantics in asynchronous systems. Intuitively, this is because $\mathcal{P}$ offers no information about the relative ordering of the crashes with respect to other application events. Consequently, we require a *plausible* failure detector that is still in $\mathcal{P}$. However, we show that this plausible failure detector is actually stronger than any failure detector in $\mathcal{P}$.

Definition 3 (ordered perfect failure detector) *An* ordered perfect failure detector *is a perfect failure detector which satisfies the following additional order property: Together with every "suspicion of* p_i*" event, the failure detector issues the event number of the last event that happened on* p_i.

We denote the class of all ordered perfect failure detectors by $\widehat{\mathcal{P}}$.

Theorem 7 $\widehat{\mathcal{P}} \succ \mathcal{P}$.

PROOF SKETCH: The fact that $\widehat{\mathcal{P}}$ is at least as strong as $\mathcal{P}$ is obvious. The proof that $\mathcal{P}$ is not at least as strong as $\widehat{\mathcal{P}}$ reuses the idea of Theorem 5 since $\widehat{\mathcal{P}}$ allows to distinguish the two situations which were indistinguishable if only $\mathcal{P}$ is available. Since we use a later theorem (Theorem 8) in this proof, we postpone the proof until later.

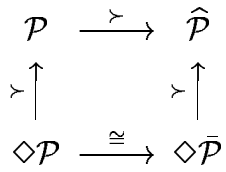

Figure 10: The ordering relations between $\widehat{\mathcal{P}}$ and other failure detector classes.

Figure 10 shows how $\widehat{\mathcal{P}}$ relates to other failure detector classes with respect to $\succ$. An ordered perfect failure detector allows to order crashes and normal process events causally, i.e., if a suspicion is issued by the failure detector and the associated sequence number is x, then delivery of the suspicion event can be held back until all control messages which have sequence numbers below or equal to x have been delivered. Hence, plausibility for an ordered perfect failure detector is achieved, which in turn means that the detection algorithm from Figure 6 allows to detect predicates with detection semantics Sem_1.

Theorem 8 *In asynchronous systems with crash failures and an ordered perfect failure detector it is possible to detect general predicates with detection semantics* Sem_1.

PROOF SKETCH: We use the standard algorithm in Figure 6 and show that it satisfies the properties of Sem_1. Again, we restrict ourselves to the case where $\varphi \equiv \alpha \wedge \beta$.

1 $\langle 1 \rangle 1$. The algorithm satisfies L.
PROOF: The proof is the same as in Theorem 6. □

2 $\langle 1 \rangle 2$. The algorithm satisfies S.
PROOF: The proof is the same as in Theorem 6 except that the time t' is the beginning of the computation. □

3 $\langle 1 \rangle 3$. Q.E.D.
PROOF: From steps $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ and the definition of Sem_1. □

We now give the proof of Theorem 7. Note that in the previous theorem we do not refer to Theorem 7, i.e., we have no circular dependencies.

ASSUME: There is an algorithm *Alg* which can be used to transform any failure detector in $\mathcal{P}$ into a failure detector in $\widehat{\mathcal{P}}$.
PROVE: false

1 $\langle 1 \rangle 1$. Consider a wrapper around the failure detector emulated with *Alg* which suspects a process only after the final control message has been received. This wrapper is implementable in asynchronous systems.
PROOF: Obvious since it is similar to enforcing FIFO ordering on control messages. □

2 $\langle 1 \rangle 2$. Sem_1 can be achieved using a perfect failure detector.
PROOF: From step $\langle 1 \rangle 1$ and Theorem 8. □

3 $\langle 1 \rangle 3$. Q.E.D.
PROOF: Step $\langle 1 \rangle 2$ is a contradiction to Theorem 5. □

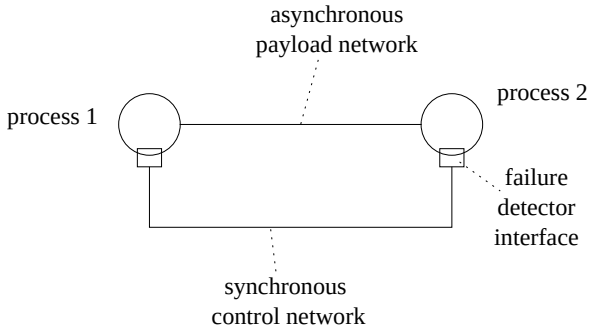

Figure 11: Implementing $\widehat{\mathcal{P}}$ using a low-bandwidth realtime network.

