Verification of Fault Tolerant Algorithms Using PEP*

Stephan Merkel

Institut für Informatik
Arcisstraße 21
Technische Universität München
D-80333 München, Germany
e-mail: merkel@informatik.tu-muenchen.de

Abstract. Petri net theory is an accepted approach to modelling and verifying distributed algorithms. In this paper we show how a tool based on Petri net theory, the PEP tool, can be used to model and verify fault tolerant algorithms for distributed systems. We conduct three case studies to illustrate our approach and to show the more specific problems we ran into, such as synchronization and crash detection in asynchronous systems, and we explain how we tried to solve them.

Table of Contents

* This work is partially supported by the Sonderforschungsbereich SFB-342 A3.

1 Introduction

In the design of distributed algorithms fault tolerance has become an indispensable issue. Much research has been carried out to classify failures and to determine the necessary processing resources to tolerate them. On the other hand, Petri net theory [14] is a successful approach to model and to verify concurrent algorithms. The PEP tool (**P**rogramming **E**nvironment based on **P**etri nets) [2] is based on Petri net theory. Its programming notation, the **B**asic **P**etri **N**et **P**rogramming **N**otation (B(PN)^2) [4], is an at the same time simple and flexible instrument to model parallel or distributed systems. The net semantics of B(PN)^2-programs provides for powerful verification methods. In this paper we discuss the possibility to make use of the advantages of PEP in the verification of fault tolerant algorithms for distributed systems. We first introduce our notion of fault tolerance in distributed systems in section 2. Then we briefly present the PEP tool and its functionality (section 3). In section 4 we discuss how the two can be combined to verify desired properties of solutions to typical problems. We will illustrate our ideas by three case studies (sections 5, 6, and 7), and we point out some conclusions we might draw from them.

2 Fault Tolerance in Distributed Systems

First of all, let us give the reader an idea of what fault-tolerant distributed algorithms are all about. Here and in the remainder, the material on fault-tolerance in distributed systems comes from [15], if not explicitely stated otherwise.

All distributed systems are subject to failures, but because of the dispersion of processing resources in a distributed system, these failures are usually only *partial*. Therefore it may be hoped that the tasks of failing processes can be taken over by the remaining components. Failures may occur as errors in message transmission (loss, corruption, ...) or as processor failures. The former type is handled by protocols. When talking about fault-tolerance in distributed systems, usually the latter is understood.[2]

There are two basic types of fault-tolerance: *robustness* and *self-stabilization*.

2.1 Self-Stabilization

Self-stabilizing algorithms offer protection against so-called *transient* failures, i.e., temporary misbehaviour of system components. Any number of failures of arbitrary types is allowable, but correct behaviour of the algorithm is suspended until some time after the repair of the failures. A self-stabilizing algorithm (sometimes simply called *stabilizing*) can be started in any system configuration, and eventually reaches an allowed state, and behaves according to its specifications from then on. Consequently, the effects of temporal failures die out, and also, there is no need to initialize the system consistently.

2.2 Robustness

Robust algorithms continuously show correct behaviour even when failures occur, but the number of failures is limited and the failure model must usually be known precisely. They rely on strategies such as voting, whereby a process will only accept certain information when sufficiently many other processes have also declared receipt of this information.

Robust algorithms protect against the *permanent* failure of a limited number of components. The surviving processes maintain correct (though possibly less efficient) behaviour during the repair and reconfiguration of the system. Therefore, robust algorithms must be used when the temporal interruption of service is unacceptable.

The following fault models are considered:

1. *Initially-dead processes.* A process is called initially dead if it does not execute a single step of its local algorithm
2. *Crash model.* A process is said to crash if it executes its local algorithm correctly up to some moment, and does not execute any step thereafter.
3. *Byzantine behaviour.* A process is said to be Byzantine if it executes steps that are arbitrary steps and, not in accordance with its local algorithm. In particular, a Byzantine process sends messages with an arbitrary content.

Observe that there is a clear hierarchy in these models: An initially-dead process is a special case of a crashed process, namely, where the crash occurs before the process executes the first step of its local algorithm. On the other hand, Byzantine behaviour includes the execution of no steps at all after some moment, in other words, a crash.

Robustness essentially deals with *decision problems*: consensus (in the simplest case), where all processes must agree on the same output value, call it v; election, where one process decides on a value

[2] Actually, this interpretation is only common in the context of distributed systems. It is a rather narrow notion of fault-tolerance. Many more interpretations of the term *fault* and what it means to *tolerate* it exist depending on the system considered and the design objectives, but we are not interested in them here.

meaning "the general" (e. g., $v = 1$), and all the others decide on a value meaning "lieutenant" ($v = 0$); renaming, where all processes decide on a different value within a given range, meaning their new identification, etc.

Robustness in Asynchronous Systems. A fundamental result, shown by Fischer, Lynch and Paterson [8], is that

(∗) in asynchronous systems there exists no deterministic 1-crash-robust consensus algorithm.

There are solutions, however, to many non-trivial problems when weakening the assumptions of this result:

- *Weaker fault model.* With initially dead processes, consensus and election are deterministically achievable.
- *Weaker coordination.* A less close coordination than consensus is required between processes. Some problems, e. g. renaming, are solvable in the crash-model.
- *Randomization.* Probabilistic consensus for the crash-model and for the Byzantine model.
- *Weak termination.* Termination only when a given process is correct. Byzantine broadcast for a correct general is solvable.

Initially-dead processes will be the subject of section 5, whereas renaming will be discussed in section 6.

Robustness in Synchronous Systems. In *synchronous systems* processes operate in *lockstep*, i. e., messages are received in the same pulse (a discrete step of the entire network) as that in which they are sent. No process can enter the next pulse before it has received all the messages expected in the current pulse. Note that no assumptions are made as to the *duration in time* of a pulse. Implementation of the pulse model (a so-called *synchronizer*) is always possible in the asynchronous model. In the simplest synchronizer, each process sends exactly one message to each neighbour in each pulse. If the simulated algorithm does not send a message in some pulse, the synchronizer adds an "empty" message. If the simulated algorithm sends more than one message, these messages are packed in one message of the synchronizer.

In the absence of failures, synchronous systems and asynchronous systems are equivalent with respect to their computational power. In particular, problems that are unsolvable in the asynchronous model remain unsolvable in the synchronous model. In synchronous systems, however, consensus is deterministically achievable even in the Byzantine model. Thus, in the presence of failures, they exhibit a strictly stronger computational power than asynchronous ones. This fundamentally higher resilience of synchronous systems over asynchronous systems and (∗) together imply the impossibility of any 1-crash-robust deterministic implementation of a synchronizer in the asynchronous model. Intuitively, a crashed process can be detected because it has failed to send a message expected in a certain pulse. But any synchronizer must rely on the messages expected in a pulse to determine the pulse's end.

This dilemma can be resolved if processes possess clocks and an upper bound on message delay is known. This model is referred to as asynchronous bounded delay (ABD) network after Chou *et al.* [5]. In ABD networks, crash-robust implementation of the pulse model is possible. An algorithm achieving consensus in a crash-robust synchronous system is the subject of section 7.

3 Introduction to PEP

PEP (**P**rogramming **E**nvironment based on **P**etri nets) [2] is a software tool which allows for the design, modelling, analysis and verification of parallel systems. Although it is possible to edit and verify Petri nets directly, in a typical session the system is modelled in a special programming notation that has a net semantics. This notation, called $B(PN)^2$, will be the subject of the following subsection. Many algorithms to automatically check system properties and to analyse the net representing the semantics of a $\mathrm{B(PN)}^2$ program are implemented in PEP. Two of them are of a special interest for us. We will talk about them in subsection 3.2.

3.1 The Programming Notation $\mathbf{B(PN)^2}$

There exists a large number of parallel programming notations to encode parallel systems, e.g. CCS [11], CSP [9], and $\mathrm{B(PN)}^2$ [4], and just as many parallel programming languages, e.g. `occam`, Ada, Modula-3. In order to apply formal methods to practical problems, PEP uses a programming notation which is very simple and yet very flexible. $\mathrm{B(PN)}^2$ (**B**asic **P**etri **N**et **P**rogramming **N**otation) supports shared memory- as well as message based-topologies. Message buffers are modelled via FIFO buffers with a capacity ranking from zero (handshake) up to infinity (unbounded capacity). It is a block-structured programming notation which features procedures, a nested parallel construct, guarded commands and non-determinism, but it does not support priorities and structural data.

