On the design of communication protocols that support coordination loss

Antonio A.F. Loureiro *Osvaldo S.F. de Carvalho*

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Caixa Postal 702, 30161-970 Belo Horizonte, MG
E-mail: {loureiro,vado}@dcc.ufmg.br

Resumo

Em teste de conformidade de protocolos, existe uma classe importante de erros chamada "perda de coordenação" que é difícil de ser prevista já que sua origem é externa ao ambiente que executa o teste e a implementação que está sendo testada.

Neste trabalho nós usamos "auto-estabilização" como um princípio para tratar deste problema o que permitirá obter implementações de protocolos mais confiáveis. Nós apresentamos um novo algoritmo para projetar protocolos auto-estabilizantes, seguido de um exemplo e de duas novas relações de conformidade que consideram ambientes que podem exibir erros referentes à perda de coordenação.

Abstract

In protocol conformance testing, there is an important class of errors, namely coordination loss that cannot be anticipated because some source of error is *external* to both the tester and the implementation under test (IUT). Furthermore, it is not possible to simulate their occurrence exhaustively in the test environment. Therefore they are very difficult to catch in the testing phase.

In this paper we propose using self-stabilization as a design principle to overcome this testing problem. This will improve the reliability of protocol implementations derived from self-stabilizing protocol specifications. We present a novel algorithm and the corresponding design principles to design self-stabilizing protocols, give an example of a self-stabilizing protocol, and define two new relations based on an environment that exhibits coordination loss.

1 Introduction

The ultimate goal of OSI is to allow the interconnection of different systems that follow the same set of standards. In a real open system it is common to have the same set of protocols implemented on different architectures and environments by different people. These implementations must be tested for conformance to the specifications.

Conformance testing is one of the basic activities in the protocol development process. The conformance testing and methodology framework (CTMF) [14] identifies two types of conformance requirements: static and dynamic. Static requirements are related to functions requirements. Dynamic requirements refer to the dynamic behavior of the system. There is also a protocol implementation conformance statement (PICS) produced by the implementor describing the functions and options present in the implementation, and the protocol implementation extra information for testing (PIXIT) that provides specific information for testing purposes.

According to this framework, an implementation conforms to its specification if it satisfies both the static and dynamic conformance requirements, and is consonant with its PICS. This should be demonstrated through one of the four test methods defined in the CTMF.

Note that if a protocol specification S contains an error and an implementation I faithfully implements S then I conforms to S. Of course, if the error is identified in one of the protocol development phases, i.e., specification, implementation, testing, or even after that, the specification must be revised and all changes made in S should be reflected in I accordingly.

In practice, when I is derived from S it is assumed that the specification is correct. The fundamental point in this statement is the meaning of correctness. The specification may be only correct with respect to a specific property such as the absence of deadlock although there are other protocol properties that should be checked. Due to the complexity of protocol testing there are limitations in this process and different solutions have been proposed. For instance, some alternatives to exhaustive reachability analysis — the technique used in most automated validation systems — are based on search heuristics, hashing techniques, and reduction methods. For an overview of validation methods, their limitations and alternatives see for instance [12, 13].

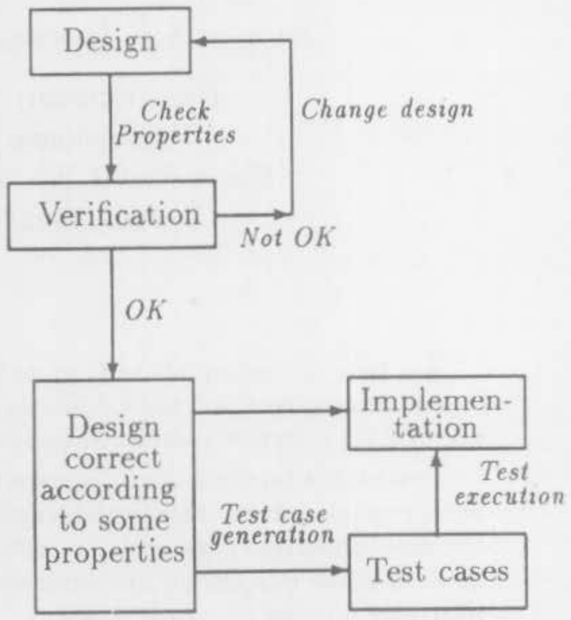


Figure 1: Partial view of the protocol development process.

In Figure 1 we show a partial view of the protocol development process. It is clear that in the protocol development process as is in software in general, verification and testing are complementary and mutually supportive techniques.

Different types of protocol testing have been proposed: diagnostic, conformance (C), interoperability (I), performance (P), and robustness (R). Each one has different testing goals (or objectives) and this is represented in Figure 2.

Basically, Figure 2 shows that when performing interoperability, performance, or robustness testing, the IUT should conform to the specification and also to specific testing goals defined by each type of testing. Diagnostic testing is not shown in the figure because it is often done in-house and is the first type of testing performed. In particular, robustness testing can also be considered when performing interoperability or performance testing.

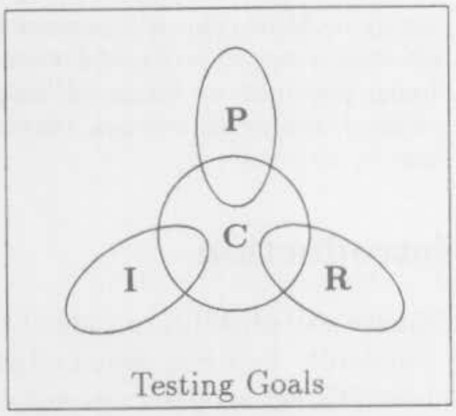


Figure 2: The different types of protocol testing and their goals.

According to the conformance testing methodology and framework [14] a pass verdict should be assigned to a test case if a given event e is valid for the current IUT state s. By valid we mean that e satisfies all testing requirements as defined in the specification. Unfortunately, there is an important class of errors, namely coordination loss,[1] [9, 13, 17, 19, 21] that cannot be

[1]Some errors that may cause coordination loss are inconsistent initialization, process failure and recovery, and memory crash. These errors are explained in more detail in Section 2.

anticipated because some source of error is *external* to both the tester and the implementation under test (IUT). Furthermore, it is not possible to simulate their occurrence exhaustively in the test environment. Therefore they are very difficult to catch in the testing phase. In this paper we assume that a message with an invalid checksum can be detected by a lower level protocol.

This is to be expected since the goal of conformance testing is not to catch this type of error. In fact coordination loss is a kind of problem that should be checked in robustness testing. In other words, there is a relationship between the faults remaining undetected and the type of protocol testing used.

In this case, protocol verification techniques are also not helpful. There is no procedure to check for errors caused by coordination loss since the fault model to be used during the verification process is not known in advance because we cannot precisely identify the possible consequences of coordination loss to the protocol.

Furthermore, most protocol specifications do not deal with all types of errors that may arise from coordination loss. Therefore, this class of errors should be addressed in the design phase.

These facts have some important consequences. Conformance testing is not enough from the point of view of reliability. In fact, only recently have people started to pay more attention to interoperability and performance testing. Ideally, all protocol implementations should be in conformity with their specifications, interoperate in an open system, have adequate performance characteristics, and be reliable.

In this paper we propose using self-stabilization as a design principle to overcome this testing problem. This will improve the reliability of protocol implementations derived from self-stabilizing protocol specifications. We present a novel algorithm and the corresponding design principles to design self-stabilizing protocols, give an example of a self-stabilizing protocol, and define two new relations based on an environment that exhibits coordination loss. The paper is organized as follows. Section 2 presents a brief overview of self-stabilization. Section 3 describes the formal model used in this paper. Section 4 presents a set of design principles to design self-stabilizing protocols. Section 5 gives an example of a protocol that is not self-stabilizing and its self-stabilizing version using the design principles described earlier. Section 6 presents a new conformance relation based on the external behavior. Section 7 discusses the related work. Finally, Section 8 presents the conclusions for this paper.

