Proceedings of the 2nd International Workshop on

Logical Aspects of Fault-Tolerance (LAFT)

In conjunction with LICS'11

20 June 2011, Toronto, Canada

Borzoo Bonakdarpour and Tom Maibaum (Eds.)

This book constitutes the refereed proceedings of the 2nd International Workshop on Formal Aspects of Fault-Tolerance, LAFT 2011, held in Toronto, Canada, in June 2011.
The 3 revised full papers presented together with 3 invited talks were carefully reviewed and selected from submitted papers. The papers are organized in topical sections on synthesis and transitions systems, fault-tolerant distributed systems, and verification.
The international program committee included:

Efficient Computation of Most Permissive Observers in Dynamic Sensor Activation Problems

Eric Dallal[1] and Stéphane Lafortune[2]

[1] University of Michigan
Department of EECS
edallal@umich.edu
[2] University of Michigan
Department of EECS
stephane@eecs.umich.edu

Abstract. We consider the problem of dynamic fault diagnosis for discrete event systems modeled by finite state automata under the constraint that any fault must be diagnosed within no more than $K + 1$ events after its occurrence, a property called K-diagnosability. We begin by defining an appropriate notion of information state for the problem and defining dynamic versions of the projection operator and information state evolution. We continue by showing that the problem can be reduced to that of state disambiguation, then we define the most permissive observer structure that contains all the solutions to the problem, and we recall previous results showing that maintining the K-diagnosability property is equivalent to satisfying the extended specification of the state disambiguation problem. We then prove a monotonicity property of the extended specification, and show that this allows us to reduce our information state, which in turn allows us to significantly reduce the complexity of our solution. Finally, we show how to efficiently compute the extended specification by reducing the problem to maximal weight paths on a particular graph and provide an algorithm for constructing the most permissive observer.

Keywords: dynamic fault diagnosis, discrete event systems, sensor selection

1 Introduction

The problem under consideration in this work is that of dynamic fault diagnosis for discrete event systems modeled by finite state automata. We assume that there are sensors capable of detecting event occurrences for a subset of the events of the automaton model of the system. Among those events that are monitorable, it is further assumed that there is a subset whose sensors are costly to operate. This may be because of limited availability of energy or bandwidth, out of a desire to minimize communication for security reasons, or for any other reason.

We use the sensor outputs and system model to diagnose past fault occurence. In this work, we consider the K-diagnosability property, which stipulates that faults must be diagnosed within no more than $K+1$ events after their occurrence. It is assumed that we do not have sensors capable of detecting these fault events. Thus, the structure of the problem presents a trade-off: if sensors are turned on too infrequently, we may fail to diagnose the fault in time; if sensors are turned on too frequently, fault diagnosis will be needlessly costly. The problem is dynamic because we assume that sensors can be turned on or off at different points in the system's execution. A controller in this problem is (in the most general sense) a function mapping histories of past control decisions and observed events to a set of sensor activations.

Other works to have considered the problem of sensor selection under some diagnosability constraint include [1, 6, 7]. In [6], the emphasis is on finding a single dynamic controller that solves the diagnosability problem optimally according to a discounted numerical cost criterion. They use an information state based approach in combination with dynamic programming. In [1], the solution concept of most permissive observer (MPO) is introduced, which they use as a basis for optimization according to a non-discounted numerical cost criterion. Instead of an information state based approach, they use results from game theory (particularly saftey games) and games on graphs. Finally, [7] considers the decentralized version of the dynamic diagnosis problem. They use a "window-based partition" approach to obtain polynomial time algorithms for computing solutions in the centralized and decentralized cases.

In previous work [2], we redefined the MPO structure originally defined in [1] by developping it through an information state based approach similar to that used in [6]. We proved in that paper that our information state was sufficient to fully determine the set of all control decisions (i.e., sensor activations) that maintained the K-diagnosbility property. Our goals in redefining the MPO were to obtain a more readily interpretable solution, to better study informational properties of the MPO, and to have a method that could more easily be adapted to other dynamic optimization problems in discrete event systems. In a recent technical report [3], we showed that the fault diagnosis problem under consideration could be reduced to a state disambiguation problem. We used this to prove a number of monotonocity properties about the MPO and, most notably, we showed that satisfying the K-diagnosability property was equivalent to satisfying the extended specification of the state disambiguation problem. This will be discussed in Sect. 4. In this paper, we prove a monotonicity property of the extended specification, and show how this allows us to reduce our information state, without losing any "useful" information for the purpose of diagnosis (see Sect. 5). Significantly, this results in a substantial reduction in the complexity in size of our solution. Our focus then moves to algorithmic aspects of the MPO in Sect. 6. In Subsection 6.1, we show how to efficiently compute the extended specification. In conjuction with a linear time algorithm for computing the reduced unobservable reach (see Def. 7 in Sect. 3 and Def. 20 in Sect. 5) presented in Appendix 1, these new results allow for a very efficient algorithm for comput-

ing satisfactory control decisions that is used in an algorithm for computing the MPO, presented in Subsection 6.2.

The remainder of this paper is organized as follows. In Sect. 2, we recall definitions related to our notion of information state. In Sect. 3, we formally define the K-diagnosability and state disambiguation problems, and show how the former can be mapped to the latter. In Sect. 4, we define the extended specification and very briefly summarize the main results of [3]. Sections 5, 6 and Appendix 1 were discussed in the preceding paragraph. Finally, we conclude in Sect. 7. Appendix 2 contains the proofs of our new results.

2 Towards an Information State

We begin by defining the dynamic diagnosability problem more precisely. Assume that the system to be diagnosed is modeled by a *deterministic* finite state automaton. We use the standard deterministic model that has been adopted in the literature on supervisory control [4] and diagnosis [5] in discrete event systems. Specifically, $G = (X, E, f, x_0)$, where X is the set of states, E is the set of events, $f : X \times E \to X$ is a partial transition function, and x_0 is the initial state. We denote by $\mathcal{L}(G)$ the language of automaton G, which is the set of all strings of events that can occur through f. The set of events E is partitioned into four categories: $E = E_o \cup E_s \cup E_{uo} \cup E_f$, where E_o is the set of freely monitorable events (the set of events for which we have zero cost sensors), E_s is the set of costly monitorable events (the set of events for which we have costly sensors), E_{uo} is the set of non-faulty unobservable events (the set of non-faulty events for which we do not have sensors), and E_f is the set of fault events whose occurrence we would like to diagnose. In this work, we assume that E_f is a singleton.

Our goal is to dynamically diagnose the occurrence of the fault event $e_f \in E_f$ within no more than $K + 1$ events after the occurrence of the fault. Note that this is $K + 1$ events of any kind, whether we can observe the events or not. This is referred to as the *K-diagnosability property*. A controller for this problem is a function that chooses a set of events to monitor for each possible past history of control decisions and observed events (see Def. 4). We wish to find all controllers that allow us to maintain the K-diagnosability property throughout system execution.

This section provides a number of definitions related to defining the information state and its evolution. Many of these definitions were contained in our two previous papers [2], and [3].

Definition 1 (Augmented State). The augmented state is a pair $(x, n) \in X \times \{-1, 0, 1, \ldots\}$, where, n represents a "count" of the number of events (of all kinds) that have occurred since a fault event occurred, or -1 if no fault event has occurred. The set of such states is denoted by $X^+ = X \times \{-1, 0, 1, \ldots\}$. The initial augmented state is $x_0^+ = (x_0, -1)$. For any augmented state $x^+ \in X^+$, let the state and count components be denoted by $S(x^+)$ and $N(x^+)$, respectively, so that $x^+ = (S(x^+), N(x^+))$. □

Definition 2 (Augmented Transition Function). We next define the augmented transition function $g : X^+ \times E \to X^+$ on augmented states that is *induced* by the automaton $G = (X, E, f, x_0)$ and the partition of event set E. Formally, for any $u = (x_u, n_u)$ and event e, we have:

- **Case 1:** If $n_u = -1$ and $e \notin E_f$ then $g(u, e) = (f(x_u, e), -1)$.
- **Case 2:** If $n_u = -1$ and $e \in E_f$ then $g(u, e) = (f(x_u, e), 0)$.
- **Case 3:** If $n_u \geq 0$ then $g(u, e) = (f(x_u, e), n_u + 1)$.