Behind $\mathrm{B(PN)}^2$ an abundant theory [3] has been glued together that allows a compositional translation of a program π into a Petri net N_π. Properties of π can be translated into properties over elements of N_π. This enables the usage of the Petri net theory for the verification of desired properties of π.

In the following two subsections, we first present the syntax of $\mathrm{B(PN)}^2$ and then show how to encode Peterson's mutual exclusion algorithm [13] in $\mathrm{B(PN)}^2$.

Syntax of $\mathbf{B(PN)^2}$. In Figure 1 the formal syntax of $\mathrm{B(PN)}^2$ is given. Keywords are printed in `teletype` while identifiers are printed in *italic*.

program	::= block	
block	::= `begin` scope `end`	
scope	::= com | `var` varlist : type; scope	
varlist	::= *var-name* | varlist, varlist	
type	::= set | set `init` const | `chan` *cap* `of` set	($cap \in \mathbb{N} \cup \{\omega\}$)
com	::= $\langle$ expr $\rangle$ | com$\|$com | com;com | `do` alt-set `od`	
alt-set	::= com;`exit` | com;`repeat` | alt-set [] alt-set	
expr	::= 'v | v' | v | c! | c? | const | expr *op* expr | *op* expr | (expr)	
		$op \in \{+, -, *, /, \wedge, \vee, \neg, <, >, =\}$

Fig. 1. Syntax of $\mathrm{B(PN)}^2$.

In order to become more familiar with this programming notation we stress upon some features of $\mathrm{B(PN)}^2$ [2]:

- *Atomic actions.* The expressions given in between angular brackets denote atomic actions. In the translation into a Petri net, every such action is associated with one, or a set of alternative, single transitions. The expression of an atomic action is a predicate over pre- and post-values of variables touched by the action. For instance, an atomic action expressing the assignment $x := y$ would be written as $\langle x' = {}'y \wedge y' = {}'y\rangle$, whereas ${}'v$ (resp. v') denotes the pre-value (resp. post-value) of v. Any value change which makes the predicate true enables the action. For convenience, we use the unprimed version of a variable to denote its pre- and post-value in a single term, e.g., $\langle x' = {}'y \wedge y' = {}'y\rangle$ can also be written as $\langle x' = y\rangle$.
- *Unification of shared memory and channel communication.* To describe channel communication in predicative style, $\mathrm{B(PN)}^2$ features $c?$ and $c!$ as primitives denoting the value last read on channel c and the value last output to channel c, respectively. They are analogous to the pre- and post-values of variables.
- *Unification of choices and loops.* $\mathrm{B(PN)}^2$ contains a single `do-od` clause both for choices and for loops. The symbol [] separates alternatives, which can be ended either by the keyword `exit` (indicating the exit from the loop) or by the keyword `repeat` (indicating a repetition of the loop).

Peterson's Mutual Exclusion Algorithm. In Figure 2 we show how to encode Peterson's mutual exclusion algorithm [13] in $\mathrm{B(PN)}^2$. The two `do-od` constructs encode processes p_1 and p_2. p_1 is in the critical section when `in_cs1` is `true` and leaves it when `in_cs1` is set to `false`, and so forth. The mutual exclusion property specifies that both processes never enter their critical sections simultaneously.

3.2 Verification of $\mathrm{B(PN)}^2$ Programs in PEP

Of the different possible ways to verify system properties in PEP, two have turned out useful for our purposes. Both use the unfolding [10] of the net underlying the $\mathrm{B(PN)}^2$ program. The unfolding is the finite prefix of the (possibly infinite) branching process of the Petri net. It represents the complete behaviour of the system, but in most cases it avoids the state explosion problem. For more detail, please refer to [10].

The first method is a deadlock check. It searches the unfolding for a marking in which no transition is enabled. In the case of Peterson's algorithm, for example, the answer will be "system is deadlock-free", which means that the two processes cannot wait for one another at the same time. The second method is model checking. We specify a formula in a branching time logic [6] and check if our system is a "model" of the formula, i.e., if it satisfies the constraints. A formula consists of a combination of place labels and boolean or temporal operators. A formula consisting of a place label, e.g. 'P1', is true iff the place labelled 'P1' carries a token. The operators are the familiar boolean operators such as $\wedge$, $\vee$, $\neg$, and $\Rightarrow$, and two temporal operators, $\Diamond$ and $\Box$. $\Diamond\Phi$ means that a marking is reachable where formula Φ holds. $\Box\Phi$ means that Φ holds in every reachable marking and can be derived as $\neg\Diamond\neg\Phi$. This temporal logic is usually referred to as L_1; see [7]. To give an example, consider the mutual exclusion property of Peterson's algorithm. We would express this property as $\neg\Diamond($'*in_cs1*=**true**' $\wedge$ '*in_cs2*=**true**'$)$. The model checker tells us that this formula evaluates to **true**. Note that we can also write constraints for system states in single quotes, which greatly facilitates the specification of formulae.

```
begin
    var in1, in2: {false, true} init false;
    var in_cs1, in_cs2: {false, true} init false;
    var hold: {1,2} init 1;

    do
        ⟨ in1' = true ⟩;
        ⟨ hold' = 1 ⟩;
        ⟨ in2=false ∨ hold=2 ⟩;
        ⟨ in_cs1'=true ⟩; ⟨ in_cs1'=false ⟩;
        ⟨ in1' = false ⟩; repeat
    od
||
    do
        ⟨ in2' = true ⟩;
        ⟨ hold' = 2 ⟩;
        ⟨ in1=false ∨ hold=1 ⟩;
        ⟨ in_cs2'=true ⟩; ⟨ in_cs2'=false ⟩;
        ⟨ in2' = false ⟩; repeat
    od
end
```

Fig. 2. Peterson's mutual exclusion algorithm.

4 Methodology for Modelling Fault Tolerant Algorithms in PEP

In this section we want to discuss how we can model fault tolerant algorithms in PEP such that critical properties can easily be verified. The two types of fault tolerance differ radically as to their aims and their characteristics we are usually interested in.

4.1 Difficulties with Self-Stabilizing Algorithms

For stabilizing algorithms, we want to prove that the system eventually reaches a legal configuration, no matter what the initial configuration is. Any kind and number of faults is allowed, but it is assumed that they all are eventually repaired and each component behaves correctly according to its local algorithm from then on. This could be modelled by writing a $B(PN)^2$-program which implements the algorithm with all variables uninitialized. The initial configuration is then completely arbitrary. In order to verify self-stabilization of the system, we must find a formula Φ which describes what we call a "legal" configuration and then we must show that in every execution, there will eventually be a configuration satisfying Φ. This cannot be expressed in our particular kind of branching time logic, L_1. We can merely express that a stable configuration is reachable from every reachable configuration, and that once the system is stable, it will not destabilize itself again. But neither is truly a proof of self-stabilization. However, we point out that we do not *generally* exclude the possibility of applying PEP to self-stabilizing algorithms; on the contrary, this might be the subject of future work. Here we will focus on robustness.

4.2 Modelling Robustness

The situation is more favourable to our methods if the desired system property is coordination between communicating system components even in the presence of failures, leading to consistently correct, though possibly less efficient system behaviour. On the one hand, robust algorithms usually terminate in a final state as soon as sufficient coordination has been achieved. On the other hand, if the number of faults exceeds the resilience of the algorithm, some, if not all processes will be unable to complete their local algorithm. We took advantage of these characteristics to develop a simple verification method. The approach to model and verify robust algorithms was to encode the algorithm for one process as a procedure in B(PN)^2 and then invoke a number of incarnations of this procedure in parallel. Only the successful completion of sufficiently many processes would result in an infinite loop. Hence, if and only if the algorithm fulfils the specified requirements, the system is deadlock-free, allowing for a simpler check of the desired properties than model checking. Nonetheless, we used some model checking to complement our verification.

Two deadlock checkers are available in PEP. The heuristics of these two programs differ significantly, depending on the case. For the first one, the time needed depends mainly on the property itself: if some configuration is a deadlock configuration, it is found very quickly, but a long time is needed to show that *none* of them is. For the other, which is based on linear constraint programming, heuristics are good if the number of cutoff events is around ten percent or higher. This condition was met for most of our deadlock-free systems. We chose which program to use accordingly.