2 An overview of self-stabilization

In 1974 Dijkstra introduced the concept of self-stabilization [5]. Informally, a distributed system is called self-stabilizing iff it will converge automatically to a safe state in a finite number of steps regardless of the initial state. Of course the meaning of safe and unsafe[2] states depend on the specification of the system.

Self-stabilization makes the initialization of the system irrelevant. Therefore, if we are concerned about fault-tolerant issues, the property of self-stabilization guarantees the system to recover from an unsafe state caused by some perturbations to its normal operation. Some typical situations that might cause a coordination loss in a distributed system are:

- *Inconsistent initialization*—individual processes may be started up in states that are inconsistent with one another.
- *Transmission errors*—messages sent by a sender process may be lost, corrupted, reordered or delivered much later. In this situation the sender's state is no longer consistent with that of the receiver.
- *Reconfiguration errors*—systems may be reconfigured on-the-fly (e.g., addition or deletion of processes, nodes and links) and the new configuration may present an inconsistent view among the processes.

[2] Also called legal or legitimate, and illegal or illegitimate states respectively.

- *Mode change*—it is common to allow a system to operate in different modes depending on different factors such as number of users and load. Due to the distributed nature of the system, the processes may execute in different modes for some time. Eventually, all processes should converge to the same mode. If this is not the case, the state of the system has become unsafe.
- *Software failure*—if a piece of software (e.g., process or application) becomes temporarily unavailable, its local state may become inconsistent with that of the others when it resumes normal operation.
- *Hardware failure*—the same situation may happen to a piece of hardware such as memory or processor.

Self-stabilization is one of the principles of well-formed protocols. The property of self-stabilization provides a built-in safeguard against events that might corrupt the data. In practice, these events are very difficult to catch in the conformance testing process. The interested reader is referred to [20] for a survey on self-stabilization.

3 Formal model

In this section we present the communicating extended finite state machine (CEFSM) model used in describing both the communication protocols and the definition of self-stabilization. First we present the nomenclature for identifiers that will be used in this paper.

Nomenclature 1 (Identifiers) Identifiers can have a subscript and/or a superscript both of which are used to indicate an element in a set. Let $\square_i^x$ represent an identifier. The subscript i refers to the i-th element of $\square$. For example, process P_i. The superscript refers to the x-th element of $\square_i$ in the case $\square_i$ is also a set.

Definition 2 (Communicating extended finite state machines) A communicating extended finite state machine is a labeled directed graph where vertices represent states and edges represent transitions. A designated vertex represents the initial state of the machine. Transitions are labeled with the pair $\frac{event}{action}$. An event is the sending of a message, the reception of a message, or an internal event not related to a channel (e.g., a timeout or a service request from a service user). Messages should be defined in a finite set Σ of message types. The communication between machines is asynchronous. The communicating channels between each pair of machines are not perfect so that messages can be reordered, corrupted or lost. The channels are assumed to have infinite capacity.[3]

Formally, a communicating extended finite state machine P_i ($i = 1 \ldots r$) is a five-tuple

$$P_i = (S_i, \Sigma_i, \delta_i, \lambda_i, s_i^0),$$

where

- S_i is the set of local states of machine P_i.
- Σ_i is the set of message types that machine P_i can exchange with other machines. This is represented by the sets $\{\Sigma_{i,j}\}$ and $\{\Sigma_{j,i}\}$, respectively. Therefore, $\Sigma_i = \{\Sigma_{i,j}\} \cup \{\Sigma_{j,i}\}$. The set $\{\Sigma_{i,i}\}$ is empty, i.e.. machine P_i cannot send or receive messages from itself.
- δ_i is the transition function and is defined as δ_i: $S_i \times \Sigma_i \to S_i$.
- λ_i is the output function and is defined as λ_i: $S_i \times \Sigma_i - \Sigma_i$.
- s_i^0 is the initial state of machine P_i.

[3]In practice, we can model an infinite buffer using a finite buffer by discarding new incoming messages when the buffer is full.

In the labeled directed graph that represents a CEFSM, a message preceded by a + sign means that it was received, and a message preceded by a − sign indicates that it was sent. ■

Let P be a protocol specification comprised of processes $P_1, P_2, \ldots, P_r$. Each process is modeled as a CEFSM and they are interconnected by a set of communicating channels $C_1, C_2, \ldots, C_s$. Each process P_i $(i = 1 \ldots r)$ has a finite set of variables $V_i^1, V_i^2, \ldots, V_i^k$. In this model, the concept of a global state plays a fundamental role in the correctness of a communication protocol.

Definition 3 (Global state) The global state of protocol P is defined by the contents of each variable in each process, and the contents of each channel in the protocol. This can be expressed as:

$$\mathcal{S} = (V_1^1, \ldots, V_1^{k_1}) \times (V_2^1, \ldots, V_2^{k_2}) \times \ldots \times (V_r^1, \ldots, V_r^{k_r}) \times C_1 \times C_2 \times \ldots \times C_s$$ ■

On the other hand, the correctness of a protocol is expressed in terms of the global predicates.

Definition 4 (Global predicate) A global predicate represents certain system properties that should hold in all possible protocol computations. The global predicate[4] is defined in terms of global states. ■

Since there is no shared memory in the system where protocol P is running, the local variables of P_i are only updated by commands present in the protocol implementation of P_i. Furthermore, a send command in P_i (**send** $\langle m \rangle$ **to** P_j) appends message $\langle m \rangle$ to the tail of the messages in channel C_{ij}, and a receive command (**rcv** $\langle m \rangle$ **from** j) removes the head message $\langle m \rangle$ from the messages in channel C_{ji}.

Definition 5 (Protocol computation) A protocol computation is a sequence of global states. This can be expressed as:

$$\mathcal{C}_P = \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_t, \ldots$$ ■

Since protocols are reactive systems, the computation sequence $\mathcal{C}_P$ is, in general, infinite. In the case $\mathcal{C}_P$ is finite it means that no event/action is enabled in the last state, i.e., all processes are idle.

A property of a protocol is defined using global predicates $(R_1, R_2, \ldots)$ that involve global states. Two properties are particularly important for protocols:

- safety properties—defined by closed predicates, and
- progress properties—defined by a convergence relation over closed predicates.

Definition 6 (Closed predicate) A predicate R is called closed iff a state s^* and all other states thereafter that appear in $\mathcal{C}_P$ satisfy R. In this case, s^* and each subsequent state in $\mathcal{C}_P$ is called an R–state. If R and S are two closed predicates of protocol P then R converges to S iff for each possible protocol computation $\mathcal{C}_P$ that starts in an R–state there is a succeeding S-state. This is illustrated in Figures 3-(a) and 3-(b) respectively. ■

Note that in a protocol computation $\mathcal{C}_P$ each state is a *true*–state and no protocol state is a *false*–state. Therefore, both *true* and *false* are closed predicates in all protocols. This leads to the definition of self-stabilization.

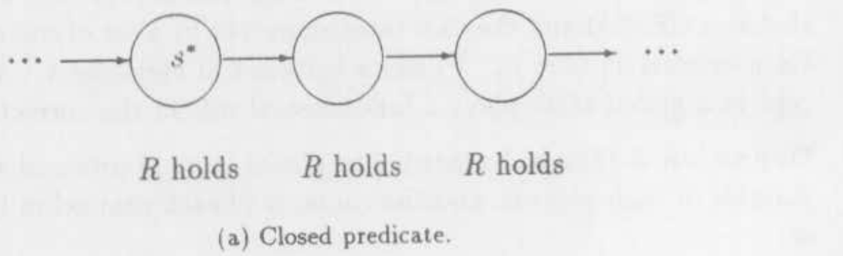
Definition 7 (Self-stabilizing protocol) A protocol P is said to be self-stabilizing iff given any closed predicate R, true converges to R. ■

As mentioned in the definition of global predicate. R should represent a correct behavior of P in all possible protocol computations.