For any $U \subseteq X^+$, let $g(U, e) = \bigcup_{u \in U} g(u, e)$. This definition is extended to strings (rather than merely events) in the usual way. Finally, we define $g(s) = g(x_0^+, s)$ for brevity. □

Remark 1 (Multiple fault occurrences in an execution). In our past work in this domain, we defined a fourth case in the augmented transition function. Case 4 stated that, in the event of multiple fault occurrences, there would be *two* transitions, one corresponding to a count of zero and one corresponding to an increment of the count. That is, if $n_u \geq 0$ and $e \in E_f$, then $g(u, e) = \{(f(x_u, e), 0), (f(x_u, e), n_u + 1)\}$. As a consequence of this, g was defined as $g : X^+ \times E \to 2^{X^+}$. We ultimately determined that this added irrelevant information for the purposes of maintaining K-diagnosability, and that none of the theorems presented relied on this special case. However, the change in the *form* of g does require us to alter a few definitions. □

Definition 3 (Run). A run ρ of length n is defined as a sequence $C_0, e_0, \ldots, C_{n-1}, e_{n-1}$ of *control decisions* or *sensor activations* (the C_i's, which are subsets of events to monitor) and observed events (the e_i's). Since the events are observed, they must be among the monitored events. That is, $e_i \in C_i$, for all $i = 0, \ldots, n-1$. On the other hand, the strict alternation of control decisions and observed events reflects the assumption that control decisions are only changed upon the observance of an event. Denote by R_n the set of runs of length n. Finally, let $\rho(k) = C_0, e_0, \ldots, C_{k-1}, e_{k-1}$ denote the subsequence of ρ of length k. □

Definition 4 (Admissible Controller). An admissible controller is a sequence $C = (C_0, C_1, \ldots)$ of functions $C_n : R_n \to 2^E$, $n = 0, 1, \ldots$ from runs to control decisions that satisfies the following two conditions:

1. $C_n(\rho) \supseteq E_o$ for all $\rho \in R_n$ and all $n = 0, 1, \ldots$ (C_n includes the set of events that are always monitored).
2. $C_n(\rho) \cap (E_{uo} \cup E_f) = \emptyset$ for all $\rho \in R_n$ and all $n = 0, 1, \ldots$ (C_n cannot include any event that is unobservable, whether faulty or non-faulty). □

Note that C_n is used to denote both the controller and the control decision. Hereafter, the context will make it clear which is which.

Definition 5 (Information State). An information state (IS) is a subset $S \subseteq X^+$ of augmented states. We denote by $I = 2^{X^+}$ the set of information states. □

Definition 6 (Information State Based Controller). An information state based controller (or IS-controller) is a function $C : I \to 2^E$ that satisfies the two conditions of an admissible controller (i.e., $C(i) \supseteq E_o$ and $C(i) \cap (E_{uo} \cup E_f) = \emptyset$ for all $i \in I$). □

Constructing an observer for an automaton and a *fixed* set of unobservable events is a relatively simple task. To construct the observer when the set of observable events is a dynamic control decision, we must explicitly model the effect of the controller on the evolution of the information state.

Definition 7 (Total Observer). The total observer is defined as a directed bipartite graph $(Y \cup Z, A)$. Here, Y is the set of information states (i.e., $Y = I$), Z is the set of information states augmented with monitored event decisions (i.e., $Z = I \times 2^E$), and A is the set of edges in the graph. A Y state is an information state in which a control decision is taken and a Z state is a pair consisting of an information state and a set of monitored events, in which an observable event occurs, among those in the current set of monitored events. Thus, any run results in the alternation between Y and Z states. The set A contains all transitions from Y states to Z states (all admissible control decisions) and all transitions from Z states to Y states (all observable events). The initial Y state is just the initial information state, i.e. $y_0 = \{x_0^+\}$. Specifically, a transition from a Y state to a Z state represents the unobservable reach. As a transition from a Z state to a Y state occurs upon the observance of a *monitored* event, it is necessary for each Z state to "remember" the set of monitored events from the Y state that led to it. Let $I(z)$ and $C(z)$ denote z's information state and control decision components, respectively, so that $z = (I(z), C(z))$. Formally, $(y, z) \in A$, labeled with $C(y)$ if:

$$I(z) = UR(y, C(y)) = \left\{ \begin{array}{c} v \in X^+ : (\exists u \in y)(\exists t \in (E \setminus C(y))^*) \\ \text{s.t. } v = g(u,t) \end{array} \right\} \tag{1}$$

$$= \bigcup_{u \in y} \bigcup_{t \in (E \setminus C(y))^*} g(u,t) \tag{2}$$

$$C(z) = C(y) \tag{3}$$

where $g(\cdot,\cdot)$ is the augmented transition function. In words, this means that $I(z)$ is the set of augmented states reachable from some augmented state of the preceding Y state through some string of unmonitored events, and that $C(z)$ is the set of monitored events chosen in the preceding Y state. We write $h_{YZ}(y, C(y)) = z$. Formally, $(z, y) \in A$, labeled with $e \in C(z)$ if:

$$y = \{v \in X^+ : (\exists u \in I(z)) \quad [v = g(u,e)]\} \tag{4}$$

$$= \bigcup_{u \in I(z)} g(u,e) \tag{5}$$

In words, this means that y is the set of augmented states reachable from some augmented state of the information state component of the preceding Z state

through the *single* event e. As before, we write $h_{ZY}(z, e) = y$. The initial state of the system is the Y state corresponding to the initial information state, i.e., $y_0 = \{x_0^+\}$. □

Definition 8 (Y and Z State Controller Induced Information State Evolution). Given a controller C, we define $IS_C^Y(y, s)$ to be the Y state that results from the occurrence of string s, when starting in Y state y. This can be computed as follows:

$$\begin{aligned} IS_C^Y(y, \varepsilon) &:= y \\ IS_C^Y(y, e) &:= \begin{cases} h_{ZY}(h_{YZ}(y, C(y)), e) & \text{if } e \in C(y) \\ y & \text{if } e \notin C(y) \end{cases} \\ IS_C^Y(y, es) &:= IS_C^Y(IS_C^Y(y, e), s) \end{aligned} \tag{6}$$

For brevity, we define $IS_C^Y(s) := IS_C^Y(y_0, s)$. We define $IS_C^Z(z, s)$ analogously:

$$\begin{aligned} IS_C^Z(z, \varepsilon) &:= z \\ IS_C^Z(z, e) &:= \begin{cases} h_{YZ}(y', C(y')) \\ \text{where } y' = h_{ZY}(z, e) & \text{if } e \in C(z) \\ z & \text{if } e \notin C(z) \end{cases} \\ IS_C^Z(y, es) &:= IS_C^Z(IS_C^Z(z, e), s) \end{aligned} \tag{7}$$

As before, we define $IS_C^Z(s) := IS_C^Z(z_0, s)$, where $z_0 = C(y_0)$ (which is well defined for a fixed controller). □

For a fixed set of monitored events, it is a trivial task to define the projection of a string. When the set of monitored events changes dynamically along the string's execution, in a way that depends on the particular controller C, it is necessary to define a *controller induced* projection.

Definition 9 (Controller Induced Projection). Given a controller C, we define $P_C(z, s)$ as the string t that is observed upon the occurrence of the string s, when starting in Z state z. This can be computed as follows:

$$\begin{aligned} P_C(z, \varepsilon) &:= \varepsilon \\ P_C(z, e) &:= \begin{cases} e \text{ if } e \in C(z) \\ \varepsilon \text{ if } e \notin C(z) \end{cases} \\ P_C(z, es) &:= \begin{cases} e.[P_C(z', s)] \text{ if } e \in C(z) \\ P_C(z, s) \qquad \text{if } e \notin C(z) \end{cases} \\ &\text{where } z' = h_{YZ}(y', C(y')) \text{ and } y' = h_{ZY}(z, e) \end{aligned} \tag{8}$$

For the last case, the first argument of P_C must be updated with the new Z state. If the event e is not monitored, then the system remains within the same unobservable reach, and hence z remains the same. If e is observed, then we must first determine the Y state to which the system transitions to, and then determine the next Z state given the Y state and the controller C. For brevity, we define $P_C(s) := P_C(z_0, s)$. □

Remark 2 (IS-Controllers vs. General Controllers). Obviously, the total observer is defined based on the information state, since the monitoring decision taken at any point in time is a function of the information state. It must be noted that, however, the functions h_{YZ} and h_{ZY} are well defined without assuming that that the controller is information state based. Indeed, we may simply define them in terms of the run $\rho = C_0, e_0, \ldots$ as follows:

$$I(z_k) = \{v \in X^+ : (\exists u \in y_k)(\exists t \in (E \setminus C_k)^*) \text{ s.t. } v \in g(u,t)\} \tag{9}$$

$$C(z_k) = C_k \tag{10}$$

$$y_{k+1} = \{v \in X^+ : \exists u \in I(z_k) \text{ s.t. } v \in g(u, e_k)\} \tag{11}$$

For any given run $\rho = C_0, e_0, \ldots$, this leads to a sequence $y_0, z_0, y_1, z_1, \ldots$. Thus, the controller induced information state evolution and controller induced projection are also well defined without assuming that the controller is information state based. In fact, these definitions need only be modified by changing y to y_k, z to z_k, and both $C(y)$ and $C(z)$ to C_k. This also shows that both the Y and Z states satisfy the two properties of proper information states. First, they are both uniquely determined by the past sequence of observations (the e_i's) and control decisions (the C_i's). Second, the next Y or Z state is a function of the current Y or Z state, the control decision, and the observation, which is accomplished through the h_{YZ} and h_{ZY} functions. □