In principle, there is only one major problem with our approach for deterministic algorithms: they can only be verified for a given number of processes. As will be seen below, under this restriction algorithms can be modelled and verified without loss of generality even for synchronous systems, even though the notion of bounded delays is absent in general Petri nets. In contrast, deadlock checking and model checking do not yield information about the *probability* of a specific execution of the system. Probabilistic algorithms are thus (in PEP) impossible to prove correct, because they have infinite computations in which no decision is ever taken, only the probability of such an execution converges to 0 for infinitely many events.

5 Initially-dead Processes

The interest of this section is to give a first example of how we can apply the techniques available in PEP to a fairly simple problem related to fault-tolerance. Consensus is achievable in the initially-dead model even in asynchronous systems because we know that a process will not fail after having sent at least one message. We will see some typical elements of algorithms whose aim is coordination between processes, such as shouting messages and sending sets as message content. We describe the coding of these elements in B(PN)^2 in detail in appendix A.1.

5.1 A Solvable Problem: Computation of a Knot

Let N denote the number of processes in a given system, and t the number of faulty processes. Consensus and election are solvable in the initially-dead model as long as a minority of the processes can be faulty ($t < N/2$). A larger number of initially-dead processes cannot be tolerated. To achieve consensus (and any other kind of coordination between processes) in a system where less than $N/2$ processes are initially dead, it suffices to compute a knot of correct processes, call it K. The solution of any other problem is then easy. To choose a correct process as a leader, for example, simply elect the process with the largest identity in K. To reach consensus, the leader broadcasts its input and all correct processes decide on it.

5.2 A t-initially-dead-robust Consensus Algorithm

Specification. For the following algorithm and for those in the remainder, suppose a given distributed system which consists of N processes $P_1, \ldots, P_N$ that can be definitely identified by *identification numbers* $id_1, \ldots, id_N$. The processes form a complete graph G, also called a *clique*, that is, each pair of processes is connected via a bidirectional communication channel of unbounded capacity. By the algorithm by Fischer, Lynch, and Paterson [8], each of the correct processes computes *the same collection of correct processes* in the system. Since termination is only guaranteed for $t < N/2$, there must be at least $L = \lceil (N+1)/2 \rceil$ correct processes.

Implementation. The algorithm operates as follows [15]: Initially every process P_i shouts its identity id_i, that is, it sends a message of type *name* containing id_i to all its neighbours, including itself. It then waits for the receipt of L identities, which altogether form the private set of successors of the process in G, $Succ_{P_i}$. Next, P_i shouts $Succ_{P_i}$ and then computes the collection of correct processes $Alive_{P_i}$ by unifying the sets of successors received from all processes already known to be alive. The following program presents the implementation of the algorithm for N processes in B(PN)^2. Instead of bidirectional channels we use pairs of unidirectional ones, where $c_{i \to j}$ stands for a channel from which P_j can read a value written by P_i. For the sake of simplicity, let $\mathbf{I} = \{id_1, \ldots, id_N\}$. Comments are introduced by `//` and are printed in `teletype`.

begin

$\underbrace{\textbf{var } c_{i\to j} : \textbf{chan } \omega \textbf{ of } 2^{\mathbf{I}};}_{(i,j)\in\mathbf{I}\times\mathbf{I}}$

var *started*, *num_done* : $\{0, \ldots, N-t\}$ **init** 0;

proc process (

  const p: $\mathbf{I}$,

  $\underbrace{\textbf{ref } in_i : \textbf{chan } \omega \textbf{ of } 2^{\mathbf{I}},}_{i\in\mathbf{I}}$

  $\underbrace{\textbf{ref } out_i : \textbf{chan } \omega \textbf{ of } 2^{\mathbf{I}},}_{i\in\mathbf{I}}$

)

begin

  var *num_succ* : $\{0, \ldots, N\}$ **init** 0;

  var *succ*, *alive*, *rcvd*, *pre* : $2^{\mathbf{I}}$ **init** $\emptyset$;

  `// shout <id>`

  begin $\langle\, out_1! = p \,\rangle \parallel \langle\, out_2! = p \,\rangle \parallel \ldots \parallel \langle\, out_N! = p \,\rangle$ **end**;

  do

    $\langle\, num\_succ = L \,\rangle$;

    exit

$\square$

$\langle$ *num_succ* $< L \rangle$;

`// receive L <id>-messages`

do

$\langle$ *succ'* $=$ *'succ* $\cup\ in_1? \rangle$; **exit**

$\square$

$\langle$ *succ'* $=$ *'succ* $\cup\ in_2? \rangle$; **exit**

$\square$

$\vdots$

$\square$

$\langle$ *succ'* $=$ *'succ* $\cup\ in_N? \rangle$; **exit**

od;

$\langle$ *num_succ'* $=$ *'num_succ* $+ 1 \rangle$;

repeat

od;

`// shout <succ>`

begin $\langle out_1! = succ \rangle \parallel \langle out_2! = succ \rangle \parallel \ \ldots \parallel \langle out_N! = succ \rangle$ **end**;

$\langle$ *alive'* $= succ \rangle$;

`// receive all <succ>-messages`

do

$\langle\ alive \subseteq rcvd\ \rangle$;

exit

$\square$

$\langle\ alive \not\subseteq rcvd\ \rangle$;

`// first: if a new process has been seen: clear its <id>-message`

do

$\langle\ succ = alive\ \rangle$;

exit

$\square$

$\langle\ id_1 \notin succ \wedge id_1 \in alive\ \rangle$;

$\langle\ in_1? = in_1?\ \rangle$; $\langle$ *succ'* $=$ *'succ* $\cup\ id_1 \rangle$;

repeat

$\square$

$\vdots$

$\square$

$\langle\ id_N \notin succ \wedge id_N \in alive\ \rangle$;

$\langle\ in_N? = in_N?\ \rangle$; $\langle$ *succ'* $=$ *'succ* $\cup\ id_N \rangle$;

repeat

od;

`// then: receive a <succ>-message`

do

$\langle\ pre' = in_1?\ \rangle$;
$\langle\ alive' = {}'alive \cup id_1\ \rangle$; $\langle\ rcvd' = {}'rcvd \cup id_1\ \rangle$;
exit
$[]$
$\langle\ pre' = in_2?\ \rangle$;
$\langle\ alive' = {}'alive \cup id_2\ \rangle$; $\langle\ rcvd' = {}'rcvd \cup id_2\ \rangle$;
exit
$[]$
$\vdots$
$[]$
$\langle\ pre' = in_N?\ \rangle$;
$\langle\ alive' = {}'alive \cup id_N\ \rangle$; $\langle\ rcvd' = {}'rcvd \cup id_N\ \rangle$;
exit
od;
$\langle\ alive' = {}'alive \cup pre\ \rangle$;
repeat
od;
$\langle\ num\_done' = {}'num\_done + 1\ \rangle$;
`// all correct processes have decided: loop`
do $\langle\ num\_done = N - t\ \rangle$; **repeat od**
end; `// proc process`

begin
do
$\langle\ ({}'started < N - t) \wedge (started' = {}'started + 1)\ \rangle$;
$\underbrace{\text{process } (id_1, c_{1\to 1}, \ldots, c_{N\to 1}, c_{1\to 1}, \ldots, c_{1\to N})}_{P_1}$
exit
$[]$
$\langle\ {}'started = N - t\ \rangle$;
exit
od
$\|$
do
$\langle\ ({}'started < N - t) \wedge (started' = {}'started + 1)\ \rangle$;
$\underbrace{\text{process } (id_2, c_{1\to 2}, \ldots, c_{N\to 2}, c_{2\to 1}, \ldots, c_{2\to N})}_{P_2}$
exit
$[]$
$\langle\ {}'started = N - t\ \rangle$;
exit
od
$\|$
$\vdots$

$\|$

do

$\langle$ ($'started < N - t$) $\wedge$ ($started' = 'started + 1$) $\rangle$;

$\underbrace{\text{process } (id_N, c_{1\to N}, \ldots, c_{N\to N}, c_{N\to 1}, \ldots, c_{N\to N})}_{P_N}$

exit

$\Box$

$\langle$ $'started = N - t$ $\rangle$;

exit

od

end

end

Verification. The main property of the algorithm is the termination of the main loop, after which a process has computed the desired collection of correct processes. This can be verified by checking if the system as a whole contains a deadlock, since a process can perform infinitely many events if and only if all correct processes have exited their main loops. If the bound on the number of faulty processes $t < N/2$ is sharp, for $N = 3$, for example, the system should be deadlock-free if $t \leq 1$ and it should deadlock otherwise. We have verified these properties for $N = 3$ and $N = 4$ as shown in Table 1. The case where $N = t$ is omitted because it is trivial.