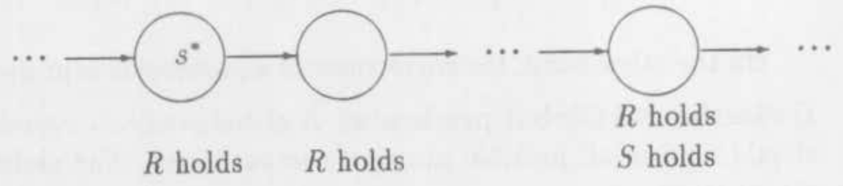
[4]In the remaining of this paper, the term *predicate* and *global predicate* will be used interchangeably.

4 Design of self-stabilizing protocols

In this section we present a set of novel design principles to incorporate self-stabilization into a protocol specification. These design principles seem to be general enough to apply to a large class of communication protocols as we will see in Section 5. To the best of our knowledge there is no work reported in the literature that transforms a given non self-stabilizing protocol into a self-stabilizing protocol in a systematic way. See, for instance, the survey presented by Schneider [20] and the related work in Section 7. The main contribution of this paper is to present a set of design principles to create self-stabilizing protocols in a systematic way.



(a) Closed predicate.



(b) Convergence relation.

Figure 3: Example of closed predicates.

The design of self-stabilizing protocols can be divided into two steps: (i) definition of some elements related to the protocol specification, and (ii) application of the algorithm to introduce the self-stabilizing features into the protocol specification. As we will see, these steps define a logical sequence of activities that will end with the stabilization proof of the communication protocol.

In the following each step is discussed in detail.

4.1 Elements related to the protocol specification

The elements discussed in this section will be used in Section 4.2 when introducing the self-stabilizing features into the communication protocol.

4.1.1 Formal model

Each protocol entity should be defined by a communicating extended finite state machine model where the basic elements are states, messages, transitions between states according to some rules defined in the protocol, and the initial state. In particular, the set of messages should be divided into two sub-sets: one contains the messages that can be sent and the other contains the messages that can be received. The communicating channels should also be specified.

The self-stabilizing "features" will be added to the CEFSM model as explained in Section 4.2. In the case of the example given in Section 5 we will present the code for the protocol that corresponds to the CEFSM model so it is possible to have a more concrete idea of how the self-stabilizing features are introduced in an implementation.

4.1.2 Type of communication among entities

Basically, peer entities can communicate one-to-one, one-to-many, many-to-one, and many-to-many. The latter three cases are called group communication. Furthermore, each entity may or may not be autonomous in initiating a communication. A non-autonomous entity can only initiate a communication in response to a message received.

The type of communication among the peer entities should be specified.

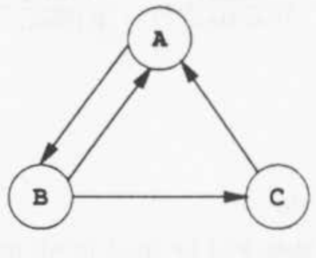
4.1.3 Timeout actions

Identify, if any, the timeout actions present in all states indicating the message associated with the timeout.

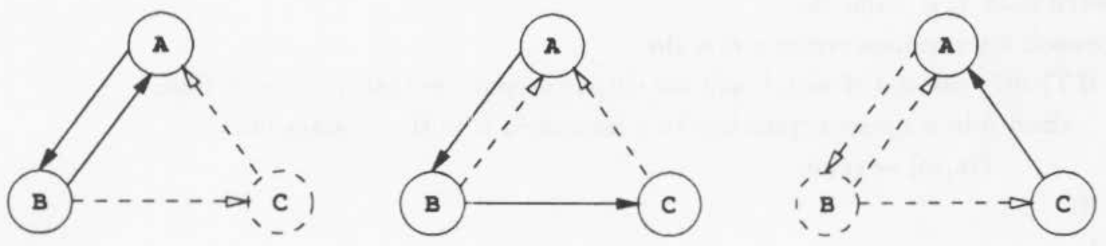
4.2 Algorithm to introduce the self-stabilizing features into the protocol specification

In the algorithm below we use the concepts of phases and paths. Typically a protocol behavior can be partitioned into a number of distinct phases, each one responsible for a specific task. Examples of phases include connection establishment, transfer of data, and orderly termination of the connection. Furthermore, each protocol phase has a set of messages and associated rules for interpreting them. In general, there is only one initial state associated with each protocol phase. Once that initial state is reached, it is assumed that the function performed by the previous phase is completed and the protocol is ready to execute a new function. Of course, this is not valid for the very first phase of a protocol when it starts execution.

It is common to have in each phase two or more distinct paths that reflect the possible outcomes when the phase is executed. A path is defined by the sequence of states traversed in a phase. Some of the paths may return to the initial state of that phase, others may go to the beginning of other phases. In Figure 4-(a) a CEFSM with three states that represent part of a protocol is shown. Suppose there are two phases α and β with initial states A and C, respectively. The paths in this automaton are A–B–A, A–B–C, and C–A as shown by solid lines in Figure 4-(b). The first path stays in phase α, the second moves from phase α to β, and the third moves from phase β to α.



(a) Partial CEFSM of a protocol.



(b) Distinct paths (shown by solid lines).

Figure 4: Example of different paths in a phase.

The algorithm to introduce the self-stabilizing features into the communication protocol is presented in Figure 5. Recall from Definition 2 that the CEFSM is a labeled directed graph where vertices and edges represent states and transitions respectively. The algorithm uses the concept of an intermediate vertex, i.e., a vertex in a path except the initial and the last ones.

The algorithm is divided in three parts:

- Lines 2–6: generates all paths in the CEFSM (set *Paths*) and identifies the last transition in each path (set *LTP*).
- Lines 7–9: introduces the lock-step mode principle as explained in Section 4.2.1.
- Lines 10–17: introduces timeout actions in each path as explained in Section 4.2.2.

Begin of algorithm to introduce self-stabilizing features

Input:
- Directed graph $G = (V, E)$ representing the CEFSM. The set V represents the vertices and the set E the transitions that have the form $\frac{event}{action}$.
- Set $IS = \{v_1, v_2, \ldots, v_p\}$ of vertices that represent the initial state of each phase in the protocol.
- Matrix $T = [1..|V|, 1..|\Sigma_{output}|]$ indicates whether a given vertex has a timeout action associated with the reception of a particular message. The set Σ_{output} represents the set of messages that the machine can send to other machines. The entry $T[i, j]$ is true if vertex v_i has a timeout action associated with message $m_j \in \Sigma_{output}$. Otherwise it is false.

Output:
- Directed graph representing the CEFSM with the self-stabilizing features.

```
(1)  $Paths \leftarrow \{\}$;    $LTP \leftarrow \{\}$;
(2)  foreach vertex $v_i \in IS$ do
(3)    Find all paths $\pi_i = v_i, v_{i+1}, \ldots, v_l$ in the graph $G$ that start with $v_i$ and finish with $v_l$, where
       $v_l \in IS$, such that the only two vertices in the path that are in $IS$ are $v_i$ and $v_l$.
       /* The path $\pi_i$ can be expressed as a sequence of vertices and edges. Re-writing $\pi_i$ in terms
       of states and transitions we have:
            $\pi_i = v_i \frac{event_i}{action_i} v_{i+1} \frac{event_{i+1}}{action_{i+1}} \cdots \frac{event_{l-1}}{action_{l-1}} v_l$        */
(4)    $Paths \leftarrow Paths \cup \pi_i$ ;
(5)    $LTP \leftarrow LTP \cup \frac{event_{l-1}}{action_{l-1}}$ ;
(6)  od;
(7)  foreach pair $\frac{event}{action} \in LTP$ do
(8)    Generate a new identifier that will be used in all messages in the next path. The generation
       of the new id should be done as part of the action executed in this transition.
(9)  od;
(10) foreach path $\pi_i \in Paths$ do
(11)   foreach intermediate vertex $v \in \pi_i$ do
(12)     if $T$[entry associated with $v$ and the output message $m$ that led to $v$] = false
(13)       then Add a timeout transition to $v$ associated with the message $m$.
(14)            $T[v, m] \leftarrow$ true;
(15)     fi;
(16)   od;
(17) od;
```

End of algorithm to introduce self-stabilizing features

Figure 5: Algorithm to introduce self-stabilization features into a CEFSM.