Remark 3 (Feasible run). Remark 2 allows us to define the concept of a *feasible* run by stating that a run is feasible if and only if $\exists x \in S(I(z_k)) : f(x, e_k)$ is defined, for $k = 0, 1, \ldots$. In words, this means that a run is feasible if event e_k can occur from some state in the unobservable reach z_k (i.e., event e_k is feasible), for all k. This definition is in a remark rather than a definition of its own because none of the other definitions require a run to be feasible: a non-feasible event leads to an empty information state, from which everything remains well defined (we will simply have an empty information state from that point on). □

Lemma 1 (Relation between projection and information state). *For any string s and controller C, $I(IS_C^Z(s)) = \{v \in X^+ : \exists s' \text{ s.t. } P_C(s) = P_C(s') \wedge v = g(s')\}$.*

This lemma shows the relationship between the string-based information state defined in the first sections of [6] and our information state (which is the analog of the information state they use when considering general untimed cyclic automata, adapted to the problem of K-diagnosability). The first kind of information state, which consists of the sets of undifferentiable strings, is in some sense the most general form of information state for a partially observed DES. Our information state is more compact but less general in two ways. First, by being sets of augmented states rather than strings, there exists the possibility that, for some string s, controller C and Z state z, $u \in g(s)$ and $u \in I(z)$, but $IS_C^Z(s) \neq z$ (the same is true for Y states as well). This is because a particular augmented state may be reached through many different strings (i.e., many paths in the automaton G^+), and these strings need not all have the same projection

or lead to the same information state. Second, a particular Y or Z state may be reached through different strings that do not even share a common projection. This occurs since there may be multiple paths to a particular Y or Z state in the total observer, even if we restrict the total observer to the particular control decisions made by some controller C. As we will see later on, this refinement of the information state makes no difference when determining the set of control decisions that maintain the K-diagnosability condition.

3 *K*-Diagnosability, State Disambiguation, and the Most Permissive Observer

This section consists of three parts. In the first part, we formally define the property of K-Diagnosability. In the second part, we briefly describe the state disambiguation problem and show that the K diagnosability property can be formulated as a state disambiguation problem. Finally, we define the Most Permissive Observer (MPO) in the last part.

Definition 10 (K-Diagnosability). We recall the standard definition of diagnosability from [5]. Adapted for a fixed K and the context of a dynamic observer, we say that a system G is K-diagnosable given controller C if there do not exist a pair of strings $s_Y, s_N \in \mathcal{L}(G)$ such that:

1. s_Y has an occurrence of a fault event $f \in E_f$ and s_N does not.
2. s_Y has at least $K+1$ events after the fault event f
3. $P_C(s_Y) = P_C(s_N)$, that is, the observed string of events is identical given the controller.

We also say that a system G is K-diagnosable if there exists a controller C such that G is K-diagnosable given controller C. We call such a controller *safe*. □

Definition 11 (State Disambiguation Problem). The state disambiguation problem is defined as a triple $\langle G^{sd}, \Sigma_o, T_{\text{spec}} \rangle$, where $G^{sd} = (X^{sd}, \Sigma, f^{sd}, x_0^{sd})$ is an automaton, $\Sigma_o \subseteq \Sigma$ is a set of monitorable events, and $T_{\text{spec}} \subseteq X^{sd} \times X^{sd}$ is a set of pairs that must not be confused. The state disambiguation problem consists of finding a controller C for G^{sd}, which chooses sensors to activate, such that the state of G^{sd} is never confused between any pair of states in the specification T_{spec}. The controller C is defined as a sequence of functions from runs to control decisions, as in the previous section. That is, $C = (C_0, C_1, \ldots)$, where $C_n : R_n \to 2^{\Sigma_o}$, for $n = 0, 1, \ldots$. Using the notation defined in the previous section of this work, we can define the problem formally as that of finding a controller C such that:

$$s_1, s_2 \in \mathcal{L}(G) : P_C(s_1) = P_C(s_2) \Rightarrow (f(x_0, s_1), f(x_0, s_2)) \notin T_{\text{spec}} \ . \tag{12}$$

□

To formulate the K-diagnosability problem as a state disambiguation problem, we specify each of G^{sd}, Σ_o, and T_{spec}:

- $G^{sd} = G_K^+$ which is the automaton $G^+ = (X^+, E, g, x_0^+)$, but restricted to augmented states having counts no greater than $K+1$, where G^+ is the automaton defined over augmented states (i.e., using the augmented transition function g of Def. 2).
- $\Sigma_o = E_o \cup E_s$ is the set of monitorable events (we assume that C is an admissible controller).
- Finally, we define T_{spec} as:

$$T_{\text{spec}} = \{(u, v) \in X^+ \times X^+ : N(u) = -1 \wedge N(v) = K+1\} \ . \tag{13}$$

Comparing definitions 10 and 11, we see that this state disambiguation problem can be satisfied if and only if there do not exist two strings s_1 and s_2 with $P_C(s_1) = P_C(s_2)$, $N(g(x_0^+, s_1)) = -1$, and $N(g(x_0^+, s_2)) = K+1$. Recall that $N(g(x_0^+, s))$ is equal to -1 if there is no fault in s, and the number of events since a fault event otherwise. If we take s_1 and s_2 in this problem to correspond to s_N and s_Y of definition 10, we see that the two problems are identical, except for the fact that, in the definition of K-diagnosability s_Y must have *at least* $K+1$ events after a fault, whereas in T_{spec} the string s_2 has *exactly* $K+1$ events after a fault. To see that this makes no difference, suppose that there exist two strings s_N and s_Y that violate K-diagnosability and that s_Y has r events after a fault. Then we may simply truncate s_Y to obtain an s'_Y with exactly $K+1$ events after a fault. If this shortens the projection $P_C(s_Y)$, we can shorten s_N as well so that $P_C(s'_Y) = P_C(s'_N)$.

Definition 12 (K-diagnosable binary function for information states). An information state $i \in I$ violates K-diagnosability if there exist two augmented states $x_1^+, x_2^+ \in i$ where $x_1^+ = (x_1, -1)$ and $x_2^+ = (x_2, n)$ for some $n > K$. In light of the definition of T_{spec}, we define the K-diagnosibility binary function for information states $D_I : I \to \{0, 1\}$ as:

$$D_I(i) = \begin{cases} 0, \exists u, v, \in I : (u, v) \in T_{\text{spec}} \\ 1, \text{else} \end{cases} \tag{14}$$

In words, $D_I(i) = 1$ if and only if i does not violate the K-diagnosability property. □

Theorem 1 (Formulation of the K-diagnosability property through the information state). *Controller C maintains K-diagnosability if and only if $D_I(I(z)) = 1$, for all reachable Z states. Mathematically:*

$$\begin{aligned} &\exists s \in \mathcal{L}(G) : z = IS_C^Z(s) \wedge \exists u, v \in I(z) \ \textit{s.t.}\ N(u) = -1 \ \textit{and}\ N(v) = K+1 \\ \Leftrightarrow\ &\exists s_Y, s_N \in \mathcal{L}(G) : P_C(s_Y) = P_C(s_N), N(g(s_N)) = -1 \ \textit{and}\ N(g(s_Y)) = K+1. \end{aligned}$$

The above theorem has two consequences. The first is that we are justified in using an information state based approach with our information state to find safe controllers for the K-diagnosability problem. The second consequence

is that we have a test for determining whether or not a particular controller is safe, in terms of the Z states that are reachable. Specifically, they must all satisfy $D_I(I(z)) = 1$. Mathematically, we can write that controller C is safe if $D_I(I(IS_C^Z(s))) = 1$, for all $s \in \mathcal{L}(G)$. A minor additional consequence is that we can consider only Z states. This could have been guessed from the definition of the Z state as the unobservable reach of the preceding Y state, from which it follows that, for any y and any $C(y)$, $y \subseteq I(z)$ for $z = h_{YZ}(y, C(y))$.