It is clear that the problem is symmetrical for any combination of faulty processes. The t faulty processes can hence more simply be modelled w.l.o.g. by omitting the last t procedure calls instead of counting them in the variable `started`. This was done for the combination $N = 4, t = 2$. Given the choice, the computation of the unfolding was unacceptably complex otherwise, i.e. 100 MBytes of main memory were exceeded.[3] In the case of $N = 4, t = 1$ our loop condition was $\langle$**true**$\rangle$ instead of $\langle$ $done \geq N - t$ $\rangle$, because here too the unfolding was more easily computed. But then deadlock-freedom only tells us that at least one process can terminate. To be sure that really all three correct processes will, we additionally used model checking. The result of checking the formula $\Diamond\Box$($'done$=1' $\vee$ $'done$=2') was **false** and was returned immediately (0.00 seconds). So in no execution can the number of looping processes be stable at one or two, but must eventually be three.

The table nicely illustrates how programs which only differ in the value of one constant, namely those for $N = 3, t = 1$ and $N = 3, t = 2$, lead to nets whose size is almost equal but whose unfoldings clearly reflect their different behaviour. Also, the above mentioned characteristics of our deadlock checking programs are confirmed.

6 Crashed Processes

As stated above, crashed processes, even though they are still considered as "benign" failures because they don't "lie" about their own state, are more difficult to handle than initially-dead processes. Consensus cannot be deterministically guaranteed in asynchronous systems even in the presence of a single crash

[3] Of course, this complexity would still have been absolutely acceptable if we hadn't had the choice.

N	t	Petri net $\lvert P\rvert$	$\lvert T\rvert$	unfolding $\lvert C\rvert$	$\lvert E\rvert$	cutoffs	time (sec.) unfolding	deadl. check
3	0	423	1676	5001	2717	683	17.48	698.30
3	1	511	2281	1610	744	68	3.18	19.20
3	2	529	2362	226	86	3	2.43	0.02
4	1	1199	12180	4444	2248	186	51.85	749.00
4	2	747	6220	160	77	2	0.07	0.00

Table 1. Results for the t-initially-dead consensus algorithm.

failure. In this section, we present an algorithm for a problem which is nonetheless solvable in the crash model. The focus here was to gain insights in the possible ways to model a crash. Except for that, the same approach as for initially-dead processes could be used.

6.1 A Solvable Problem: Renaming

In the renaming problem, each process has a distinct identity which can be taken from an arbitrarily large domain. Each process must decide on a new name, taken from a smaller domain, such that all new names are different. For our purposes, let the new domain be $D = \{1, \ldots, K\}$. The renaming problem is solvable in the asynchronous model because it requires less close coordination between processes than does consensus or election. To better understand what this means, compare the collection of legal output vectors for consensus and renaming. Achieving consensus means that all processes decide on the same output value. Consequently, if output values are in $\{0, 1\}$, the two legal output vectors are $(0, 0, \ldots, 0)$ and $(1, 1, \ldots, 1)$. In the case of election, the output vectors must be in $\{(1, 0, \ldots, 0), (0, 1, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}$. In contrast, in the renaming problem the collection of legal output vectors can be characterized as $\{(n_1, \ldots, n_N) \in D^N \mid i \neq j \Rightarrow n_i \neq n_j\}$. It is the much greater power of this collection over consensus or election which makes the problem solvable in the crash model.

6.2 A t-crash-robust Renaming Algorithm

Specification. Tel [15] gives an algorithm for renaming by Attiya *et al.* [1] which tolerates up to $t < N/2$ crashes (t is a parameter of the algorithm) and renames in a space of size $K = (N - t/2)(t + 1)$. It has been shown that no renaming algorithm can tolerate $N/2$ or more crashes.

Implementation. The algorithm operates as follows [15]: Process P maintains a set V_P of process inputs that P has seen; initially, V_P contains just P's input value x_P. Every time P receives a set of inputs including ones that are new for P, V_P is extended by the new inputs. Upon starting and every time V_P is extended, P shouts its set. In the variable c_P P counts the number of times it has received copies of its current set V_P. Initially c_P is 0, and c_P is incremented each time a message containing V_P is received. The receipt of a set V may cause V_P to grow, necessitating a reset of c_P. If the new value of V_P equals V (i. e., if V is a strict superset of the old V_P), c_P is set to 1, otherwise to 0. Process P is said to reach a *stable set* V if c_P becomes $N - t$ when the value of V_P is V. In other words, P has received for the $(N - t)$th time the current value V of V_P. The following program presents the implementation of the algorithm for N processes in B(PN)^2. Since processes here are not required to identify the sender of a message they receive, every process P_i has exactly one input channel c_i associated with it. Let $\mathbf{I} = \{id_1, \ldots, id_N\}$ be the set of input values, i. e., the original process names.

```
begin

var num_done: {0, ..., N − t};
var c_1, ..., c_N: chan ω of 2^I;

proc process(
    const x: I,
    ref in: chan ω of 2^I,
    ref out1: chan ω of 2^I,
    ⋮
    ref outN: chan ω of 2^I)
begin
    var Vp, V: 2^I init ∅;
    var cp: {0, ..., N} init 0;
    var done: {true,false} init false;

    ⟨ Vp' = x ⟩;
    // shout private value
    begin ⟨ out1! = x ⟩ ‖ ⟨ out2! = x ⟩ ‖ ... ‖ ⟨ outN! = x ⟩ end;
    do
      do
        // if all correct processes have decided: loop
        ⟨ num_done ≥ N − t ⟩;
        repeat
      []
        // else: receive a set of private values
        ⟨ num_done < N − t ⟩;
          ⟨ V' = in? ⟩;
        exit
      od;

      do
        ⟨ V = Vp ⟩;
          ⟨ cp' = 'cp + 1 ⟩;
          do
            // if Vp is stable for the first time: decide
            ⟨ (cp = N − t) ∧ ¬ done ⟩;
              ⟨ done' = true ⟩;
              ⟨ num_done' = 'num_done + 1 ⟩;
            exit
```

```
          □
            ⟨ (cp < N − t) ∨  done ⟩;
            exit
          od;
        exit
      □
        // ignore "old" information
        ⟨ V ⊂ Vp ⟩;
        exit
      □
        // else: new input; update Vp and restart counting
        ⟨ V ⊈ Vp ⟩;
          ⟨ (Vp ⊂ V ∧ cp' = 1) ∨ (Vp ⊄ V ∧ cp' = 0) ⟩;
          ⟨ Vp' = 'Vp ∪ V ⟩;
          // shout new Vp
          begin ⟨ out1! = Vp ⟩ ‖ ⟨ out2! = Vp ⟩ ‖ ... ‖ ⟨ outN! = Vp ⟩ end;
        exit
      od;
      repeat
    od
end;
```

$$\underbrace{\text{process } (id_1, c_1, c_1, \ldots, c_N)}_{P_1}$$

$\|$

$$\underbrace{\text{process } (id_2, c_2, c_1, \ldots, c_N)}_{P_2}$$

$\|$

$\vdots$

$\|$

$$\underbrace{\text{process } (id_N, c_N, c_1, \ldots, c_N)}_{P_N}$$

end

Verification. We first wanted to show that our implementation is correct by proving termination of the algorithm in the case of $N = 3, t = 0$. But although we tried all the variants of code we could think of as possibly reducing the state space of the system, we were unable to compute the unfolding for this B(PN)^2 program, i. e. 800 Mbytes main memory were exceeded. For the variant which should be the smallest of all in behaviour, the computation of the net representing the semantics of the program failed.[4] Consequently, we could not perform any checks to prove correctness of our implementation. To understand this, observe that compared to the t-initially-dead consensus algorithm, this algorithm does not operate in rounds.

[4] We stopped the process after 10 hours.

Stated differently, there is no moment when a process must receive a message from a majority of its neighbours before it continues with its local algorithm, so some processes may be a lot faster than others. Further, a process does not necessarily terminate after deciding, because it might still need to help slower processes decide. All this significantly increases the concurrency in the system. Also, channel capacity must be sufficiently high to cope with such slow processes. For instance, if $N = 3$, capacity must be at least 6. Taken all this into account, our difficulties to compute the unfolding are not really surprising.