4.2.1 Lock-step mode principle

At each moment in time a process is executing one of the paths in the associated CEFSM. As discussed above, each path represents a possible outcome in the current phase. In order to guarantee coordination among the peer entities, the processes should work in lock-step. Intuitively, this means that the computations among the peer entities should progress together. path-by-path. In fact, this would happen if there were no errors at all in the environment and the protocol is designed with some "nice" properties (e.g., safety and liveness).

To guarantee that the protocol will work in lock-step, each path executed along a computation has associated with it a unique identifier that is monotonically increasing. In this way, each new path executed in each phase by a process can be differentiated from the previous one. The lock-step mode is effectly achieved when all messages exchanged among the peer entities for each path carry the path's identifier. With this mechanism each process knows whether the message is valid for that path or not. If it is not it should be discarded since it does not belong to the current path but to another one that does not exist anymore.

This is an important aspect of our self-stabilization technique. Note that the identifier can be implemented using a timestamp mechanism [16]. This guarantees that the identifiers are monotonically increasing.

The next question is where and when to assign a new identifier for a path. Note that each path $\pi_i \in Paths$ can be represented by the following sequence of vertices (states) and edges (transitions):

$$\pi_i = v_i \frac{event_i}{action_i} v_{i+1} \frac{event_{i+1}}{action_{i+1}} \dots \frac{event_{l-1}}{action_{l-1}} v_l.$$

The final state (i.e., vertex) will be the initial state of the current or another phase. Therefore, the right time and place to create a new identifier is when the final state of a phase is reached, i.e.. in the transition that leads to v_l. This transition is labeled with the pair $\frac{event_{l-1}}{action_{l-1}}$. Therefore, a new identifier should be generated as part of the action $action_{l-1}$ executed in this transition.

There is also another important aspect related to the generation of identifiers. In Section 4.1.2 we classified processes as autonomous or non-autonomous with respect to the communication among entities. Non-autonomous processes cannot initiate a communication with a peer entity and therefore play a "minor" role in self-stabilization. Typically, their self-stabilization version is obtained by copying the message identifier in a request message to the response message. Autonomous processes are responsible for implementing the lock-step execution mode.

4.2.2 Timeout actions

Note that the lock-step execution mode by itself is not sufficient to guarantee self-stabilization. Recall from Section 2 that a self-stabilizing protocol will converge automatically to a safe state in a finite number of steps. Therefore, we need a mechanism to guarantee the convergence of the protocol to a safe state. The lock-step principle guarantees that only valid messages are accepted and processed by the protocol. But it does not guarantee that the protocol moves along a given path when it is in an unsafe state. The following theorem guarantees the progress of a protocol execution and thus its convergence to a safe state.

Let s_k be an intermediate state in a path and message m the action associated with the transition from state s_{k-1} to s_k, i.e., $s_{k-1} \xrightarrow{-m} s_k$.

Theorem 8 An intermediate state s_k in a path should have a timeout action associated with the output message m that led to that state.

Proof (Contradiction). Suppose that an intermediate state s_k in an autonomous process P does not have a timeout action associated with message m sent to the peer process Q. If process Q is in a state that will not send any reply message then process P can stay in state s_k forever, and thus s_k becomes an unsafe state. Therefore timeout actions should be added for all intermediate states s_k in a path. □

This theorem is based on two assumptions. First, each transition to an intermediate state has the form $\frac{event_{k-1}}{-(msg_{k-1})}$, i.e., an intermediate state is reached by sending a message. Second, once a new phase is reached the function performed by the previous phase has been completed and the protocol is ready to execute a new function. This is the reason for not including timeout actions in the initial states of each phase. Let us examine this theorem if the assumptions are removed.

If we lift the first assumption given above then an intermediate state s_k can be reached after executing a transition not involving an action related to the communication channel. There are only two possible events that can make the machine move from state s_k to another state: either an event related to the communication channel or a local event such as a request from the service user. In both cases we must have a timeout action associated with this event. Otherwise we have the same situation described in the proof of Theorem 8.

If we lift the second assumption above it means that there are overlapping phases in the communication protocol with common intermediate states. This is not a desirable characteristic in the protocol design since two distinct functions are mixed together. In practice, protocols are not designed with overlapping phases. If they do exist then Theorem 8 is still valid and must also be applied to a state that is at the same time the end state of a phase and an intermediate state of another phase.

Note that if we keep the second assumption the following theorem holds.

Theorem 9 Every initial state of a phase is a safe state.

Proof (Induction). Let set $IS = \{s_1, s_2, \ldots, s_p\}$ be the states that represent the initial state of each phase in the protocol. State s_1 is the initial state of the protocol when it starts to execute. Let $C_P = s_1, \ldots, s_i, \ldots, s_j, \ldots, s_k, \ldots$ represent the states in a protocol computation of process P where only the states in set IS are shown. Clearly, s_1 is a safe state. Let us assume that state s_j is a safe state and the next state that appears in the computation from the set IS is s_k. If no perturbation occurs in the system between states s_j and s_k, then state s_k is also a safe state according to the assumptions. If a perturbation occurs then the computation will progress until state s_k is eventually reached. This is guaranteed by the timeout actions. Furthermore, only valid messages will be accepted and processed by P because of the lock-step execution mode. Therefore, state s_k is also a safe state. □

A final remark about timeout actions is that they do not apply to non-autonomous processes since they cannot initiate a communication.

4.2.3 Complexity of the algorithm

As mentioned earlier, the algorithm has three sequential parts. The first part (lines 2–6) generates all the paths in the graph. This can be done using a breadth-first search algorithm which can be carried out in time $O(v + e)$. The second part (lines 7–9) introduces the lock-step mode principle in each transition of the set LTP. This is clearly bounded by $O(v + e)$ which is the time required to find all the paths. The third part (lines 10–17) introduces the timeout actions along each path. Line 10 is bounded by $O(v + e)$, line 11 by $O(e)$ and lines 12 to 14 are executed in constant time. Thus lines 10 to 17 are bounded by $O(v + e) \times O(e)$ which gives $O(ve + e^2)$. Therefore, the three parts together are bounded by a quadratic function.

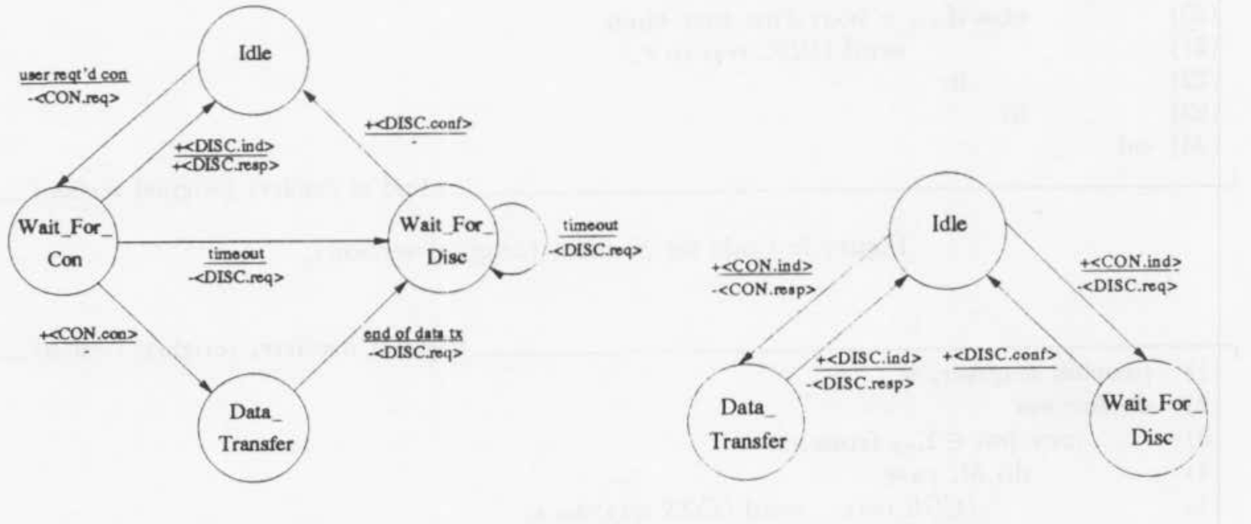
If we execute the part associated with the timeout actions when we are generating each path (the first part) we can avoid the cost incurred by the third part and thus the algorithm can be carried out in time $O(v+e)$. The algorithm was presented in three parts for didactical purposes but in a real implementation an optimization like this should be used.