Definition 13 (K-diagnosibility binary functions for Y and Z states). The purpose of defining the Most Permissive Observer is to capture all controllers that satisfy the K-diagnosability property. In light of the preceding theorem and the fact that we can deterministically compute future Z states from the current Y or Z state and the the sequence of future control decisions and observed events, we see that we can speak not only of safe controllers but also of safe information states. Specifically, we say that Y-State y is safe if it currently satisfies the K diagnosability property and there exists some controller that maintains the K-diagnosability property for all future executions of the system. Since we can choose control decisions but not event occurrences, we therefore define two K-diagnosability binary functions, $D_Y : Y \to \{0, 1\}$ and $D_Z : Z \to \{0, 1\}$ (similar to D_I, but for Y and Z states) as follows:

$$D_Y(y) = \begin{cases} 1 \text{ if } D_I(y) = 1 \text{ and } \exists C(y) : D_Z(h_{YZ}(y, C(y))) = 1 \\ 0 \text{ else} \end{cases} \tag{15}$$

$$D_Z(z) = \begin{cases} 1 \text{ if } D_I(I(z)) = 1 \text{ and } D_Y(h_{ZY}(z, e)) = 1 \quad \forall e \in C(Z) \\ 0 \text{ else} \end{cases} \tag{16}$$

□

From these definitions, we can say that G is K-diagnosable if and only $D_I(y_0) = 1$, and $\exists C(y_0) : D_I(h_{YZ}(y_0, C(y_0))) = 1$, and $\exists C(y_0) \forall e_0 \in C(y_0) : D_I(h_{ZY}(h_{YZ}(y_0, C(y_0)), e_0)) = 1$, etc... Put in terms of the information state evolution, we can equivalently say that G is K-diagnosable if and only if $\exists C$ such that $\forall s \in \mathcal{L}(G)$, $D_I(IS_C^Y(s)) = 1$ and $D_I(I(IS_C^Z(s))) = 1$ (in practice, the second condition is sufficient since $y \subseteq I(z)$ whenever $z = h_{YZ}(y, C(y))$). Thus the alternation of existential and universal quantifiers implicitly captures the idea that there must exist some controller such that some condition (namely K-diagnosability in this case) must hold for all possible strings of events. This is the same conclusion that is reached from Thm. 1.

Definition 14 (Safe Control Decision). The above development allows us to define safe control decisions in terms of Y and Z states. Specifically, we can say that control decision $C(y)$ is safe in information state Y if and only if $D_Z(h_{YZ}(y, C(y))) = 1$, since we know that there exists a future sequence of safe control decisions in this case. □

The above definition has the consequence that after any run ρ resulting in Y state y, the set of safe control decisions is uniquely determined by the information

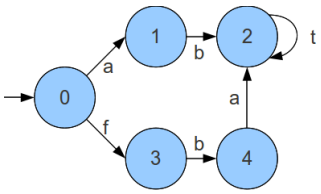

Fig. 1. A finite state automaton. Events are classified as follows: $E_o = \emptyset$, $E_s = \{a, b\}$, $E_{uo} = \{t\}$, and $E_f = \{f\}$.

state y. Thus there is no loss of generality in using information state based controllers. A further consequence of this is that a safe run-based controller is any controller that always makes control decisions within these safe sets. Finally, we can also conclude that the existence of a safe run-based controller implies the existence of a safe information state based controller.

Definition 15 (Fault diagnosis binary function). Define the fault diagnosis binary function $D_F : Y \to \{0, 1\}$ as follows:

$$D_F(y) = \begin{cases} 1 \text{ if } N(u) \neq 1 \quad \forall u \in y \\ 0 \text{ else} \end{cases} \tag{17}$$

In words, this means that $D_F(y) = 1$ if and only if all possible executions $s \in \mathcal{L}(G)$ resulting in information state y had a fault occurrence. A Y state y satisfying $D_F(y) = 1$ is called *diagnosed*. □

Definition 16 (Most Permissive Observer). The most permissive observer is defined as the K-diagnosable connected subgraph of the total observer that includes state y_0, together with an additional state F (called the "fault detected" state). By the K-diagnosable subgraph of the total observer, we mean the subgraph of the total observer consisting only of K-diagnosable Y and Z states, and the transitions between them. The recursive structure of D_Y therefore captures all "safe" control decisions $C(y)$, for all $y \in Y$ (and only safe control decisions), where by safe we mean that these decisions do not violate K-diagnosability. The single state F is used to denote any state where a fault has been diagnosed. That is, a state y satisfying $D_F(y) = 1$. All such Y states are replaced by the single state F, which is a terminal node in the sense that there are no transitions out of it (i.e., we make no further control decisions from F). □

An example of the MPO is useful at this point:

Example 1 (A simple example).

Consider the automaton shown in Figure 1. Initially, it is necessary to choose to monitor event b, for otherwise it will not be possible to differentiate between the string $fbat^n$ and the string a, which means K-diagnosability is violated for any K. If, after choosing to monitor just $\{b\}$, the event b is observed, then it

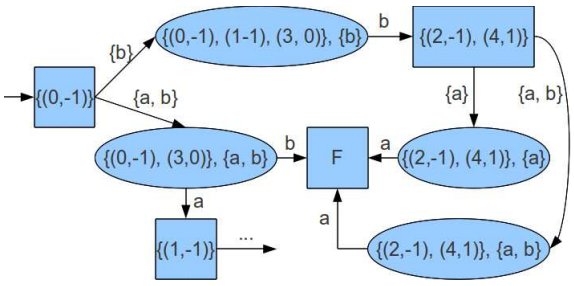

Fig. 2. The MPO corresponding to the automaton of Fig. 1, for $K \geq 1$.

is now necessary to monitor event a, for otherwise it will not be possible to differentiate between the string ab and the string $fbat^n$, which again violates K-diagnosability for any K. The remaining decisions are made only in parts of the graph where it is certain that no fault has occurred, and hence any decision of events to monitor is valid (this part of the graph has been omitted). The resulting MPO is shown in Figure 2. Notice that we can achieve 0-diagnosability by choosing to monitor $\{a, b\}$ initially, and only 1-diagnosability if we choose to monitor only $\{b\}$. We used the convention of [1] by marking Y states with squares and Z states with circles. □

4 The Extended Specification and Properties of the MPO

In this section, we present the extended specification and explain how it allows for a simple test to determine the safety of the Y and Z states of the MPO.

Definition 17 (Natural Projection). The natural projection $P : E^* \to (E_o \cup E_s)^*$ is defined by:

$$\begin{aligned} P(\varepsilon) &:= \varepsilon \\ P(e) &:= \begin{cases} e \text{ if } e \in E_o \cup E_s \\ \varepsilon \text{ if } e \notin E_o \cup E_s \end{cases} \\ P(es) &:= P(e)P(s) \end{aligned} \tag{18}$$

In words, the natural projection "filters out" all unobservable events. It is equivalent to the controller induced projection with the dynamic controller C_{all} that always chooses to monitor $E_o \cup E_s$ (i.e., all monitorable events). □

Definition 18 (Extended Specification). We repeat the definition of the extended specification found in [8]:

$$T^e_{\text{spec}} = \begin{array}{c} \{(u, v) \in X^+ \times X^+ : \exists s_1, s_2 \text{ s.t. } P(s_1) = P(s_2) \\ \text{and } (g(u, s_1), g(v, s_2)) \in T_{\text{spec}}\} \end{array} \tag{19}$$

In words, the extended specification is defined as the set of all augmented state pairs that cannot be confused because even if all the sensors in $E_o \cup E_s$ are

turned on for the rest of time, there still exists some sequence of events such that some pair in T_{spec} will be confused. □

Definition 19 (Extended specification binary function for information states). In light of the definition of T^e_{spec}, we define the extended specification binary function for information states $D^e_I : I \to \{0, 1\}$ as follows:

$$D^e_I(i) = \begin{cases} 0, \exists u, v, \in I : (u, v) \in T^e_{\text{spec}} \\ 1, \text{ else} \end{cases} \tag{20}$$

In words, $D^e_I(i) = 1$ if and only if i does not violate the extended specification. □

In [3], we proved a number of monotonicity properties about the MPO. These can be paraphrased as: more information can never harm K-diagnosability. We also proved two results demonstrating the usefulness of the extended specification, which are merged into one theorem and presented here (for proofs, see [3]).

Restated Theorem 1 (Safety is equivalent to satisfying the extended specification for Y and Z states). *For any Y state y, $D_Y(y) = D^e_I(y)$ and for any Z state z, $D_Z(z) = D^e_I(I(z))$.*

Thus, control decision $C(y)$ is safe in Y state y if and only if $D^e_I(I(h_{YZ}(y, C(y)))) = D^e_I(UR(y, C(y))) = 1$. Determining whether or not a control decision is safe can therefore be accomplished through two operations: computing the unobservable reach and checking if the extended specification is satisfied.

5 Reducing the Information State

In this section, we prove new results using a monotonicity property on the extended specification. Specifically, we use this to show that we can "reduce" our information state. That is, we will show that, as currently defined, our information state carries more information than what is strictly necessary for the problem of K-diagnosability.