7 Byzantine Processes

We will now turn to crash-robust synchronous systems. As already mentioned, in the Byzantine model, a process performs steps that are arbitrary steps and, not in accordance with its local algorithm. Of the hierarchy of faults considered here, a Byzantine process is the most malicious kind, because not only may it crash, but it may also lie to the other processes. In particular, it may vote on an arbitrary value, rendering agreement a much more difficult task than, for example, in the crash model. It is nevertheless achievable even in the Byzantine model for synchronous systems. The main question we were concerned with in modelling a Byzantine-robust algorithm was how we could validly prove the correctness of a synchronous algorithm despite the fact that our systems are inherently asynchronous.

7.1 Reaching Agreement in the Presence of Byzantine Failures

In ABD networks, a crashed process is easily detected and there can be no doubt as to which process actually crashed. But if a process receives contradictory information from some other processes, more information is needed to sort out the faulty one. One way is to have additional rounds of information exchange and to compare the values received. Another is to let processes sign the information to be transmitted and to let the recipient verify the signature. This is called *authentication*. It relies on the assumption that a faulty process cannot "forge" a correct process's signature. Thus, a Byzantine process may "lie" about its own state, but not about the messages it has received. We won't consider the approach using authentication here. We will rather focus on an algorithm looking for a reliable majority in the collection of private values directly or indirectly "claimed" by the processes.

7.2 A 1-Byzantine Consensus Algorithm

Pease, Shostak, and Lamport [12] presented an algorithm reaching consensus in the Byzantine model. When no authentication is used, the resiliency of this algorithm is $t < N/3$, which was shown to be optimal. Hence, to tolerate a single fault, a minimum of four processors [5] is needed.

Specification. In a set of N isolated processors, of which less than $N/3$ are Byzantine, each nonfaulty processor computes an *interactive consistency (i. c.) vector*, i.e., a vector satisfying the following two requirements.

1. **Consistency.** The nonfaulty processors compute exactly the same vector.
2. **Non-triviality.** The element of this vector corresponding to a given nonfaulty processor is the private value of that processor.

[5] We use the term "processor" instead of "process" in this section, because it is the term exclusively used in [12]. For our purposes, the two terms are synonymous.

Implementation. The algorithm for the single fault case operates as follows [12]: In a first round of information exchange the processors exchange their private values. In the second round they exchange the results obtained in the first round. The exchange having been completed, each nonfaulty processor p records its private value V_p for the element of the interactive consistency corresponding to p itself. The element corresponding to every other processor q is obtained by examining the three received reports of q's value (one of these was obtained directly from q in the first round, and the others from the remaining two processors in the second round). If at least two of the three reports agree, the majority value is used. Otherwise, a default value such as "NIL" is used. The following program presents the implementation of the algorithm for the single fault case in B(PN)^2.

begin

var *icv1_1, icv1_2, icv1_3, icv1_4*: $\{0, 1, nil\}$;
var *icv2_1, icv2_2, icv2_3, icv2_4*: $\{0, 1, nil\}$;
var *icv3_1, icv3_2, icv3_3, icv3_4*: $\{0, 1, nil\}$;
var *icv4_1, icv4_2, icv4_3, icv4_4*: $\{0, 1, nil\}$;
var $\underbrace{c_{i \rightarrow j} : \textbf{chan } \omega \textbf{ of } \{0, 1, nil\};}_{i,j \in \{1,2,3,4\},\ i \neq j}$

proc process (
  const *vp*: {0,1},
  ref *in1*: **chan** ω **of** {0,1,*nil*},
  ref *in2*: **chan** ω **of** {0,1,*nil*},
  ref *in3*: **chan** ω **of** {0,1,*nil*},
  ref *out1*: **chan** ω **of** {0,1,*nil*},
  ref *out2*: **chan** ω **of** {0,1,*nil*},
  ref *out3*: **chan** ω **of** {0,1,*nil*},
  ref *z1*: {0,1,*nil*},
  ref *z2*: {0,1,*nil*},
  ref *z3*: {0,1,*nil*},
  ref *z4*: {0,1,*nil*})
begin
  var *v1, v2, v3*: {0,1,*nil*};
  var *u1, u2, u3*: {0,1,*nil*};
  var *w1, w2, w3*: {0,1,*nil*};

  `// first round: send private value to all neighbours, then receive`
  begin $\langle$ *out1*! = *vp* $\rangle$ ‖ $\langle$ *out2*! = *vp* $\rangle$ ‖ $\langle$ *out3*! = *vp* $\rangle$ **end**;
  begin $\langle$ *v1'* = *in1*? $\rangle$ ‖ $\langle$ *v2'* = *in2*? $\rangle$ ‖ $\langle$ *v3'* = *in3*? $\rangle$ **end**;

  `// second round, part 1: transmit each neighbour's value`
  `// to the first of the other two, then receive`
  begin $\langle$ *out1*! = *v2* $\rangle$ ‖ $\langle$ *out2*! = *v1* $\rangle$ ‖ $\langle$ *out3*! = *v1* $\rangle$ **end**;
  begin $\langle$ *u2'* = *in1*? $\rangle$ ‖ $\langle$ *u1'* = *in2*? $\rangle$ ‖ $\langle$ *w1'* = *in3*? $\rangle$ **end**;

  `// second round, part 2: transmit each neighbour's value`

```
// to the second of the other two, then receive
begin ⟨ out1! = v3 ⟩ ‖ ⟨ out2! = v3 ⟩ ‖ ⟨ out3! = v2 ⟩ end;
begin ⟨ u3' = in1? ⟩ ‖ ⟨ w3' = in2? ⟩ ‖ ⟨ w2' = in3? ⟩ end;

// compute i.c.vector (z1,z2,z3,z4)
⟨ z1' = vp ⟩;
do
  ⟨ u1 = w1 ∨ u1 = v1 ⟩;
    // there is a majority for u1
    ⟨ z2' = u1 ⟩;
  exit
▯
  ⟨ v1 = w1 ⟩;
    // there is a majority for v1
    ⟨ z2' = v1 ⟩;
  exit
▯
  ⟨¬ (u1 = w1 ∨ u1 = v1 ∨ v1 = w1) ⟩;
    // there is no majority
    ⟨ z2' = nil⟩;
  exit
od;
do
  ⟨ u2 = w2 ∨ u2 = v2 ⟩;
    ⟨ z3' = u2 ⟩;
  exit
▯
  ⟨ v2 = w2 ⟩;
    ⟨ z3' = v2 ⟩;
  exit
▯
  ⟨¬ (u2 = w2 ∨ u2 = v2 ∨ v2 = w2) ⟩;
    ⟨ z3' = nil⟩;
  exit
od;
do
  ⟨ u3 = w3 ∨ u3 = v3 ⟩;
    ⟨ z4' = u3 ⟩;
  exit
▯
  ⟨ v3 = w3 ⟩;
    ⟨ z4' = v3 ⟩;
  exit
```

▯

$\langle \neg\ (u3 = w3 \vee u3 = v3 \vee v3 = w3)\ \rangle$;

$\langle\ z4' = nil\ \rangle$;

exit

od

end;

begin

process (V_{P_1}, $c_{2\to1}$, $c_{3\to1}$, $c_{4\to1}$, $c_{1\to2}$, $c_{1\to3}$, $c_{1\to4}$, *icv1_1, icv1_2, icv1_3, icv1_4*)

$\|$

process (V_{P_2}, $c_{1\to2}$, $c_{3\to2}$, $c_{4\to2}$, $c_{2\to1}$, $c_{2\to3}$, $c_{2\to4}$, *icv2_2, icv2_1, icv2_3, icv2_4*)

$\|$

process (V_{P_3}, $c_{1\to3}$, $c_{2\to3}$, $c_{4\to3}$, $c_{3\to1}$, $c_{3\to2}$, $c_{3\to4}$, *icv3_3, icv3_1, icv3_2, icv3_4*)

$\|$

process (V_{P_4}, $c_{1\to4}$, $c_{2\to4}$, $c_{3\to4}$, $c_{4\to1}$, $c_{4\to2}$, $c_{4\to3}$, *icv4_4, icv4_1, icv4_2, icv4_3*)

end

end

Verification. In order to verify the consistency of the result vectors $icv_1, \ldots, icv_4$, we added atomic actions after the procedure calls whose evaluation to *true* would altogether guarantee the desired equality of vector components. Upon successful completion of these actions, an infinite loop is entered, as in the programs above, to identify deadlock-freedom with fulfilment of the algorithm's specification. This way, it was an easy matter to show that the algorithm would operate according to its specification in the case of no fault; see Table 2. The corresponding lines of code were omitted here to improve readability. Please refer to appendix A.4 for more detail.