5 Self-stabilization: An example

In this section we present a simple protocol for connection management (CM). This protocol is comprised of the connection and disconnection phases. We use this protocol as an example in this paper because it is present in all connection-oriented protocols and has been widely used. Some of the protocols that use this kind of connection management are the protocols defined for the OSI stack such as the data link, network, transport and session layers, TCP, and protocols for high speed networks such as XTP [21] and NETBLT [4].

5.1 Description of the connection management protocol

In the connection management protocol both the connection and disconnection phases use a confirmed service. The communicating finite state machines for both processes are shown in Figure 6.



(a) Sender process. (b) Receiver process.

Figure 6: Connection management protocol–CEFSM model.

The code for processes $Sender_s$ and $Receiver_r$ are given in Figures 7 and 8 respectively.

5.2 Elements related to the protocol specification

In the following we present the elements related to the protocol specification that will be used in the self-stabilizing version of the protocol as discussed in Section 4.1.

5.2.1 Formal model

In the following we identify the set of states and messages valid for both $Sender_s$ and $Receiver_r$, and their communicating channels.

Let S_s indicate the state of $Sender_s$ based on the status of its communicating channel. The states are:

- IDLE: the channel is idle and available for use. No connection is established between $Sender_s$ and $Receiver_r$.

Begin of *Sender$_s$* (original version)

```
process $Sender_s$ ≡
do forever
      ($S_s$ = IDLE ∧ $user\ requests\ connection$) ⟼
            send ⟨$CON.req$⟩ to r;
            $S_s$ ← WAIT_FOR_CON;
      ($S_s$ = DATA_TRANSFER ∧ $data\ transfer\ is\ over$) ⟼
            send ⟨$DISC.req$⟩ to r;
            $S_s$ ← WAIT_FOR_DISC;
      (rcv ⟨$m$⟩ ∈ $\Sigma_{r,s}$ from $r$) ⟼
            do $M_r$ case
                  ⟨$CON.conf$⟩: $S_s$ ← DATA_TRANSFER;
                  ⟨$DISC.ind$⟩: send ⟨$DISC.resp$⟩ to $r$;
                                $S_s$ ← IDLE;
                  ⟨$DISC.conf$⟩: $S_s$ ← IDLE;
            od;
      (timeout($S_s$ = WAIT_FOR_CON ∨ WAIT_FOR_DISC)) ⟼
            if $S_s$ = WAIT_FOR_CON then
                  send ⟨$DISC.req$⟩ to $r$;
                  $S_s$ ← WAIT_FOR_DISC;
            else if $S_s$ = WAIT_FOR_DISC then
                        send ⟨$DISC.req$⟩ to $r$;
                  fi;
            fi;
od;
```

End of *Sender$_s$* (original version)

Figure 7: Code for *Sender$_s$* (original version).

Begin of *Receiver$_r$* (original version)

```
process $Receiver_r$ ≡
do forever
      (rcv ⟨$m$⟩ ∈ $\Sigma_{s,r}$ from $s$) ⟼
            do $M_s$ case
                  ⟨$CON.ind$⟩:  send ⟨$CON.resp$⟩ to $s$;
                                $S_r$ ← DATA_TRANSFER;
                                      ∨
                                send ⟨$DISC.req$⟩ to $s$;
                                $S_r$ ← WAIT_FOR_DISC;
                  ⟨$DISC.conf$⟩: $S_r$ ← IDLE;
                  ⟨$DISC.ind$⟩:  send ⟨$DISC.resp$⟩ to $s$;
                                $S_s$ ← IDLE;
            od;
od;
```

End of *Receiver$_r$* (original version)

Figure 8: Code for *Receiver$_r$* (original version).

- WAIT_FOR_CON: *Sender$_s$* sent a ⟨*CON.req*⟩ (connection request) to *Receiver$_r$* and is waiting for a response.
- WAIT_FOR_DISC: *Receiver$_r$* sent a ⟨*DISC.req*⟩ (disconnection request) to *Receiver$_r$* and is waiting for a response.
- DATA_TRANSFER: a connection between *Sender$_s$* and *Receiver$_r$* is established and they can start transferring data.

The set of valid messages for $Sender_s$ is $\Sigma_s = \{\langle CON.req\rangle, \langle CON.conf\rangle, \langle DISC.req\rangle, \langle DISC.ind\rangle, \langle DISC.conf\rangle\}$. Let $\Sigma_{r,s}$ indicate the set of valid messages that $Sender_s$ can receive from $Receiver_r$:

- $\langle CON.conf\rangle$: receiver accepted the request made by the sender.
- $\langle DISC.ind\rangle$: receiver rejected the request made by the sender.
- $\langle DISC.conf\rangle$: receiver acknowledged the disconnection request made by the sender.

The set of valid messages that $Sender_s$ can send to $Receiver_r$ is explained in the receiver part since the protocol CM uses a confirmed service.

Let S_r indicate the state of $Receiver_r$. The states are:

- IDLE: the channel is idle and available for use. No connection is established between $Sender_s$ and $Receiver_r$.
- WAIT_FOR_DISC: $Receiver_r$ sent a $\langle DISC.req\rangle$ (disconnection request) to $Sender_s$ and is waiting for a response.
- DATA_TRANSFER: a connection between $Sender_s$ and $Receiver_r$ is established and they can start transferring data.

Note that WAIT_FOR_CON is not a valid state for $Receiver_r$ since upon receipt of a $\langle CON.ind\rangle$ the receiver goes to either the WAIT_FOR_DISC or DATA_TRANSFER state.

The set of valid messages for $Receiver_r$ is $\Sigma_r = \{\langle CON.ind\rangle, \langle CON.resp\rangle, \langle DISC.req\rangle, \langle DISC.ind\rangle, \langle DISC.resp\rangle, \langle DISC.conf\rangle\}$. Let $\Sigma_{s,r}$ indicate the set of valid messages that $Receiver_r$ can receive from $Sender_s$:

- $\langle CON.ind\rangle$: sender requested a connection establishment.
- $\langle DISC.conf\rangle$: receiver rejected the sender's request for a connection establishment and sent a $\langle DISC.req\rangle$ to $Sender_s$.
- $\langle DISC.ind\rangle$: the sender wants to disconnect.

Let C_{sr} indicate the set of messages sent from the sender to the receiver. Similarly, let C_{rs} indicate the set of messages sent from the receiver to the sender.

5.2.2 Type of communication among entities

Process $Sender_s$ is autonomous in initiating a communication with $Receiver_r$ whereas the other way around is not possible. $Receiver_r$ only reacts to requests sent by $Sender_s$.