Theorem 2 (Monotonicity Property for the Extended Specification). *Suppose that $(u, v) \in T^e_{spec}$. Then, for any v' such that $S(v') = S(v)$ and $N(v') > N(v)$, we also have $(u, v') \in T^e_{spec}$.*

The above theorem allows us to reduce the extended specification to a $|X| \times |X|$ table filled with elements of $\{-1, 0, \ldots, K + 1\}$, where an entry of n at location (x_1, x_2) signifies that $((x_1, -1), (x_2, n)) \in T^e_{\text{spec}}$ and that, for all $n' < n$, $((x_1, -1), (x_2, n')) \notin T^e_{\text{spec}}$. We will see with the next definition and theorems that this has even more significant consequences.

Definition 20 (Information State Reducing Function). Define $R : I \to I$ by:

$$R(i) = \{u \in i : [N(u) = -1] \vee [\nexists v \in i \text{ s.t. } S(v) = S(u) \text{ and } N(v) > N(u)]\} \ . \tag{21}$$

Also define $R : Z \to Z$ by $R(z) = (R(I(z)), C(z))$. Finally, define reduced versions of h_{YZ}, UR, h_{ZY}:

$$h^R_{YZ}(y, C(y)) = R(h_{YZ}(y, C(y))) \tag{22}$$

$$RUR(y, C(y)) = R(UR(y, C(y))) \tag{23}$$

$$h^R_{ZY}(z, e) = R(h_{ZY}(z, e)) \tag{24}$$

□

In words, $R(\cdot)$ *reduces* an information state by keeping only those augmented states within it that have a count of -1 and, for each state in X, keeping only the augmented state with that state component that has the highest count. As the following corollary will show, this information is sufficient to determine the safety of a Y or Z state.

Corollary 1 (Reduced information state carries all necessary information in determining safety). *For any information state $i \in I$, $D^e_I(i) = D^e_i(R(i))$.*

This proves that $R(\cdot)$ conserves all the necessary information for determining the safety of a Y or Z state. However, it is still possible that this "filtering out" of information in the present could change safety properties of Y or Z states in the future (i.e., those Y and Z states that are reachable further in the execution of the system). The following two theorems preclude this possibility.

Theorem 3 (There is no loss in applying h^R_{ZY} to a reduced Z state). *For any Z state z and event $e \in C(z)$, $h^R_{ZY}(z, e) = h^R_{ZY}(R(z), e)$.*

Theorem 4 (There is no loss in applying h^R_{YZ} to a reduced Y state). *For any Y state y and control decision $C(y)$, $h^R_{YZ}(y, C(y)) = h^R_{YZ}(R(y), C(y))$.*

Note that the proofs of these two theorems use a lemma that is presented only in Appendix 2. By induction, Thms. 3 and 4 show that we do not lose any information relevant to determining safety of Y or Z states by working solely with reduced information states. Significantly, this substantially reduces the number of "distinct" information states. Without this reduction, the number of information states is $2^{(K+2)|X|}$, since there are $|\{-1, 0, 1, \ldots, K\}| = K + 2$ values for the count component of each augmented state in the information state. For reduced information states, we must indicate, for each $x \in X$, whether or not $(x, -1)$ is present and what the maximal value of n is such that (x, n) is present. Thus, there are only $2^{|X|} \cdot (K + 3)^{|X|}$ distinct reduced information states (there are $K + 3$ rather than $K + 2$ possibilities for the maximal count since it may be that no (x, n) is present for any n).

6 Algorithmic Aspects of the MPO

This section consists of two subsections. In the first, we describe how to use the reduced information state to efficiently compute the extended specification.

In the second, we provide an algorithm for computing the MPO, as well as an example that demonstrates how the reduction of the information state allows us to reduce the size of the MPO.

6.1 Computing the Extended Specification

In this section, we show that computing the extended specification is equivalent to finding maximal weight paths on a particular graph. This idea was presented in [1], in which the authors reduced the problem of finding the minimal K for which K-diagnosability can be achieved for a given automaton (i.e., the minimal detection delay). Computing the extended specification is a similar, but more general problem. In fact, in the notation to follow, the problem of computing this minimal K is equivalent to determining the single value $m^F(x_0, x_0)$. The method used here is effectively the same as that presented in [9], except that they use the more general notion of masks and also only use the algorithm to determine the minimal detection delay.

In what follows, let $L = \mathcal{L}(G)$ be the language of the automaton $G = (X, E, f, x_0)$ and let $L^{NF} = L \cap (E \setminus E_f)^*$, the language of G but excluding strings with fault occurrences. Also, for any state $x \in X$, let L/x denote the language $\mathcal{L}(G')$, where $G' = (X, E, f, x)$. That is, L/x denotes the language that is possible given the automaton G when starting in state x. Define L^{NF}/x analogously. Finally, for any string s and fault event e_f, let s/e_f denote the part of s beginning at the *first* occurence of event e_f in s, or ε if e_f does not occur in s. As before, we assume that there is only a single fault event, so that $E_f = \{e_f\}$.

In previous work, we proved that safety is equivalent to satisfying the extended specification. By Thm. 2, it suffices to find (for each $(x_1, x_2) \in X^2$) the minimum count n_2 such that $((x_1, -1), (x_2, n_2)) \in T^e_{\text{spec}}$ to compute the extended specification. But by Def. 18, this is equivalent to finding:

$$\min \left\{ \begin{array}{l} n_2 \in \{-1, 0, 1, \ldots\} : \exists s_1 \in L/x_1, s_2 \in L/x_2 \text{ s.t. } P(s_1) = P(s_2), \\ N(g((x_1, -1), s_1)) = -1, \text{ and } N(g((x_2, n_2), s_2)) = K + 1 \end{array} \right\} \tag{25}$$

Instead of computing this minimum, we compute the following two maxima:

$$m(x_1, x_2) = \max_{s_1 \in L^{NF}/x_1, s_2 \in L/x_2 : P(s_1) = P(s_2)} |s_2| \tag{26}$$

$$m^F(x_1, x_2) = \max_{s_1 \in L^{NF}/x_1, s_2 \in L/x_2 : P(s_1) = P(s_2)} |s_2/e_f| \tag{27}$$

Suppose that, for some $(x_1, x_2) \in X^2$, we have $m^F(x_1, x_2) = c$. Then we know that there exist $s_1 \in L^{NF}/x_1$ and $s_2 \in L/x_2$ such that $P(s_1) = P(s_2)$ and $|s_2/e_f| = c$. First, since $s_1 \in L^{NF}/x_1$, it follows that $N((g(x_1, -1), s_1)) = -1$. Second, since $|s_2/e_f|$ is the number of events in s_2 starting at the first occurrence of e_f, it follows that $N((g(x_2, -1), s_2)) = c - 1$. Thus, the minimum in equation (25) is $n_2 = -1$ if $c - 1 \geq K + 1$. Now suppose that $m(x_1, x_2) = c$. Then we know that there exist $s_1 \in L^{NF}/x_1$ and $s_2 \in L/x_2$ such that $P(s_1) = P(s_2)$ and $|s_2| = c$. As before, this implies that $N((g(x_1, -1), s_1)) = -1$. It also implies

that $N((g(x_2, n_2), s_2)) = c + n_2$ if $n_2 \geq 0$. Thus, the solution to equation (25) is $\max\{0, K + 1 - m(x_1, x_2)\}$ if $m^F(x_1, x_2) < K + 2$ (otherwise, the solution is -1, from the discussion above). An advantage to using this method for computing the extended specification is that the values in equations (26) and (27) are not dependent on the particlar value of K. This means that we do not need to recompute the extended specification for different values of K. The procedure for computing the values of $m(\cdot, \cdot)$ and $m^F(\cdot, \cdot)$ is described below.

1. Create the graph $G^e_{\text{spec}} = (V^e_{\text{spec}}, A^e_{\text{spec}})$, where $V^e_{\text{spec}} = X \times X \times \{N, Y\}$, and $A^e_{\text{spec}} \subseteq V^e_{\text{spec}} \times V^e_{\text{spec}}$ is the set of (directed) edges between them, with labels in the set $E \times E$. Here, N and Y are fault labels, where N represents no fault (i.e., a count of -1) and Y represents the occurence of a fault at some point in the past (i.e., any count not equal to -1). We use $FL = \{N, Y\}$ to denote the set of fault labels. Define the function $FL^e_{\text{spec}} : V^e_{\text{spec}} \to FL$ to be the fault label associated with a vertex. That is:

$$FL^e_{\text{spec}}(v) = \begin{cases} N, \text{ if } v \in X \times X \times \{N\} \\ Y, \text{ if } v \in X \times X \times \{Y\} \end{cases} \tag{28}$$

   Also, let $EL^e_{\text{spec}} : A^e_{\text{spec}} \to E \times E$ be the function that assigns labels to edges. The set of edges A^e_{spec} is defined by three cases:

   Observed Events For any vertex $v_1 = (x_1, x_2, fl)$ and any event $e \in E_o \cup E_s$, if $f(x_1, e)$ and $f(x_2, e)$ are both defined, then there exists an edge $(v_1, v_2) \in A^e_{\text{spec}}$ with label (e, e), where $v_2 = (f(x_1, e), f(x_2, e), fl)$.