To model the worst possible fault, we replaced one call of a correct processor by a call of an additional procedure which merely receives all messages sent to it and sends arbitrary messages. We argue that this behaviour is equivalent to Byzantine behaviour, although our procedure does not explicitely include the possibility of a crash. In order to be able to simulate a crash, the capability of the correct processors to handle a NIL-value received was included in the program, a NIL-message meaning that the sender has crashed. Reaching agreement is clearly easier in the case of a crash, since if the faulty processor sends only a "crash-signal" after some moment, there might be a majority of NIL-values (depending on the moment of the crash), thus eliminating the case of true inconsistency. We have verified the correctness of the i.c. vector for the case of arbitrary messages. The low number of cutoff events (only around five percent) discouraged us from trying to be faster using the program relying on linear constraint programming. The time needed to check deadlock-freedom (more than 264 hours) illustrates the complexity of this particular problem. It led us to the assumption that explicitely including a crash would strain our patience beyond measure.

It is almost trivial to show that consistency cannot be guaranteed in the presence of more than one fault. If e.g. processors A and B consistently lie to C about the private value of D, C will decide on the value received from A and B, convinced that D is faulty, whereas D will decide on its private value. Here again we see how quickly our deadlock checker finds existing deadlocks.

		Petri net		unfolding			time (sec.)	
N	t	$\|P\|$	$\|T\|$	$\|C\|$	$\|E\|$	cutoffs	unfolding	deadl. check
4	0	701	879	1630	587	82	2.13	14.10
4	1	504	409	42276	14724	752	1098.97	952288.67
4	2	579	473	396	124	4	0.13	0.02

Table 2. Results for the 1-Byzantine consensus algorithm.

8 Conclusion

We have discussed how fault tolerance in distributed systems can be modelled and verified using PEP. We have pointed out what remains to be done to be able to verify self-stabilization and we have shown that in principle PEP is well suited to model and to verify robust algorithms. We have conducted three case studies to illustrate this point. We had to find ways to mimic synchronization and detectable crashes.

We had some problems in computing the unfoldings of nets, which we need in order to use an exact verification method, and deadlock checking in several cases took quite long. Note, however, that currently only identical rather than symmetric markings are considered to find cut-off events in the unfolding. But taken into account the ever same basic structure of the programs, i. e., parallel process procedures at the top level, much symmetry is inherent to every system in consideration. As a matter of fact, this is true also for many more typical distributed algorithms, not only in fault tolerance. An improvement of the unfolding algorithm with respect to symmetry might therefore significantly cut down on the size of the resulting nets and further improve the usefulness of the PEP tool.

The greatest difficulty resides in the set-oriented approach of robust algorithms. It makes verification of large systems impossible. In our implementations we have shown how we can deal with sets even when no structured data is available. As far as we know, dealing with sets is a problem for all automatic verification methods.

References

1. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM* **37** (1990), 524-548
2. E. Best. Partial order verification with PEP. In D. Peled, G. Holzmann, and V. Pratt, editors, *Proc, POMIV'96, Partial Order Methods in Verification.* American Mathematical Society, August 1996.
3. E. Best, R. Devillers, and J. G. Hall. The Box Calculus: A New Causal Algebra with Multi-Label Communication. In *Advances in Petri Nets 92*, volume 609 of *LNCS*, 21 – 69, 1992.
4. E. Best and R. P. Hopkins. $B(PN)^2$ – a Basic Petri Net Programming Notation. In *Procs. of PARLE '93*, volume 694 of *LNCS*, 379 – 390, 1993.

5. C. T. Chou, I. Cidon, I. S. Gopal, and S. Zaks. Synchronizing asynchronous bounded delay networks. *IEEE Trans. Commun.* **38**, 2 (1990), 144-147.
6. E. A. Emerson. Temporal and Modal Logic. In Jav von Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, chapter 16, pages 995 - 1072. Elsevier, 1990.
7. J. Esparza. Model Checking Using Net Unfoldings. In *Science of Computer Programming* **23** (1994), 151-195.
8. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. In *J. ACM* **32** (1985), 374-382.
9. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666 – 677, 1978.
10. K. L. McMillan. *Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits*. School of Computer Science, Carnegie Mellon University, May 1992.
11. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
12. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. In *J. ACM* **27** (1980), 228-234.
13. M. Raynal. *Algorithms for Mutual Exclusion.* North Oxford Academic, 1986.
14. W. Reisig. Petri Nets. An Introduction, volume 4 of EATCS, *Monographs on Theoretical Computer Science*, Springer 1985/86.
15. G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press (1994).

A B(PN)^2 Source Code

A.1 Practical Problems and Solutions

B(PN)^2 does not provide any structured data, only integer numbers. But for the collections of names used in two of the considered algorithms, some data structure to represent sets of process identities is needed. This was solved by using prime numbers as identifiers and the product of their respective first powers as a set. The number of values needed to cover all possible multiplications equals the number of elements in the power set, of course. Operations on these sets such as union and test for subset create a large number of transitions and need to be modified every time the number of processes grows.

Another problem is that sometimes the sender of a message must be known by the recipient, as for example in the t-initially-dead robust consensus algorithm. The sender can add its identity to the message, but to encode this is costly if the only type of message available is an integer number. It is more efficient to allow exactly one process to write a value to a given channel, so the sender is automatically known. This makes the number of channels grow to N^2, compared to only N channels required if message content can be more complex. And, of course, we limited the capacity of the channels to the maximum number of messages in the queue at a time.

In the implementations presented below, the feature that processes send messages to themselves was preserved. In many cases, it facilitates design, analysis and proof of correctness for conventional methods. For our methods it is a considerable disadvantage, and dropping it could certainly reduce the size of the state space for the algorithms concerned. But we decided to stick to the originals here, with one exception: the t-initially-dead consensus algorithm for $N = 3, t = 0$.