5.2.3 Timeout actions

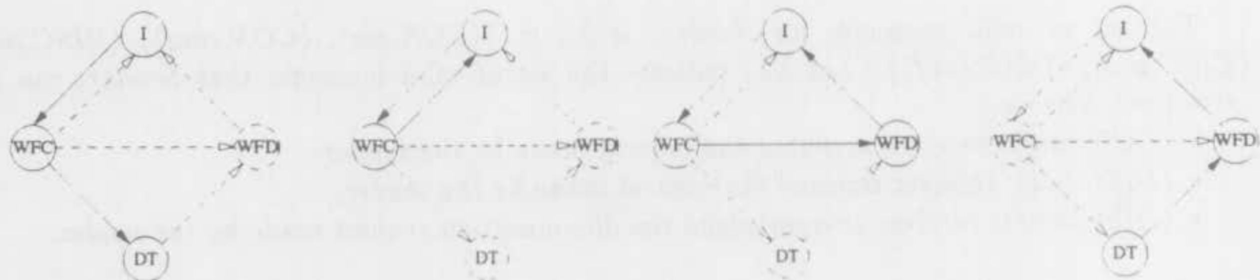
Process $Sender_s$ has timeout actions in the following states:

- WAIT_FOR_CON: associated with a response to the message $\langle CON.req\rangle$.
  In this case $Sender_s$ can receive either a $\langle CON.conf\rangle$ or a $\langle DISC.ind\rangle$.
- WAIT_FOR_DISC: associated with message $\langle DISC.req\rangle$.
  In this case $Sender_s$ can only receive a $\langle DISC.conf\rangle$.

Process $Receiver_r$ does not have any timeout actions since it is not autonomous.

5.3 Applying the algorithm to introduce the self-stabilizing features into the protocol specification

The three parts of the algorithm shown in Figure 5 are given below.



(a) Successful connection. (b) Rejection. (c) Unsuccessful connection. (d) Disconnection.

Figure 9: Distinct paths in the connection management protocol.

5.3.1 Phases and paths

Since process *Receiver*$_r$ may only react to messages sent by *Sender*$_s$, this element is relevant to *Sender*$_s$ only.

The CM protocol as shown in Figure 6 is comprised of the connection and disconnection phases and they are treated together in the CEFSM model. The connection phase starts in the state IDLE, and the disconnection phase in DATA_TRANSFER. Therefore, the set of initial states can be defined as $IS = \{\text{IDLE}, \text{DATA\_TRANSFER}\}$. Note that the state DATA_TRANSFER is the initial state for the data transfer phase which is not shown in the figure.

In the case of connection establishment there are three possible outcomes: successful connection, connection rejection, and unsuccessful connection. These possibilities together with the disconnection case define four distinct paths in the graph that represents the CM protocol:

1. Successful connection (Figure 9-(a))—*Receiver*$_r$ accepted a connection request and both entities are ready to start transmitting data.

2. Rejection (Figure 9-(b))—*Receiver*$_r$ rejected a connection request.

3. Unsuccessful connection (Figure 9-(c))—*Sender*$_s$ was not able to establish a connection with *Receiver*$_r$.

4. Disconnection (Figure 9-(d))—there was no more data to be transferred to *Receiver*$_r$ and *Sender*$_s$ disconnected.

The set *LTP* (last transition of the path) contains three transitions:

1. $\frac{+\langle CON.conf\rangle}{-}$ for the successful connection path.

2. $\frac{+\langle DISC.ind\rangle}{-\langle DISC.resp\rangle}$ for the rejection path.

3. $\frac{+\langle DISC.conf\rangle}{-}$ for the unsuccessful connection and disconnection paths.

5.3.2 Lock-step mode

The lock-step mode will be introduced in process *Sender*$_s$ since it alone can initiate a communication. The variable N_s contains the identifier to be associated with each new path, i.e., N_s is responsible for implementing the timestamp mechanism. The following lines were added to the code of *Sender*$_s$ as shown in Figure 10:

- (10)-(13): to discard any message that does not belong to the current path.

- (17): lock-step mode for the successful connection case.
- (22): lock-step mode for the connection rejection case.
- (26): lock-step mode for the unsuccessful connection and disconnection cases.

All **send** and **rcv** commands in both $Sender_s$ and $Receiver_r$ have a new variable that contains the path id as shown in Figures 10 and 11, respectively. Note that at any moment in time, a valid message in the channel has an id N, where $N_s - 1 \leq N \leq N_s$. The messages have the id $N = N_s$ when $Sender_s$ is in any state in a path except the last one. When the last state is reached, N_s is incremented and we have $N = N_s - 1$.

At any moment in time, $Sender_s$ is executing *only* one the following actions (given by the line numbers):

- ≪*ConAction*≫ (3)–(5): user requested a connection.
- ≪*DiscAction*≫ (6)–(8): user requested a disconnection.
- ≪*RespAction*≫ (9)–(28): $Sender_s$ received a response.
- ≪*TimeAction*≫ (29)–(36): a timeout occured.

On the other hand, $Receiver_r$ has just one action to execute:

- ≪*ReplyAction*≫ (3)–(12): $Receiver_r$ received a request and should reply to it.

These actions are marked along the line numbers in the code.

Figure 12 shows the self-stabilizing version of the communicating extended finite state machines for both processes.

5.3.3 Timeout actions

The four paths shown in Figure 9 with their respective states are:

1. Successful connection: I → WFC → DT.
2. Connection rejection: I → WFC → I.
3. Unsuccessful connection: I → WFC → WFD → I.
4. Disconnection: DT → WFD → I.

All four paths have intermediate states and therefore for each one of them we need to check whether it has a timeout action associated with the message that led to that state. The first three paths have the intermediate state WFC (wait for connection) which has a timeout action associated with ⟨*CON.req*⟩. This is the message that led the machine to the state WFC in the three cases. Therefore, we do not need to add any new timeout action. The last two paths above have the intermediate state WFD (wait for disconnection) which has a timeout action associated with ⟨*DISC.req*⟩. This is the message that led the machine to the state WFD in both cases. Therefore no timeout action is needed here as well.

5.4 Applying a proof technique to verify the self-stabilization

Gouda and Mutari [9] present a proof technique called convergence stair to show whether a protocol is self-stabilizing for some closed predicate R. The input to the proof technique is the self-stabilizing version P_{SS} of the original protocol P. Therefore, the main problem is to come up with a self-stabilizing version P_{SS} of P such that in the presence of perturbations the system will converge to a safe state in a finite number of steps where R holds again. This can be accomplished by applying the design principles presented in Section 4. Due to space limitations we will not present how the the proof technique described in [9] can be applied to show that the self-stabilizing version of the connection management protocol is correct indeed.

Begin of *Sender*$_s$ (self-stabilizing version)

```
process $Sender_s$ ≡
do forever
    ($S_s$ = IDLE ∧ user requests connection) ⟼
        send ⟨CON.req,$N_s$⟩ to $r$;
        $S_s$ ← WAIT_FOR_CON;
    ($S_s$ = DATA_TRANSFER ∧ data transfer is over) ⟼
        send ⟨DISC.req,$N_s$⟩ to $r$;
        $S_s$ ← WAIT_FOR_DISC;
    (rcv ⟨$m \in \Sigma_{r,s}$, $N$⟩ from $r$) ⟼
        if $N \neq N_s$ then
            discard message;
            continue;        /* goes to next iteration */
        fi;
        do $M_r$ case
            ⟨CON.conf⟩: if $S_s$ = WAIT_FOR_CON then
                            $S_s$ ← DATA_TRANSFER;
                            $N_s \leftarrow N_s + 1$;
                        fi;
            ⟨DISC.ind⟩: if $S_s$ = WAIT_FOR_CON then
                            send ⟨DISC.resp,$N_s$⟩ to $r$;
                            $S_s$ ← IDLE;
                            $N_s \leftarrow N_s + 1$;
                        fi;
            ⟨DISC.conf⟩: if $S_s$ = WAIT_FOR_DISC then
                            $S_s$ ← IDLE;
                            $N_s \leftarrow N_s + 1$;
                        fi;
        od;
    (timeout($S_s$ = WAIT_FOR_CON ∨ WAIT_FOR_DISC)) ⟼
        if $S_s$ = WAIT_FOR_CON then
            send ⟨DISC.req,$N_s$⟩ to $r$;
            $S_s$ ← WAIT_FOR_DISC;
        else if $S_s$ = WAIT_FOR_DISC then
                send ⟨DISC.req,$N_s$⟩ to $r$;
            fi;
        fi;
od;
```

End of *Sender*$_s$ (self-stabilizing version)

Figure 10: Code for *Sender*$_s$ (self-stabilizing version).