   Unobservable Events For any vertex $v_1 = (x_1, x_2, fl)$ and any event $e \in E_{uo}$, if $f(x_1, e)$ is defined, then there exists an edge $(v_1, v_2) \in A^e_{\text{spec}}$ with label (e, ε), where $v_2 = (f(x_1, e), x_2, fl)$. Similarly, if $f(x_2, e)$ is defined, then there exists an edge $(v_1, v_2) \in A^e_{\text{spec}}$ with label (ε, e), where $v_2 = (x_1, f(x_2, e), fl$.

   Faulty Events For any vertex $v_1 = (x_1, x_2, fl)$ and any event $e \in E_f$, if $f(x_2, e)$ is defined, then there exists an edge $(v_1, v_2) \in A^e_{\text{spec}}$ with label (ε, e), where $v_2 = (x_1, f(x_2, e), Y)$.

2. Assign weights to each edge of G^e_{spec} through the function $W^e_{\text{spec}} : A^e_{\text{spec}} \to \{0, 1\}$ as follows. For any $a = (v_1, v_2) \in A^e_{\text{spec}}$ with label $EL(a) = (e_1, e_2)$,

$$W^e_{\text{spec}}(a) = \begin{cases} 0, \text{ if } [FL^e_{\text{spec}}(v_2) = N \text{ and } e_2 \notin E_f] \text{ or } [FL^e_{\text{spec}}(v_2) = Y \text{ and } e_2 = \varepsilon] \\ 1, \text{ else} \end{cases} \tag{29}$$

   Thus, if $W^e_{\text{spec}}(a) = 1$, then edge a corresponds either to a fault event or to an event occuring after a fault event that causes a transition on the faulty state. For any $(x_1, x_2) \in X^2$, the value of $m(x_1, x_2)$ is equal to the maximal weight of a path starting at (x_1, x_2, Y). Similarly, the value of $m^F(x_1, x_2)$ is equal to the maximal weight of a path starting at (x_1, x_2, N). Let the weight of the maximal weight path starting from $v \in V^e_{\text{spec}}$ be denoted by $d^e_{\text{spec}}(v)$

3. Find the strongly connected components (SCCs) of the graph G^e_{spec}. For any vertex $v \in V^e_{\text{spec}}$, let $SCC(v)$ denote the SCC containing vertex v. Since all edge weights are non-negative, the maximal path weights starting from

any two vertices in the same strongly connected components must be equal. That is, $SCC(v_1) = SCC(v_2) \Rightarrow d^e_{\text{spec}}(v_1) = d^e_{\text{spec}}(v_2)$. This is clearly true if all edges in a SCC have weight zero. If there is a non-zero edge, it can be traversed infinitely often on a maximal weight path starting from any vertex in the SCC, in which case the maximal path weights will be infinite for any vertex in the SCC.

4. Create a new graph $G^{SCC}_{\text{spec}} = (V^{SCC}_{\text{spec}}, A^{SCC}_{\text{spec}})$, which is the graph over strongly connected components. That is, $V^{SCC}_{\text{spec}} = \{SCC(v) : v \in V^e_{\text{spec}}\}$ and $A^{SCC}_{\text{spec}} = \{(S_1, S_2) \in V^{SCC}_{\text{spec}} \times V^{SCC}_{\text{spec}} : \exists v_1 \in S_1, v_2 \in S_2 \text{ s.t. } (v_1, v_2) \in A^e_{\text{spec}}\}$. Also, assign weights to each edge of G^{SCC}_{spec} through the function $W^{SCC}_{\text{spec}} : A^{SCC}_{\text{spec}} \to \{0, 1\}$ as follows. For any $a = (S_1, S_2) \in A^{SCC}_{\text{spec}}$,

$$W^{SCC}_{\text{spec}}(a) = \max_{v_1 \in S_1, v_2 \in S_2 : (v_1, v_2) \in A^e_{\text{spec}}} W^e_{\text{spec}}((v_1, v_2)) \ . \tag{30}$$

   By definition, the graph G^{SCC}_{spec} is a directed acyclic graph (DAG).
5. As previously noted, the maximal path weights from any two vertices of V^e_{spec} in the same strongly connected components must be equal. Thus, we can define $d^{SCC}_{\text{spec}}(S)$ for any $S \in V^{SCC}_{\text{spec}}$ and give it the value of $d^e_{\text{spec}}(v)$ for any $v \in S$. There are three cases to consider:
   - For any $S \in V^{SCC}_{\text{spec}}$ such that there exist two vertices $v_1, v_2 \in S$ and an edge $a = (v_1, v_2)$ with weight 1, assign $d^{SCC}_{\text{spec}}(S) = \infty$, since this edge can be traversed an infinite number of times in the maximal weight path starting from any $v \in S$.
   - For any $S_1 \in V^{SCC}_{\text{spec}}$ that has a path to some $S_2 \in V^{SCC}_{\text{spec}}$ in G^{SCC}_{spec} with $d^{SCC}_{\text{spec}}(S_2) = \infty$, assign $d^{SCC}_{\text{spec}}(S_1) = \infty$, since there exists a path from any $v_1 \in S_1$ to any $v_2 \in S_2$ in this case (and hence the weight of the maximal weight path from any such v_1 is at least as great as that of the maximal weight path from any such v_2).
   - For the remainder of the vertices of G^{SCC}_{spec}, we can do a topological sort, assign $d^{SCC}_{\text{spec}}(S) = 0$, for any $S \in V^{SCC}_{\text{spec}}$ that is a sink of G^{SCC}_{spec}, and work backwards from these to compute the remaining walues of $d^{SCC}_{\text{spec}}(S)$.

Proposition 1 (Running Time of Extended Specification Computation). *The running time of the procedure described in this section is $O(|X|^2|E|)$ if G is deterministic.*

Remark 4. If G is not deterministic, it can be shown that the running time becomes $O(|X|^4|E|)$, which occurs in the case that there exists a transition in G from every state to every other state, one for each event. □

6.2 Computing the MPO

In this section, we provide an algorithm for the computation of the MPO and determine its running time. We also give a simple example in which we construct the MPO.

The basic outline of an implemented algorithm for constructing the MPO is shown in algorithm 1. The algorithm simply performs a depth-first search (DFS). The parameter G represents the finite-state automaton, the parameter y is a Y state, and the parameter E contains the set of events E_o ($E.eo$ in the algorithms), E_s ($E.es$ in the algorithms), E_{uo} and E_f. Algorithm 1 searches through the space of Y states and, for each encountered Y state y, finds the safe control decisions. Finding the safe control decisions is done in lines 1-12. This is accomplished by considering each subset of events $el \subseteq E.es$, and determining whether it is safe to choose to monitor only the events $el \cup E.eo$. This determination is made by a call to DeI, which simply computes the value of $D_I^e(RUR(y, el \cup E.eo))$. This can be very efficiently computed, thanks to the linear time algorithm for computing the reduced unobservable reach presented in Appendix 1 and the reduced format for the extended specification from Sect. 5. If the control decision is safe, it is added to the list sl (state list) and y is marked as safe. Traversing the space of Y states is done on lines 13-21. This is accomplished by considering all safe control decisions of the current y state, determining the next Z state and, for each such Z state, computing all possible successor Y states and making a recursive call. Since there are a finite number of augmented states with count at most $K+1$, there are a finite number of information states that will be traversed and the algorithm must eventually terminate. The initial call to the algorithm (not shown here) is: DoDFS$(G, y_0, K, \emptyset, E)$.

Algorithm 1 Algorithm for constructing the MPO

```
 1: procedure DoDFS(G, y, Tespec, sl, E)
 2:     for all el ⊆ E.es do                          ▷ Try all subsets of events
 3:         ur ← GetRUR(y, E.eo ∪ el)   ▷ Get reduced unobservable reach for next z
    state
 4:         if DeI(ur, Tespec) = true then
 5:             Add (y, E.eo ∪ el) to sl   ▷ Control decision E.eo ∪ el in state y is safe
 6:         end if
 7:     end for
 8:     if y is not marked "safe" then
 9:         Mark y as "unsafe"
10:     end if
11:     for all el ⊆ es s.t. (y, E.eo ∪ el) ∈ sl do      ▷ Try all safe control decisions
12:         ur ← GetRUR(y, E.eo ∪ el)   ▷ Get reduced unobservable reach for next z
    state
13:         for all e ∈ E.eo ∪ el do                              ▷ Try all events
14:             next ← Next(ur, e)                    ▷ Get next reduced y state
15:             if next not marked then
16:                 DoDFS(G, next, Tespec, sl, E)
17:             end if
18:         end for
19:     end for
20: end procedure
```

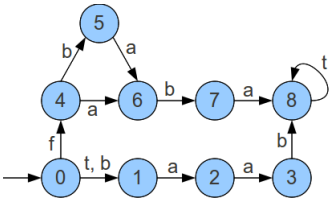

Fig. 3. A finite state automaton. Events are classified as follows: $E_o = \emptyset$, $E_s = \{a, b\}$, $E_{uo} = \{t\}$, and $E_f = \{f\}$.