4.6 Implementing $\widehat{\mathcal{P}}$

Overall, perfect detection of general predicates in asynchronous systems is achievable only if we postulate a failure detector that is strictly stronger than a perfect failure detector. This is somewhat disappointing since even perfect failure detectors are very hard to implement in practice. However, ordered perfect failure detectors can still be implemented using a *timely computing base* [19]. In such an approach, an asynchronous network is enhanced by a synchronous real-time control network (see Figure 11). The asynchronous network is assumed to be high-bandwidth and is used for regular "payload" traffic while the synchronous network is only used for small control messages and therefore can be low-bandwidth. Under these assumptions it is possible to build a failure detection service that satisfies the order requirement of $\widehat{\mathcal{P}}$ by synchronously passing information about sent messages over the control network. Unfortunately, the execution of the event, and the sending on the synchronous and asynchronous network together have to be executed as an atomic action, which is a rather strong assumption. However, with this approach, a remote process accurately detects process crashes and is aware of the number of control messages sent prior to the crash.

5 Generalization to n Processes and m Monitors

In the previous sections we considered predicates local to one process in conjunction with a predicate on the operational state of this process, i.e., $\alpha \wedge \beta$. This section generalizes our results to scenarios with multiple processes (i.e., $n > 1$) and to multiple monitors (i.e., $m > 1$). The algorithms presented in Figures 6 and 7 are thus executed on every monitor. In the context of n processes and m monitors, the predicates are of the form $(\alpha_1 \wedge \beta_1)$ *op* $(\alpha_2 \wedge \beta_2)$ *op* $\ldots$, where *op* denotes either $\wedge$ or $\vee$.

In a system with n processes and m monitors, a causal broadcast mechanism is used so that the control messages are received by the monitors in *causal order* [16].

5.1 Observer Independence

Generalizing predicate detection to systems with multiple processes and multiple monitors gives rise to the issue of *observer independence*. Depending on the predicate φ and the setting in which it is evaluated, the validity of certain predicates depends on the observer [16]. Observer independence is achieved if *all possible* observations of the system result in the same truth value for φ [5]. Assume, for instance, that process p_1 executes an assignment $x := x + 1$ (i.e., event e_1^1 in Figure 12) and p_2 an assignment $y := y + 1$ (i.e., event e_1^2) on variables x and y which are initially 1. While monitor b_1 detects the predicate $\varphi \equiv x = 1 \wedge y = 2$, b_2 does not; the predicate φ is thus not observer independent, although the corresponding local predicates (i.e., $x = 1$ and $y = 2$) are. Charron-

Bost et al. [5] have shown that observer independence is maintained for the disjunction of observer independent predicates, whereas it generally is not for the conjunction.

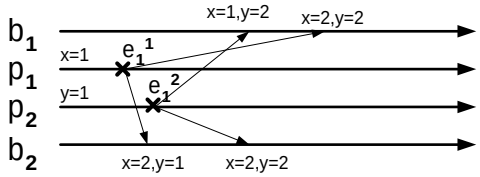

Figure 12: Example of an observer dependent predicate, where e_1^1 specifies the event $x := x + 1$ and e_1^2 the event $y := y + 1$.

In general, two approaches are possible to address the problem of observer independence: (a) limiting the set of observed predicates or (b) defining a different notion of what it means for φ to hold. We will focus on the former approach here. The latter approach has been studied by Gärtner and Kloppenburg [12].

5.2 Limiting the Set of Observed Predicates

The global predicates we are considering consist of the conjunction and disjunction of local predicates α_i and predicates about the operational state of processes β_i. Unreliable failure detection introduces a new source of observer dependence; observer independence for a global predicate generally depends on the failure detector. Obviously, β_i is detectable in an observer independent way if a perfect failure detector is available. However, predicates of type $\alpha_i \wedge \beta_i$ need an ordered perfect failure detector to be detectable in an observer independent way. A failure detector of class $\mathcal{P}$ is sufficient, if α_i is stable. On the other hand, a failure detector in $\Diamond\mathcal{P}$ only achieves "eventual" observer independence, whereas with $\Box\Diamond\mathcal{P}$, observer independence may never be achieved.

Limiting the set of observed predicates to observer independent predicates considerably reduces the number of global predicates that can be detected. However, following Charron-Bost et al. [5] and the above findings, we can construct new observer independent global predicates from smaller building blocks. For example, disjunctions of stable local predicates in conjunction with predicates on the operational state of processes, i.e., $(\alpha_i \wedge \beta_i) \vee (\alpha_j \wedge \beta_j)$, remain observer independent if a failure detector in $\mathcal{P}$ is available. On the other hand, conjunctions of observer independent local predicates and predicates about the operational state of processes, i.e., $(\alpha_i \wedge \beta_i) \wedge (\alpha_j \wedge \beta_j)$, generally are not observer independent.