5.5 Some remarks on the self-stabilizing version

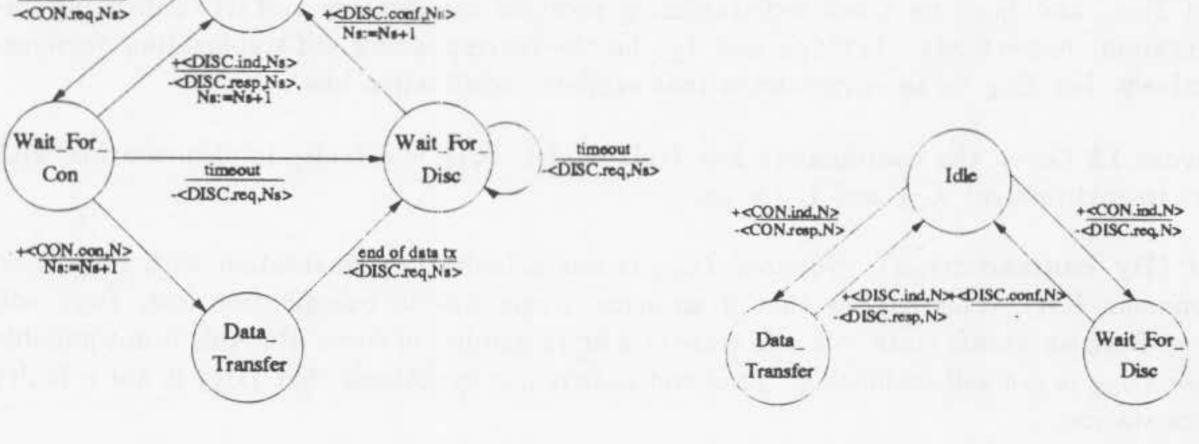
In the solution presented, the variable N_s is unbounded. There are two possible solutions to this problem. One may consider that a large variable with 48 or 64 bits is unbounded for practical purposes, or one may use an aperiodic sequence (e.g., a random sequence that is easy to generate) so each path starts with a different sequence number. The latter alternative is called pseudo-stabilization [2]. In practice, these solutions satisfy the needs of most applications.

One particular point related to this protocol is that process *Sender*$_s$ plays an active role in the self-stabilization process. *Receiver*$_r$ plays a passive role since it only responds to *Sender*$_s$. Finally, the following theorem is easily proven.

Theorem 10 The number of rounds (in this case receive messages in *Sender*$_s$) necessary to bring the protocol CM to a safe state when there are m messages in the system is bounded by $\Theta(m)$. □

```
Begin of $Receiver_r$ (self-stabilizing version)
process $Receiver_r \equiv$
do forever
    (rcv $\langle m \in \Sigma_{s,r}, N \rangle$ from $s$) $\longmapsto$
        do $M_s$ case
            $\langle CON.ind \rangle$:  send $\langle CON.resp, N \rangle$ to $s$;
                                 $S_r \leftarrow$ DATA_TRANSFER;
                              $\vee$
                                 send $\langle DISC.req, N \rangle$ to $s$;
                                 $S_r \leftarrow$ WAIT_FOR_DISC;
            $\langle DISC.conf \rangle$: $S_r \leftarrow$ IDLE;
            $\langle DISC.ind \rangle$:  send $\langle DISC.resp, N \rangle$ to $s$;
                                 $S_r \leftarrow$ IDLE;
        od;
od;
End of $Receiver_r$ (self-stabilizing version)
```

Figure 11: Code for $Receiver_r$ (self-stabilizing version).



(a) Sender process. (b) Receiver process.

Figure 12: Self-stabilizing version–CEFSM model.

6 A new conformance relation based on the external behavior

This section is related to the theory of testing and is motivated by the design principles described in Section 4.

In order to improve the confidence in the implementation under test, it is necessary to define a formal notion of conformance that relates descriptions at different levels of abstraction. In particular, we are interested in two descriptions: specification and implementation. Informally, conformance can be defined as follows:

Definition 11 (Conformance relation) Protocol description P_1 conforms to protocol description P_2 (expressed as P_1 **conf** P_2) iff for all possible environments E in which P_1 and P_2 can run, all behaviors of P_1 in E observed at the external interaction points[5] are possible

[5]In the OSI context, interaction points (IPs) represent service access points, and points of control and observation.

when P_2 is run in the same environment.

From the point of view of protocol engineering, this is a strong definition which is difficult to realize. It is not practical because the set of possible environments can be very large and hard to be anticipated.

Despite this, several conformance relations that are environment-independent have been proposed in the literature. Some of them are:

- Observational equivalence between CCS processes [18].
- Correctness between a concurrent program and a formula of temporal logic [10].
- The *satisfy*-relation between a CSP process and a formula of trace logic [11].
- The *conf*-relation and testing equivalence between Lotos processes [1]

As pointed out by Gotzhein [8], these relations do not necessarily imply conformance as given in Definition 11. Gotzhein describes some problems that arise when different semantics for interaction points are considered in an environment-independent relation.

If we consider an environment that may cause coordination loss and the specification is not designed to handle coordination loss, then the relations above do not help either. Therefore, if the environment is known in advance, a more specific conformance relation can be given. Since coordination loss is a common problem in practice [9, 13, 17, 19, 21], it is quite reasonable to take it into consideration in defining conformance relations. This leads to the following theorem.

Let S_{NSS} and I_{NSS} be a non self-stabilizing protocol specification and its conforming implementation, respectively. Let S_{SS} and I_{SS} be the corresponding self-stabilization versions, respectively. Let E_{CL} be an environment that exhibits coordination loss.

Theorem 12 Given the coordination loss fault model, I_{NSS} is a faulty implementation with respect to environment E_{CL} and I_{SS} is not.

Proof (By contradiction). Suppose I_{NSS} is not a faulty implementation with respect to environment E_{CL}. This implies that if an error occurs due to coordination loss, I_{NSS} will converge from an unsafe state to a safe state in a finite number of steps. But this is not possible because I_{NSS} is not self-stabilizing. That contradicts our hypothesis that I_{NSS} is not a faulty implementation. □

Figure 13 shows the relationships among the four versions of specifications and implementations.

Despite the fact that I_{NSS} **conf** S_{NSS}, we can see that I_{NSS} $\neg$**conf** S_{SS} and I_{SS} $\neg$**conf** S_{NSS}.

Note that Theorem 12 does not say anything directly about the specifications. But since the implementations conform to their specifications, it means that the specification must not accept a faulty behavior related to coordination loss.

This leads us to the following conformance definition that takes into consideration an environment that may cause coordination loss:

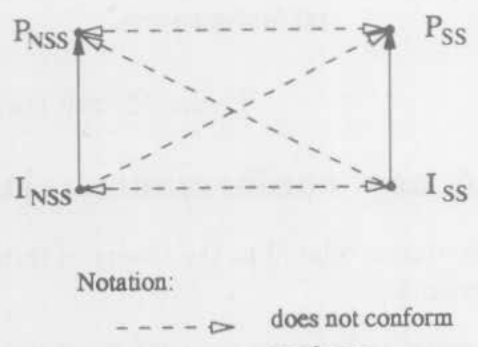


Figure 13: Relationships among S_{NSS}, I_{NSS}, S_{SS}, and I_{SS}.

Definition 13 (Conformance relation conf$_{CL}$) Protocol description P_1 conforms to protocol description P_2 iff for all possible environments that may cause coordination loss (E_{CL}), all behaviors of P_1 in E_{CL} observed at the external interaction points are possible when P_2 is placed in the same environment. Furthermore, any error caused by coordination loss is not valid for P_2. This is expressed as P_1 **conf**$_{CL}$ P_2.