Proposition 2 (Running time of DoDFS). *The running time of algorithm 1 is in* $O([(2(K+2))^{|X|}][2^{|E_s|}][|X|^2])$.

Remark 5 (Effect of reduced information states on running time). Recall from the end of Sect. 5 that the size of the state space changes from $2^{(K+2)|X|}$ to $(2(K+2))^{|X|}$ by using the reduced information state. For a small value of K, say $K = 2$, this means algorithm 1 can run as fast with an automaton that is twice as large. These savings are amplified for larger values of K. For $K = 5$, the above algorithm 1 can run as fast with an automaton that is nine times as large. □

Remark 6 (Multiple faults). If there are multiple different faults to diagnose, it is actually preferable to define mutiple different MPOs, one for each fault, rather than defining a single one that diagnoses all faults, since the latter method would require keeping a separate count for each type of fault event, which would dramatically increase the size of the state space. □

Following is a second MPO example that demonstrates the usefulness of the reduced information state:

Example 2 (MPO with reduced information state).

Consider the automaton of Fig. 3. The entire extended specification consists of $2|X|^2 = 162$ values. Of these, only four actually restrict behavior: $m(0, 6) = \infty$, $m(3, 6) = 1$, $m(2, 7) = \infty$, and $m(8, 8) = \infty$. These values can be determined by inspection for this example, by computing longest strings with the same projection for each pair. The remaining 158 values are either irrelevant because the corresponding pair of states does not occur, or are relevant but do not require any more events to be observed. If we choose not to monitor event a initially, the unobservable reach will include both augmented states $(0, -1)$ and $(6, 1)$, which causes a lack of K-diagnosability since $m(0, 6) = \infty$. From Y states y_1, y_2, and y_4, we must monitor event a since the unobservable reach from any of these Y states would otherwise include both augmented states $(3, -1)$ and $(6, 2)$, and $m(3, 6) = 1$. From Y states y_2, y_3, and y_4, we must monitor event b since the unobservable reach from any of these Y states would otherwise include both augmented states $(2, -1)$ and one of $(7, 1)$ or $(7, 2)$, and $m(2, 7) = \infty$. Finally, it is also necessary to monitor event a from Y state y_5, since the unobservable

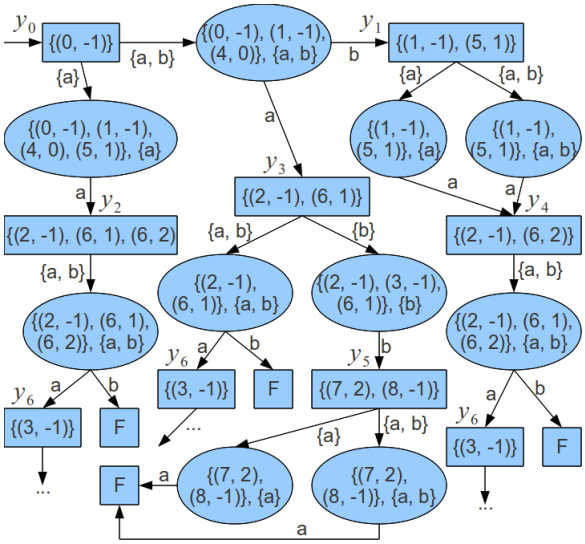

Fig. 4. The MPO corresponding to the automaton of Fig. 3, with $K = 2$.

reach would otherwise include both augmented states $(8, -1)$ and $(8, 3)$, and $m(8, 8) = \infty$. As in the MPO of Ex. 1, we omitted the portion of the MPO after which we can determine that no fault has occurred in the past and none can occur in the future.

In fact, the MPO could have been "guessed" by comparing this example to Ex. 1. In Ex. 1, the non-faulty labguage is abt^* whereas the faulty language is $fbat^*$. In this example, the non-faulty language is $(\varepsilon + b)aabt^*$ whereas the faulty language is $f(\varepsilon + b)abat^*$. Thus, once we have observed a first occurence of event a, the structure of the MPO from that point on should be similar to the structure of the MPO in Ex. 1. Indeed, upon observing event a we find ourselves in one of Y states y_2, y_3, or y_4. If we have potentially reached an augmented state with a count of 2 (i.e., in y_2 or y_4), then the MPO from these states onwards has the same structure as the MPO of Ex. 1, but with $K = 0$. If, on the other hand, at most one event has occurred after a fault (i.e., in y_3), then the MPO from this state onwards has the same structure as the MPO of Ex. 1 (i.e., with $K = 1$). Notice that MPO states y_2 and y_4 have the same reduced versions (namely $(2, -1), (6, 2)$) and the same structure from that point. On the other hand, MPO state y_3 is also very similar to y_2 and y_4, but has a different reduced version (namely $(2, -1), (6, 1)$) and therefore a different structure from that point on. □

7 Conclusion

This paper considered the problem of dynamic fault diagnosis under the constraint of maintaining K-diagnosability. We presented a structure called the MPO that contains all the solutions of the problem, developped from our notion of the information state. After recalling from previous work how the problem of finding safe controllers can be mapped to the state disambiguation problem, and showing a equivalence between safety and satifying the extended specification, we proceeded to prove a monotonicity result on the extended specification that allows us to reduce our information state and the size of our MPO. Finally, we described how to efficiently compute this extended specification and presented an algorithm for computing the MPO. In future work, we will concentrate on finding a single optimal controller (according to some numerical cost criterion), using the MPO as a basis and present simlation results demonstrating the efficiency of our algorithms when applied in practice.

Acknowledgment

The work of the authors is supported in part by NSF grant CNS-0930081 and by a fellowship from Fonds FQRNT, Government of Québec, Canada. This work also benefited from useful discussions with Franck Cassez and Tae-Sic Yoo.

References

1. F. Cassez and S. Tripakis. Fault diagnosis with static and dynamic observers. *Fundamenta Informaticae*, 88(4):497–540, 2008.
2. E. Dallal and S. Lafortune. On most permissive observers in dynamic sensor optimization problems for discrete event systems. In *Proceedings of the 48th Annual Allerton Conference on Communication, Control, and Computing*, September 2010.
3. E. Dallal and S. Lafortune. A framework for optimization of sensor activation using most permissive observers. Technical report, University of Michigan, 2011.
4. P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81 –98, January 1989.
5. M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, September 1995.
6. D. Thorsley and D. Teneketzis. Active acquisition of information for diagnosis and supervisory control of discrete event systems. *Discrete Event Dynamic Systems*, 17(4):531–583, 2007.
7. W. Wang, S. Lafortune, A.R. Girard, and F. Lin. Optimal sensor activation for diagnosing discrete event systems. *Automatica*, 46(7):1165 – 1175, 2010.
8. Weilin Wang, S. Lafortune, Feng Lin, and A.R. Girard. An online algorithm for minimal sensor activation in discrete event systems. In *Proceedings of the 48th IEEE Conference on Decision and Control, CDC/CCC 2009*, pages 2242 –2247, December 2009.
9. Tae-Sic Yoo and Humberto E. Garcia. Diagnosis of behaviors of interest in partially-observed discrete-event systems. *Systems & Control Letters*, 57(12):1023 – 1029, 2008.

Appendix 1: Computing the Reduced Unobservable Reach

In this section, we show how to efficiently compute the reduced unobservable reach (i.e., how to compute $rur = RUR(y, C(y))$), so that the running time of this computation does not surpass that of verifying if the extended spefication is satisfied. The procedure for this bears some similarity to the one used to compute the extended specification in that we also make use of strongly connected components and the topological sort. We assume that y is a reduced information state (i.e., $y = R(y)$). We also assume that there is a single fault event, namely e_f. For notational convenience, let $M(x; y, C(y))$ denote the maximal count for state $x \in X$ in the unobservable reach $UR(y, C(y))$. That is, $M(x; y, C(y)) = \max\{n \in \{-1, 0, 1, \ldots\} : (x, n) \in UR(y, C(y))\}$, or $-\infty$ if this value is undefined (i.e., if $\nexists n : (x, n) \in UR(y, C(y))$). Finally, we assume that the computation of the unobservable reach is a step in determining whether or not a particular control decision is safe.