A.2 The t-initially-dead Consensus Algorithm

We only present the implementation for $N = 4, t = 1$ here. The necessary changes for other parameters are clear. Note, however, that for $N = 3$ we used the `do <done >= min>; repeat od` construct as explained above.

```
///////////////////////////////////////////////////////////////
// Implementation of the consensus algorithm for four
// processes, one of which is initially-dead
///////////////////////////////////////////////////////////////

begin
  // the channels
  var c22, c23, c25, c27,
      c32, c33, c35, c37,
      c52, c53, c55, c57,
      c72, c73, c75, c77: chan 2 of {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210};
  const max = 4;                       // N, the total number of processes
  var started, done: {0..max} init 0;
  const min = 3;                       // min. # of correct processes L = N-t
  const corr = 3;                      // actual # of correct processes

  proc KNOT( const id: {2,3,5,7},
             ref in1: chan 2 of {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210},
             ref in2: chan 2 of {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210},
             ref in3: chan 2 of {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210},
             ref in4: chan 2 of {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210},
             ref out1: chan 2 of {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210},
             ref out2: chan 2 of {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210},
             ref out3: chan 2 of {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210},
             ref out4: chan 2 of {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210},
             ref done: {0..max} )
  begin
    var num_succ: {0..max} init 0;             // # of elements in succ
    var succ, alive, rcvd, pre: {1,2,3,5,6,7,10,14,15,21,30,35,42,70,105,210} init 1;

    // begin shout <name,p>;
    <out1!=id>; <out2!=id>; <out3!=id>; <out4!=id>;

    // while #Succ(p) < L
    do <num_succ = min>;
       exit
    [] <num_succ < min>;

         // do begin receive <name,q>; Succc(p):= Succ(p) u {q} end;
         do <succ'='succ*in1?>; exit
         [] <succ'='succ*in2?>; exit
```

```
     [] <succ'='succ*in3?>; exit
     [] <succ'='succ*in4?>; exit
     od;
     <num_succ'='num_succ+1>;
   repeat
od;

// shout <pre, p, Succ(p)>;
<out1!=succ>; <out2!=succ>; <out3!=succ>; <out4!=succ>;

// Alive(p) := Succ(p);
<alive'=succ>;

// while Alive(p) not subset of Rcvd(p)
do <rcvd % alive = 0>;
   exit
[] <rcvd % alive # 0>;
     // if there are unreceived first-round-messages, discard them
     do <alive%2=0 and succ%2#0>;
          <in1?=in1?>;
          <succ'='succ*2>;
        repeat
     [] <alive%3=0 and succ%3#0>;
          <in2?=in2?>;
          <succ'='succ*3>;
        repeat
     [] <alive%5=0 and succ%5#0>;
          <in3?=in3?>;
          <succ'='succ*5>;
        repeat
     [] <alive%7=0 and succ%7#0>;
          <in4?=in4?>;
          <succ'='succ*7>;
        repeat
     [] <alive=succ>;
        exit
     od;

     // do begin receive <pre,q,succ>;
     do <pre'=in1?>;
          // alive := alive u {2};
          do <alive%2=0>;
             exit
```

```
     [] <alive%2#0>;
          <alive'='alive*2>;
        exit
     od;
     // rcvd := rcvd u {2};
     <rcvd'='rcvd*2>;
   exit
[] <pre'=in2?>;
     // alive := alive u {3};
     do <alive%3=0>;
        exit
     [] <alive%3#0>;
          <alive'='alive*3>;
        exit
     od;
     // rcvd := rcvd u {3};
     <rcvd'='rcvd*3>;
   exit
[] <pre'=in3?>;
     // alive := alive u {5};
     do <alive%5=0>;
        exit
     [] <alive%5#0>;
          <alive'='alive*5>;
        exit
     od;
     // rcvd := rcvd u {5};
     <rcvd'='rcvd*5>;
   exit
[] <pre'=in4?>;
     // alive := alive u {7};
     do <alive%7=0>;
        exit
     [] <alive%7#0>;
          <alive'='alive*7>;
        exit
     od;
     // rcvd := rcvd u {7};
     <rcvd'='rcvd*7>;
   exit
od;

// alive := alive u pre;
```

```
      do <alive % pre = 0>;
         exit
      [] <pre%2=0 AND alive%2#0>;
           <alive'='alive*2>;
         repeat
      [] <pre%3=0 AND alive%3#0>;
           <alive'='alive*3>;
         repeat
      [] <pre%5=0 AND alive%5#0>;
           <alive'='alive*5>;
         repeat
      [] <pre%7=0 AND alive%7#0>;
           <alive'='alive*7>;
         repeat
      od;
    repeat
  od;

  // Compute a knot
  <done'='done+1>;

  // loop a)
  // do <done >= min>; repeat od
  // loop b)
  do <true>; repeat od

end; // proc KNOT()

///////////////////////// main() ////////////////////////////////

// nondeterministically start $corr processes
begin
  do <('started < corr) AND (started'='started+1)>;
       KNOT ( 2, c22, c32, c52, c72, c22, c23, c25, c27, done );
     exit
  [] <started >= corr>;
     exit
  od
  ||
  do <('started < corr) AND (started'='started+1)>;
       KNOT ( 3, c23, c33, c53, c73, c32, c33, c35, c37, done );
     exit
  [] <started >= corr>;
```

```
      exit
   od
   ||
   do <('started < corr) AND (started'='started+1)>;
        KNOT ( 5, c25, c35, c55, c75, c52, c53, c55, c57, done );
      exit
   [] <started >= corr>;
      exit
   od
   ||
   do <('started < corr) AND (started'='started+1)>;
        KNOT ( 7, c27, c37, c57, c77, c72, c73, c75, c77, done );
      exit
   [] <started >= corr>;
      exit
   od
  end // start processes

end
```

A.3 The t-crash-robust Renaming Algorithm

For the renaming algorithm, we present our implementation for three correct processes. We only used one input channel for each process here, because it is a straightforward implementation of the original algorithm, where the sender of a message need not be known.

```
/////////////////////////////////////////////////////////////
// Implementation of the renaming algorithm by Attiya et al.
// for three correct processes
/////////////////////////////////////////////////////////////

begin
  const max = 3;  // N, the total number of processes
  const corr = 3; // the number of correct processes
  const min = 2;  // min. number of processes (N - t)
  const cap = 6;  // channel capacity

  proc REN( const xp: {2,3,5},
            ref num_done: {0..max},
            ref in: chan cap of {1,2,3,5,6,10,15,30},
            ref out1: chan cap of {1,2,3,5,6,10,15,30},
            ref out2: chan cap of {1,2,3,5,6,10,15,30},
            ref out3: chan cap of {1,2,3,5,6,10,15,30}
          )
```

```
begin
  var vp, vr: {1,2,3,5,6,10,15,30} init 1;
  var cp: {0..3} init 0;
  var done: {true, false} init false;

  // begin Vp := {xp}; cp := 0; shout <set, Vp>;
  <'vp=1 AND vp'=xp>;
  begin <out1!=xp> || <out2!=xp> || <out3!=xp> end;

  // while true do begin receive <set, V>;
  do <vr'=1>;
    // loop if all correct processes have decided, else receive
    do <num_done>=corr>; repeat
    [] <num_done<corr>; <'vr=1 AND vr'=in?>; exit
    od;

    // if V = Vp then begin
    do <vr=vp>;
         <cp'='cp+1>;
         // if cp = N - t and yp = b then
         do <cp=min AND NOT done>;
              // (* Vp is stable for the first time: decide *)
              <'done=false AND done'=true>;
              <num_done'='num_done+1>;
            exit
         [] <cp#min OR done>; exit
         od;
       exit

    // else if V subset of Vp then skip
    [] <vr#vp AND vp%vr=0>; exit

    // else (* new input; update Vp and restart counting *)
    [] <vr#vp AND vp%vr#0>;
         // begin if Vp strict subset of V then cp := 1 else cp := 0;
         <(vr%vp=0 AND cp'=1) OR (vr%vp#0 AND cp'=0)>;
         // Vp := Vp u V;
         <vr%2#0 OR (vp'%2=0 AND vp'='vp) OR (vr%2=0 AND 'vp%2#0 AND vp'='vp*2)>;
         <vr%3#0 OR (vp'%3=0 AND vp'='vp) OR (vr%3=0 AND 'vp%3#0 AND vp'='vp*3)>;
         <vr%5#0 OR (vp'%5=0 AND vp'='vp) OR (vr%5=0 AND 'vp%5#0 AND vp'='vp*5)>;
         // shout <set, Vp>
         begin <out1!=vp> || <out2!=vp> || <out3!=vp> end;
       exit
```

```
      od;
      repeat
    od
  end; // proc REN

  begin
    // the names before renaming
    const p1 = 2;
    const p2 = 3;
    const p3 = 5;

    var num_done: {0..max} init 0;
    var c1, c2, c3: chan cap of {1,2,3,5,6,10,15,30};

    begin
      REN( p1, num_done, c1, c1, c2, c3 )
      ||
      REN( p2, num_done, c2, c1, c2, c3 )
      ||
      REN( p3, num_done, c3, c1, c2, c3 )
    end
  end
end
```

A.4 The 1-Byzantine Consensus Algorithm.

