(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, Darmstadt, Germany, felix@informatik.tu-darmstadt.de

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

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].

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 $x_i = 1 \wedge crashed_i$, where $crashed_i$ 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 [14, 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 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 understanding, Shah and Toueg [16] 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* predicates. A predicate is set-decreasing whenever it holds for a set of processes, it also holds for a subset of these processes. 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 [16] 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 [19] or distributed termination [17].

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. For lack of space, we mostly only give proof sketches and relegate the full proofs to the extended version of the paper.

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. 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. Application processes can fail by crashing. Once a process has crashed, it does not recover any more during the execution. A process which crashes is called *faulty*. A non-faulty process is called *correct*. For simplicity, we assume that monitor processes do not fail. If it is clear from the context we will refer to application processes simply as "processes" and to monitor processes as "monitors".

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 process p_i. State changes are assumed to be atomic events local to some process such that an *execution* of the application processes can be modeled as a sequence of global stages $G_1, G_2, \ldots$, 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 the system.

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 process p_i is *suspected* by monitor b_j if the failure detector module associated with b_j detects the failure of p_i. Failure detectors are defined in terms of a *completeness* and an *accuracy* property. 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 in this paper [3]:

- (strong completeness) Eventually every application process that crashes is permanently suspected by every monitor process.
- (strong accuracy) No application process is suspected before it crashes.
- (eventual strong accuracy) There is a time after which correct application processes are not suspected by any monitor.

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. Garg and Mitchell [11] have introduced an additional accuracy property:

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

An *infinitely often accurate failure detector* satisfies strong completeness and infinitely often accuracy.

Chandra and Toueg [3] assume that failure detectors are passive modules. A monitor queries the module to learn about the operational state of other processes. In this setting, an infinitely often accurate failure detector offers no "real" accuracy at all. This is because a monitor may always query the failure detector when it is not accurate, i.e., when it erroneously suspects the process. Garg and Mitchell [11] therefore assume that the complete history of a failure detector is available so that an application does not miss the change of a failure detector output. This is equivalent to a model where the failure detector asynchronously (via *interrupts*) notifies the application about any change in its state. This model clearly makes more assumptions than the *query model* of Chandra and Toueg [3]. However, we now argue that these two models are equivalent for perfect and eventually perfect failure detectors in the context of predicate detection.

Since we use infinitely often accurate failure detectors, we necessarily use the *interrupt model* of Garg and Mitchell [11] in this paper. Note that query based failure detectors can be turned into interrupt style failure detectors by adding a concurrent task to the system which periodically queries the failure detector and interrupts the application whenever it detects state changes. For perfect failure detectors, this transformation can only increase the detection latency but maintains the accuracy and completeness properties. Similarly, for eventually perfect failure detectors, only false detections may go unnoticed. Hence, the predicate detection results we obtain for perfect and eventually perfect failure detectors in the interrupt model also hold in the query model.

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 $\Box\Diamond\mathcal{P}$. Intuitively, a failure detector fd_1 is *stronger* than a failure detector fd_2 (denoted $fd_1 \succeq fd_2$) if there exists a distributed algorithm that can be used to emulate fd_2 using fd_1 [3]. If $fd_1 \succeq fd_2$ and $fd_2 \succeq fd_1$ we write $fd_1 \cong fd_2$. If $fd_1 \succeq fd_2$ but not $fd_2 \succeq fd_1$ we say that fd_1 *is strictly stronger than* fd_2 and write $fd_1 \succ fd_2$. 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

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*). One generally assumes that the execution of the event and the sending of the control message execute as one atomic action.

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. Every such property can be written as the intersection of a *safety property* and a *liveness property* [1, 13].

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 require 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* which is 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 $Sem_1 \subseteq Sem_2 \subseteq Sem_3$.

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., $x_i = 1 \vee crashed_i$, 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.

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

Although we have not restricted the class of predicates, it should be noted that only those 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 achiev-

On monitor b:

```
variables:
  G : (s_1, ..., s_n) init (I_1, ..., I_n)
  crashed[1..n] array of {true,false} init (false, ..., false)
  φ : G × crashed → {true, false} init ⟨by application⟩
  h : {true, false} init false
algorithm:
  upon ⟨a message (i, e) arrives⟩ do
    ⟨update G[i] according to e⟩
    if φ(G, crashed) ∧ ¬h then
      h := true
      ⟨trigger detection event⟩
    elsif ¬φ(G, crashed) ∧ h then
      h := false
      ⟨trigger undetection event⟩
    end
  upon ⟨p_i is suspected or rehabilitated by failure detector⟩ do
    ⟨update crashed[i] accordingly⟩
    if φ(G, crashed) ∧ ¬h then
      h := true
      ⟨trigger detection event⟩
    elsif ¬φ(G, crashed) ∧ h then
      h := false
      ⟨trigger undetection event⟩
    end
```

Fig. 1. Generic algorithm for predicate detection in faulty environments.

able 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 1 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.*

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. 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 predicate detection algorithm before delivering the control message. Clearly, the wrapper can be implemented in asynchronous systems.

Transforming a failure detector *fd* in $\mathcal{P}$ into a plausible failure detector using the algorithm in Figure 2 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 *fd* rehabilitates p in order to receive *msg*. Later, *fd* 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 *fd* is made plausible with algorithm in Figure 2, 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 $\Box\Diamond\mathcal{P}$, e.g., an *infinitely often accurate failure detector*, can be implemented in these systems. Additionally we assume that such a failure detector is plausible. Then, predicate detection is only achievable with semantics Sem_3.