The testing problem discussed in Section 1 prompts us to compare the testability of a protocol specification S after embedding its implementation I in environment E_{CL}, and testing I directly. Intuitively, testing through an environment degrades testability. This is another way of interpreting Theorem 12. Therefore nothing can be said about the capacity of the testing process in detecting faulty implementations in an arbitrary environment. This is strongly dependent on S and its implementation I, and on the environment E used for testing.

A natural way of comparing the testability of two implementations in this context is through the relation "more testable" with respect to environment E_{CL}. This is given in the following definition where I_1 and I_2 are implementations of a protocol specification S. Note that E_{CL} defines a fault model.

Definition 14 (More testable relation) An implementation I_1 is more testable than implementation I_2 for environment E_{CL} (expressed as $I_1 \sqsubseteq_{E_{CL}} I_2$) if I_1 does not contain any errors that I_2 may have with respect to environment E_{CL} (i.e., the errors that E_{CL} may cause). ∎

Clearly, we have $I_{SS} \sqsubseteq_{E_{CL}} I_{NSS}$ for Theorem 12 since all errors found in I_{NSS} due to coordination loss will not be present in I_{SS} and therefore I_{SS} is more testable than I_{NSS}. In practice, this relation can be verified if test cases are generated to test all the errors caused by coordination loss, and the tester program is capable of leading the IUT to a state where these errors can be detected. This is difficult to achieve in a conformance testing environment as explained in Section 1. Therefore this relation can be re-phrased as follows: implementation I_1 is more *reliable* than implementation I_2 with respect to the errors caused by coordination loss if I_1 does not contain any errors that I_2 may have with respect to this fault model.

7 Related work

Gotzhein [8] defines a "compatibility relation" between external interaction points in a system. This relation is used to overcome the problem of defining conformance for an unknown environment. Gotzhein defines several properties that interaction points can have. One of the properties defined prevents duplication, corruption, and creation of interactions (messages). This property can be guaranteed for an interaction point. Unfortunately it cannot be extended to cover different environments where a protocol may be executed. Therefore the problem still remains.

Drira et al. [6, 7] propose a method for analyzing the testability of an implementation I with respect to the verdict of the test execution when I is tested through an environment. They assume that a correct verdict can be assigned to the implementation when I is tested directly when performing conformance testing. Although, they point out that for robustness testing the problem remains. This problem is solved if we apply the design principles described in this paper.

There are at least two specifications of protocols for high speed networks that try to circumvent the problem of coordination loss by using specific mechanisms. Delta-t [22] uses a time mechanism to guarantee that the lifetime of each packet is bounded and strictly enforced. Sabnani and Netravali [19] propose a transport protocol where protocol entities exchange the full local state periodically independent of changes in the states of the cooperating entities.

Katz and Perry [15] propose a mechanism to create a self-stabilization extension of a distributed program P. The idea is to superimpose onto P a self-stabilizing global monitor that repeatedly performs the following three steps: (i) take snapshots of the global state, (ii) verify whether the snapshots indicate an unsafe global state, and (iii) reset the variables of each process to a safe state when a problem is detected. Of course, each one of these steps "must function correctly no matter what the initial state" is. Note that in this method, the proof of self-stabilization of the system is given by the correctness of the algorithms that implement the method, i.e., the proof is implicit. The processes that execute P have to be modified to send snapshot messages to, and receive reset messages from the global monitor. Our method does not use a global monitor and the modifications introduced into the specification are used to guarantee self-stabilization.

8 Conclusions

In this paper, we have proposed a mechanism for tackling an important class of faults, namely coordination loss, that are very difficult to catch in the testing process. We presented a set of design principles for designing self-stabilizing protocols, and applied these principles to a real protocol.

Note that the design principles proposed in this paper and design for testability in the hardware domain share some of the same fundamentals but with different goals. DFT in the hardware domain considers one or more fault models when designing an integrated circuit. The idea is to generate test cases according to these fault models, and then check whether the behavior of the manufactured IC follows the behavior assumed by the fault models [3]. Similarly, we consider the coordination loss fault model in the design process in order to avoid errors related to this fault model.

We showed that conformance relations that are environment independent are too generic to deal with errors caused by the environment such as coordination loss. We presented a more realistic conformance relation based on external behavior and a "more testable" relation that reflects the reliability of protocol implementations. An interesting direction that we could follow from here is to define other conformance relations based on other fault models, and incorporate them into the design as well.

In designing protocols that are self-stabilizing, we are shifting the task of catching errors due to coordination loss from the testing phase to the design phase where we have a better way of handling this problem. Furthermore, the design principles we have proposed comprise of analysis and synthesis techniques.

The proposed method of converting a protocol to be self-stabilizing does not present any problem in terms of time or space. The self-stabilizing version should have the same complexity as the original protocol in processing each event. The complexity (or overhead) for a protocol to converge to a safe state depends on the protocol itself. For the protocol studied in this paper, Theorem 10 shows that the complexity is $\Theta(m)$, where m is the number of messages in the system.

References

[1] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In *PSTV, VI*, pages 349–360, 1986.

[2] J.E. Burns, M.G. Gouda, and R.E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[3] S.T. Chanson, A.A.F. Loureiro, and S.T. Vuong. Testing and testability in hardware and software. In *Proc. of the 9th Int'l Conf. on Systems Eng.*, pages 240–244, Las Vegas, USA, 14–16 July 1993.

[4] D. Clark, M. Lambert, and L. Zhang. NETBLT: A high throughput transport protocol. Network Information Center RFC 998, SRI International, March 1987.

[5] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[6] K. Drira, P. Azema, B. Soulas, and A.M. Chemali. A formal assessment of synchronous testability for communicating systems. In *Proc. of the 13th Int'l Conf. on Dist'd Comput. Syst.*, pages 149–156, Pittsburgh, PA, USA, May 1993.

[7] K. Drira, P. Azema, B. Soulas, and A.M. Chemali. Testability of a system through an environment. In *LNCS* 668, 1993.

[8] R. Gotzhein. On conformance in the context of open systems. In *Proc. of the 12th Int'l Conf. on Dist'd Comput. Syst.*, pages 236–243, Yokohama, Japan, 9–12 June 1992.

[9] M.G. Gouda and N.J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, C-40(4):448–458, April 1991.

[10] B.T. Hailpern. *Verifying concurrent processes using temporal logic*, volume 192 of *LNCS*. Springer-Verlag, 1982.

[11] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[12] G.J. Holzmann. Algorithms for automated protocol validation. *AT&T Technical Journal*, 69(1):32–44, January/February 1990. Special issue on Protocol Testing and Verification.

[13] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[14] ISO IS9646: Information Processing Systems, Open Systems Interconnection. OSI Conformance Testing Methodology and Framework, 1994. Version 10.6 of 14 March 1994.

[15] S. Katz and K.J. Perry. Self-stabilizing extensions for message-passing systems. In *Proc. of the 9th Annual ACM Symp. on Principles of Dist'd Comput.*, pages 91–101, Quebec City, Canada, 22–24 August 1990.

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

[17] L. Lamport. The mutual exclusion problem: Part II – Statements and solutions. *Journal of the ACM*, 33(2):327–348, 1984.

[18] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[19] K. Sabnani and A. Netravali. A high speed transport protocol for datagram/virtual circuit networks. In *Proc. of the SIGCOMM Symp. on Comm. Architectures, Protocols and Applications*, pages 146–157, Austin, USA, 19–22 September 1989.

[20] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[21] W.T. Strayer, B.J. Dempsey, and A.C. Weaver. *XTP: The Xpress Transfer Protocol.* Addison-Wesley, 1992.

[22] R.W. Watson. Delta-t protocols specification. Lawrence Livermore Laboratory, Calif. USA, 15 April 1983.