No faulty processor. The following B(PN)^2-code is the implementation of the algorithm for four correct processors.

```
///////////////////////////////////////////////////////////////////////
// Implementation the algorithm by Pease, Shostak and Lamport
// for Byzantine agreement with 4 processors, none of which
// is faulty.
//////////////////////////////////////////////////////////////////////

begin
  // NIL value
  const nil=2;

  // the i.c. vectors p_i
  var p11, p12, p13, p14,
      p21, p22, p23, p24,
      p31, p32, p33, p34,
      p41, p42, p43, p44: {0..nil} init 0;
```

```
// private values
const vp1=0;
const vp2=1;
const vp3=0;
const vp4=0;

begin
  // the channels
  var c12, c13, c14,
      c21, c23, c24,
      c31, c32, c34,
      c41, c42, c43: chan 1 of {0..nil};

  proc AGREE( const vp: {0..1},
            ref in1: chan 1 of {0..nil},
            ref in2: chan 1 of {0..nil},
            ref in3: chan 1 of {0..nil},
            ref out1: chan 1 of {0..nil},
            ref out2: chan 1 of {0..nil},
            ref out3: chan 1 of {0..nil},
            ref z1: {0..nil},
            ref z2: {0..nil},
            ref z3: {0..nil},
            ref z4: {0..nil} )
  begin
    var v1, v2, v3,         // private values received in the first round
        u1, u2, u3,
        w1, w2, w3: {0..2} init 0;
        // {u,w}x means: private value of x-th neighbour claimed by the
        //               {first,second} neighbour <> x

    // round 1: send private value to all neighbours
    begin <out1!=vp> || <out2!=vp> || <out3!=vp> end;

    // receive private value from all neighbours
    begin <'v1=0 AND v1'=in1?> || <'v2=0 AND v2'=in2?> || <'v3=0 AND v3'=in3?> end;

    // round 2, part 1:
    // relay p's private value to the first neighbour other than p
    begin <out1!=v2> || <out2!=v1> || <out3!=v1> end;

    // receive the respective values
```

```
  begin <'u2=0 AND u2'=in1?> || <'u1=0 AND u1'=in2?> || <'w1=0 AND w1'=in3?> end;

  // round 2, part 2:
  // relay p's private value to the second neighbour other than p
  begin <out1!=v3> || <out2!=v3> || <out3!=v2> end;

  // receive the respective values
  begin <'u3=0 AND u3'=in1?> || <'w3=0 AND w3'=in2?> || <'w2=0 AND w2'=in3?> end;

  // compute decision vector (z1,z2,z3,z4)
  <'z1=0 AND z1'=vp>;
  do <u1=w1>; <'z2=0 AND z2'=u1>; exit
  [] <u1=v1>; <'z2=0 AND z2'=u1>; exit
  [] <v1=w1>; <'z2=0 AND z2'=v1>; exit
  [] <NOT (u1=w1 OR u1=v1 OR v1=w1)>;
       <'z2=0 AND z2'=nil>;
     exit
  od;
  do <u2=w2>; <'z3=0 AND z3'=u2>; exit
  [] <u2=v2>; <'z3=0 AND z3'=u2>; exit
  [] <v2=w2>; <'z3=0 AND z3'=v2>; exit
  [] <NOT (u2=w2 OR u2=v2 OR v2=w2)>;
       <'z3=0 AND z3'=nil>;
     exit
  od;
  do <u3=w3>; <'z4=0 AND z4'=u3>; exit
  [] <u3=v3>; <'z4=0 AND z4'=u3>; exit
  [] <v3=w3>; <'z4=0 AND z4'=v3>; exit
  [] <NOT (u3=w3 OR u3=v3 OR v3=w3)>;
       <'z4=0 AND z4'=nil>;
     exit
  od
end; // proc AGREE

begin
  AGREE( vp1, c21, c31, c41, c12, c13, c14, p11, p12, p13, p14 )
  ||
  AGREE( vp2, c12, c32, c42, c21, c23, c24, p22, p21, p23, p24 )
  ||
  AGREE( vp3, c13, c23, c43, c31, c32, c34, p33, p31, p32, p34 )
  ||
  AGREE( vp4, c14, c24, c34, c41, c42, c43, p44, p41, p42, p43 )
end
```

```
  end;

  // requirements for interactive consistency even if p4 is faulty
  <p11=vp1>; <p11=p21>; <p11=p31>;
  <p22=vp2>; <p22=p12>; <p22=p32>;
  <p33=vp3>; <p33=p13>; <p33=p23>;
  <p14=p24>; <p14=p34>;

  // additional requirements if p4 is correct
  <p44=vp4>; <p44=p14>;
  <p11=p41>; <p22=p42>; <p33=p43>;

  // loop if gotten so far
  do <true>; repeat od

end
```

One faulty processor. The following source code shows the same program with one Byzantine processor.

```
//////////////////////////////////////////////////////////////////////
// Implements the algorithm by Pease, Shostak and Lamport
// for Byzantine agreement with 4 processors, one of which
// is faulty.
//////////////////////////////////////////////////////////////////////

begin
  // the i.c. vectors p_i
  var p11, p12, p13, p14,
      p21, p22, p23, p24,
      p31, p32, p33, p34: {0..1} init 0;

  // private values
  const vp1=0;
  const vp2=1;
  const vp3=0;

  begin
    // the channels
    var c12, c13, c14,
        c21, c23, c24,
        c31, c32, c34,
        c41, c42, c43: chan 1 of {0..1};
```

```
proc CORR( const vp: {0..1},
          ref in1: chan 1 of {0..1},
          ref in2: chan 1 of {0..1},
          ref in3: chan 1 of {0..1},
          ref out1: chan 1 of {0..1},
          ref out2: chan 1 of {0..1},
          ref out3: chan 1 of {0..1},
          ref z1: {0..1},
          ref z2: {0..1},
          ref z3: {0..1},
          ref z4: {0..1} )
begin
  var v1, v2, v3,         // private values received in the first round
      u1, u2, u3,
      w1, w2, w3: {0..1} init 0;
      // {u,w}x: as above

  // round 1: send private value to all neighbours
  begin
    <out1!=vp>
    ||
    <out2!=vp>
    ||
    <out3!=vp>
  end;

  // receive private values from all neighbours
  begin <'v1=0 AND v1'=in1?> || <'v2=0 AND v2'=in2?> || <'v3=0 AND v3'=in3?> end;

  // round 2, part 1:
  // relay p's private value to the first neighbour other than p
  begin <out1!=v2> || <out2!=v1> || <out3!=v1> end;

  // receive the respective values
  begin <'u2=0 AND u2'=in1?> || <'u1=0 AND u1'=in2?> || <'w1=0 AND w1'=in3?> end;

  // round 2, part 2:
  // relay p's private value to the second neighbour other than p
  begin <out1!=v3> || <out2!=v3> || <out3!=v2> end;

  // receive the respective values
  begin <'u3=0 AND u3'=in1?> || <'w3=0 AND w3'=in2?> || <'w2=0 AND w2'=in3?> end;
```

```
    // compute decision vector (z1,z2,z3,z4)
    <'z1=0 AND z1'=vp>;
    <'z2=0 AND ((z2'=u1 AND u1=w1) OR (z2'=u1 AND u1=v1) OR (z2'=v1 AND v1=w1))>;
    <'z3=0 AND ((z3'=u2 AND u2=w2) OR (z3'=u2 AND u2=v2) OR (z3'=v2 AND v2=w2))>;
    <'z4=0 AND ((z4'=u3 AND u3=w3) OR (z4'=u3 AND u3=v3) OR (z4'=v3 AND v3=w3))>
  end; // proc CORR

  proc BYZ( ref in1: chan 1 of {0..1},
            ref in2: chan 1 of {0..1},
            ref in3: chan 1 of {0..1},
            ref out1: chan 1 of {0..1},
            ref out2: chan 1 of {0..1},
            ref out3: chan 1 of {0..1} )
  begin
    // shout arbitrarily
    begin <out1!=out1!> || <out2!=out2!> || <out3!=out3!> end;
    // receive
    begin <in1?=in1?> || <in2?=in2?> || <in3?=in3?> end;
    // shout arbitrarily
    begin <out1!=out1!> || <out2!=out2!> || <out3!=out3!> end;
    // receive
    begin <in1?=in1?> || <in2?=in2?> || <in3?=in3?> end;
    // shout arbitrarily
    begin <out1!=out1!> || <out2!=out2!> || <out3!=out3!> end;
    // receive
    begin <in1?=in1?> || <in2?=in2?> || <in3?=in3?> end
  end; // proc BYZ

  begin
    CORR( vp1, c21, c31, c41, c12, c13, c14, p11, p12, p13, p14 )
    ||
    CORR( vp2, c12, c32, c42, c21, c23, c24, p22, p21, p23, p24 )
    ||
    CORR( vp3, c13, c23, c43, c31, c32, c34, p33, p31, p32, p34 )
    ||
    BYZ( c14, c24, c34, c41, c42, c43 )
  end
end;

// requirements for interactive consistency even if p4 is faulty
<p11=vp1>; <p11=p21>; <p11=p31>;
<p22=vp2>; <p22=p12>; <p22=p32>;
<p33=vp3>; <p33=p13>; <p33=p23>;
```

```
  <p14=p24>; <p14=p34>; // the correct processes must have equal values
                        // for the faulty one

  // loop if gotten so far
  do <true>; repeat od

end
```