Theorem 1. *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 $\Box\Diamond\mathcal{P}$.*

On monitor b:

```
variables:
  suspects set of ⟨processes⟩ init ∅
  wasSuspected : {true, false} init false

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

Fig. 2. Implementation of a wrapper that makes any failure detector *fd* in $\Box\Diamond\mathcal{P}$ or $\Diamond\mathcal{P}$ plausible in asynchronous systems. Note that line 2 and 16 refer to events generated by *fd* while in lines 7, 10, 13, and 19 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 1.

Proof. We prove (a) using the standard algorithm in Figure 1. 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].

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 1 (a) holds also for these failure detectors, i.e., Sem_3 can be achieved. Note that the failure detector must be made plausible[1] before being used in the algorithm of Figure 1.

[1] Although making $\mathcal{P}$ plausible with the wrapper in Figure 2 weakens the failure detector to $\Diamond\mathcal{P}$, Theorem 1 (a) still holds.

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.*

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 3. 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 3 (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 3 (a)). Since the message can be delayed for an arbitrary amount of time, b_1 cannot distinguish between both scenarios.

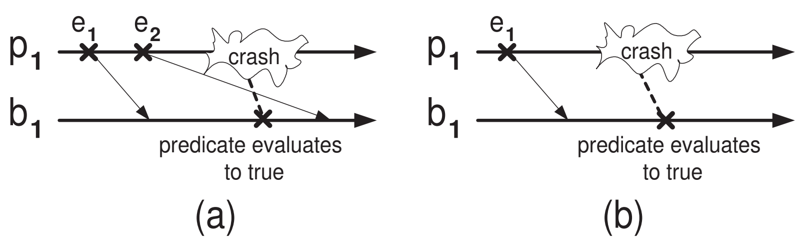

Fig. 3. Predicate $\varphi \equiv ec_1 = 1 \wedge crashed_1$ is not detectable according to detection semantics Sem_1 with any failure detector in $\mathcal{P}$.

Theorem 2. *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.*

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.*

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 *fd* 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 3. *In asynchronous systems with crash failures and any failure detector in $\diamond\mathcal{P}$ it is possible to satisfy detection semantics 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 2, 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.*

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 3).

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 4. $\widehat{\mathcal{P}} \succ \mathcal{P}$.

Proof. 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 2 since $\widehat{\mathcal{P}}$ allows to distinguish the two situations which were indistinguishable if only $\mathcal{P}$ is available.

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 1 allows to detect predicates with detection semantics Sem_1.

Theorem 5. *In asynchronous systems with crash failures and an ordered perfect failure detector it is possible to detect general predicates with detection semantics Sem_1.*

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. (This also shows that in these situations stabilizing service semantics (like Sem_2) are reasonable path to follow.) However, ordered perfect failure detectors can still be implemented using a *timely computing base* [18]. In such an approach, an asynchronous network is enhanced by a synchronous real-time control network. 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 consider 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 1 and 2 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* [15].

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 [15]. 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 4) 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

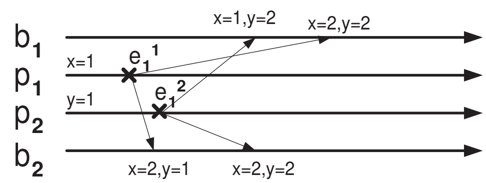

Fig. 4. Example of an observer dependent predicate, where e_1^1 specifies the event $x := x + 1$ and e_1^2 the event $y := y + 1$.

shown that observer independence is maintained for the disjunction of observer independent predicates, whereas it generally is not for the conjunction.

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 Conclusions

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

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.

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 1
crash	$\alpha \wedge \beta$	$\diamond\mathcal{P}$	Sem_2	Thm 3
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 5

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, indicating the importance of stabilizing detection semantics. 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.

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, Klaus Kursawe and the anonymous reviewers for their very insightful comments on this paper.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
2. J. Beauquier and S. 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. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.
4. K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

5. B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, Jan. 1995.
6. B. Charron-Bost, R. Guerraoui, and A. Schiper. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *International Conference on Dependable Systems and Networks (IEEE Computer Society)*, 2000.
7. C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
8. S. Dolev. *Self-Stabilization*. MIT Press, 2000.
9. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.
10. V. K. Garg and J. R. Mitchell. Distributed predicate detection in a faulty environment. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, 1998.
11. V. K. Garg and J. R. 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, Dec. 1998. Springer-Verlag.
12. F. C. Gärtner and S. 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, Oct. 2000. IEEE Computer Society Press.
13. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, Mar. 1977.
14. 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, Oct. 2000.
15. R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, 1994.
16. A. Shah and S. Toueg. Distributed snapshots in spite of failures. Technical Report TR84-624, Cornell University, Computer Science Department, July 1984.
17. S. Venkatesan. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, Apr. 1989.
18. P. Veríssimo, A. Casimiro, and C. 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.
19. P. yu Li and B. McMillin. Fault-tolerant distributed deadlock detection/resolution. In *Proceedings of the 17th Annual International Computer Software and Applications Conference (COMPSAC'93)*, pages 224–230, Nov. 1993.