6 Conclusion and Future Work

This paper investigates the predicate detection semantics that are achievable for general predicates using either failure detector classes $\Box\Diamond\mathcal{P}$, $\Diamond\mathcal{P}$, or $\mathcal{P}$. A general predicate thereby is of the form $\alpha \wedge \beta$, where α is a local predicate and β denotes a predicate on the operational state of a process, i.e., specifies whether a process has crashed or not. We define three different predicate detection semantics: perfect (i.e., Sem_1), stabilizing (Sem_2), and infinitely often accurate (Sem_3). Our results show that failure detector class $\Box\Diamond\mathcal{P}$ allows to detect general predicates according to Sem_3, whereas $\Diamond\mathcal{P}$ enables Sem_2. Somewhat surprisingly, a perfect failure detector is not sufficient to detect general predicates according to Sem_1. This leads to the definition of a stronger failure detector, called ordered perfect and denoted $\widehat{\mathcal{P}}$. With $\widehat{\mathcal{P}}$, perfect predicate detection (i.e., Sem_1) is achievable. An overview of our results is shown in Table 1.

Failures	Predicates	Failure Detector Class	Achievable Semantics	Reference
none	α	none	Sem_1	[7]
crash	α	none	Sem_1	[7]
crash	β	$\mathcal{P}$	Sem_1	[3]
crash	β	$\Diamond\mathcal{P}$	Sem_2	[3]
crash	β	$\Box\Diamond\mathcal{P}$	Sem_3	[10]
crash	$\alpha \vee \beta$	$\mathcal{P}$	Sem_1	Sect. 3.2
crash	$\alpha \wedge \beta$	$\Box\Diamond\mathcal{P}$	Sem_3	Thm 4
crash	$\alpha \wedge \beta$	$\Diamond\mathcal{P}$	Sem_2	Thm 6
crash	$\alpha \wedge \beta$	$\mathcal{P}$	Sem_2	Cor. 3
crash	$\alpha \wedge \beta$, α stable	$\mathcal{P}$	Sem_1	Sect. 4.4
crash	$\alpha \wedge \beta$	$\widehat{\mathcal{P}}$	Sem_1	Thm 8

Table 1: Strongest achievable predicate detection semantics with respect to types of predicates and the failure detector class available. Again, α denotes a predicate which refers only to normal process variables and β is a predicate referring only to the operational state of the process.

In the future, we plan to further investigate issues of observer independence in systems with n processes and m monitors and consider predicate detection under more severe fault assumptions, e.g., crash-recovery.

Acknowledgments: We wish to thank Sven Kloppenburg and Klaus Kursawe for their comments on an earlier version of this paper. We also thank the anonymous reviewers of DISC 2001 for suggesting to study the differences between the query model and the interrupt model of failure detectors.

References

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[2] Joffroy Beauquier and Synnöve Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of System Science*, 28(11):1177–1187, 1997.

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

[4] K. M. Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transations on Computing Systems*, 3(1):63–75, 1985.

[5] Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, January 1995.

[6] Bernadette Charron-Bost, Rachid Guerraoui, and André Schiper. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *International Conference on Dependable Systems and Networks (IEEE Computer Society)*, 2000.

[7] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[8] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

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

[10] Vijay K. Garg and J. Roger Mitchell. Distributed predicate detection in a faulty environment. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, 1998.

[11] Vijay K. Garg and J. Roger Mitchell. Implementable failure detectors in asynchronous systems. In *Proc. 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1530 in Lecture Notes in Computer Science, Chennai, India, December 1998. Springer-Verlag.

[12] Felix C. Gärtner and Sven Kloppenburg. Consistent detection of global predicates under a weak fault assumption. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, pages 94–103, Nürnberg, Germany, October 2000. IEEE Computer Society Press.

[13] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[14] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August/ September 1995.

[15] H. Matsui, M. Inoue, T. Masuzawa, and H. Fujiwara. Fault-tolerant and self-stabilizing protocols using an unreliable failure detector. *IEICE Transactions*, E83-D(10):1831–1840, October 2000.

[16] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, 1994.

[17] Amitabh Shah and Sam Toueg. Distributed snapshots in spite of failures. Technical Report TR84-624, Cornell University, Computer Science Department, July 1984.

[18] Subbarayan Venkatesan. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, April 1989.

[19] Paulo Veríssimo, Antonio Casimiro, and Christof Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, New York City, USA, June 2000. IEEE Computer Society Press.

[20] Pei yu Li and Bruce McMillin. Fault-tolerant distributed deadlock detection/resolution. In *Proceedings of the 17th Annual International Computer Software and Applications Conference (COMPSAC'93)*, pages 224–230, November 1993.