1. Create the graph $G^{NF}_{C(y)} = (X, A^{NF}_{C(y)})$, where $A^{NF}_{C(y)} \subseteq X \times X$ is the set of (directed) edges. The set of edges $A^{NF}_{C(y)}$ corresponds simply to all unobservable, non-faulty transitions of the automaton G, given the set of monitored events $C(y)$. That is, $A^{NF}_{C(y)} = \{(x_1, x_2) \in X^2 : \exists e \in E \setminus (C(y) \cup E_f) \text{ s.t. } x_2 = f(x_1, e)$.
2. Initialize $rur \leftarrow y$. For all $u \in y$ such that $N(u) = -1$, determine all $x \in X$ such that there exists a path from $S(u)$ to x in $G^{NF}_{C(y)}$ and set $rur \leftarrow rur \cup \{(x, -1)\}$. This can be done through a single depth-first search on the graph $G^{NF}_{C(y)}$ and gives the set of all $v \in rur$ such that $N(v) = -1$.
3. For each $u \in rur$ such that $N(u) = -1$ and for each $x \in X$ such that $f(S(u), e_f) = x$, set $rur \leftarrow rur \cup \{(x, 0)\}$, unless there exists some $v \in rur$ satisfying $S(v) = x$ and $N(v) > 0$.
4. Create the graph $G^{F}_{C(y)} = (X, A^{F}_{C(y)})$, where $A^{F}_{C(y)} \subseteq X \times X$ is the set of (directed) edges. The set of edges $A^{F}_{C(y)}$ corresponds simply to all unobservable transitions of the automaton G (including faulty ones), given the set of monitored events $C(y)$. That is, $A^{F}_{C(y)} = \{(x_1, x_2) \in X^2 : \exists e \in E \setminus C(y) \text{ s.t. } x_2 = f(x_1, e)$.
5. Create the graph $G^{SCC}_{C(y)} = (V^{SCC}_{C(y)}, A^{SCC}_{C(y)})$ of strongly connected components (SCCs) from $G^{F}_{C(y)}$. For each $x \in X$, let $SCC_{C(y)}(x)$ denote the SCC that contains x. For each strongly connected component, the maximal counts are the same. That is, $M(x; y, C(y))$ is the same for all $x \in S$, for each $S \in V^{SCC}_{C(y)}$. This is obvious if $|S| = 1$. If $|S| > 1$ and there exists some $x \in S$ such that $(x, n) \in UR(y, C(y))$ for some $n \geq 0$, then it is possible to form an infinite length path from x to each node $x' \in SCC_{C(y)}(x)$, in which case the maximal counts are all infinite. Otherwise, the maximal counts will either all be -1, or all be undefined. We can therefore define $M^{SCC}(S; y, C(y))$ to be equal to the value of $M(x; y, C(y))$ for any $x \in S$, for each $S \in V^{SCC}_{C(y)}$. Also, for any singleton $S_1 = \{x_1\} \in V^{SCC}_{C(y)}$ such that $(x_1, n) \in rur$ for some

$n \geq 0$ and there exists a path to some $S_2 \in V^{SCC}_{C(y)}$ satisfying $|S_2| > 1$, set $M^{SCC}(S_1; y, C(y)) = \infty$ as well. Thus, if there exists any $S \in V^{SCC}_{C(y)}$ with $M^{SCC}(S; y, C(y)) = \infty$, then we know that an infinite number of events can occur after a fault, and hence we stop here and conclude that $C(y)$ is not a safe control decision from Y state y.

6. Suppose that there exists no $S \in V^{SCC}_{C(y)}$ with $M^{SCC}(S; y, C(y)) = \infty$. Topologically sort the vertices of $G^{SCC}_{C(y)}$. For each source node $S \in V^{SCC}_{C(y)}$, set $M^{SCC}(S; y, C(y)) = \max\{n \in \{-1, 0, \ldots\} : \exists x \in S \text{ s.t. } (x, n) \in rur\}$, or $-\infty$ if this value is undefined. By considering the nodes in topologically sorted order, starting at source nodes, the remaining values of M^{SCC} can be computed. Finally, for each $x \in X$ such that $M(x; y, C(y)) \neq -\infty$, set $rur \leftarrow rur \setminus \{(x, n) : n \in \{0, 1, \ldots, \}\} \cup \{(x, M(x; y, C(y)))\}$.

Proposition 3 (Running Time of Reduced Unobservable Reach Computation). *The running time of the procedure described in this section is $O(|X||E|)$ if G is deterministic.*

Appendix 2: Proofs

Lemma 1 (Relation between projection and information state). *For any string s and controller C, $I(IS^Z_C(s)) = \{v \in X^+ : \exists s' \text{ s.t. } P_C(s) = P_C(s') \wedge v \in g(s')\}$.*

Proof. The proof is established by induction on the length of $P_C(s)$. Let $|P_C(s)| = n$. Furthermore, for any string t, let $t[k]$ denote the k^{th} event in t, and let $t(k)$ denote the substring $t[1] \cdots t[k]$, with $t(0) = \varepsilon$. As further shorthand, let $s_k = P_C(s)(k)$ for $k = 1, \ldots, n$ and $e_k = P_C(s)[k+1]$ for $k = 0, \ldots, n-1$, so that $s_0 = \varepsilon$, $s_1 = e_0$, etc... Define y_0 as usual. For $k = 0, \ldots, n$, let $z_k = h_{YZ}(y_k, C(y_k))$ and for $k = 0, \ldots, n-1$, define $y_{k+1} = h_{ZY}(z_k, e_k)$. Finally, for $k = 0, \ldots, k$, define C_k by $C_k = C(y_k)$. First, notice that since unobserved events do not change the information state, we have $IS^Z_C(s_k) = z_k$ and, in particular, $z = IS^Z_C(s) = IS^Z_C(P_C(s)) = IS^Z_C(s_n) = z_n$. Thus, the inductive hypothesis will be that:

$$I(z_k) = \{v \in X^+ : \exists s'_k \text{ s.t. } P_C(s'_k) = s_k \wedge v \in g(s'_k)\}, \tag{31}$$

where we have dropped the $P_C(\cdot)$ around s_k since s_k is already a projection. For the base case z_0, we have that:

$$\begin{aligned} I(z_0) = h_{YZ}(y_0, C_0) &= \{v \in X^+ : (\exists u \in y_0)(\exists t \in (E \setminus C_0)^*) \text{ s.t. } v \in g(u, t)\} \\ &= \{v \in X^+ : \exists t \in (E \setminus C_0)^* \text{ s.t. } v \in g(x_0^+, t)\} \\ &= \{v \in X^+ : \exists t \text{ s.t. } P_C(t) = \varepsilon = s_0 \wedge v \in g(t)\} \end{aligned}$$

Thus the base case is established, by taking $s'_k = t$. Now suppose that the inductive hypothesis is true at k. Then:

$$\begin{aligned}
y_{k+1} &= h_{ZY}(z_k, e_k) = \{v \in X^+ : \exists u \in I(z_k) \text{ s.t. } v \in g(u, e_k)\} \\
&= \{v \in X^+ : \exists s'_k \text{ s.t. } P_C(s'_k) = s_k \wedge v \in g(s'_k e)\} \\
z_{k+1} &= h_{YZ}(y_{k+1}, C_{k+1}) = \{v \in X^+ : (\exists u \in y_{k+1})(\exists t \in (E \setminus C_{k+1})^*) \text{ s.t. } v \in g(u,t)\} \\
&= \{v \in X^+ : (\exists s'_k)(\exists t \in (E \setminus C_{k+1})^*) \text{ s.t. } P_C(s'_k) = s_k \wedge v \in g(s'_k e_k t)\} \\
&= \{v \in X^+ : \exists s'_{k+1} \text{ s.t. } P_C(s'_{k+1}) = s_{k+1} \wedge v \in g(s'_{k+1})\},
\end{aligned}$$

where the last equality follows by taking $s'_{k+1} = s'_k e_k t$ and noting that, since s'_k can be any string satisfying $P_C(s'_k) = s_k$ and t can be any string satisfying $t \in (E \setminus C_{k+1})^*$ (which is equal to the set $\{t : P_C(z_k, t) = \varepsilon\}$), the concatenation $s'_{k+1} = s'_k e_k t$ can be any string satisfying $P_C(s'_{k+1}) = s_k e_k = s_{k+1}$. Thus the induction step is proven and the lemma follows from this. □

Theorem 1 (Formulation of the K-diagnosability property through the information state). *Controller C maintains K-diagnosability if and only if $D_I(I(z)) = 1$, for all reachable Z states. Mathematically:*

$$\begin{aligned}
&\exists s \in \mathcal{L}(G) : z = IS_C^Z(s) \wedge \exists u, v \in I(z) \text{ s.t. } N(u) = -1 \text{ and } N(v) = K+1 \\
\Leftrightarrow\ &\exists s_Y, s_N \in \mathcal{L}(G) : P_C(s_Y) = P_C(s_N) \wedge N(g(s_N)) = -1 \text{ and } N(g(s_Y)) = K+1.
\end{aligned}$$

Proof. ($\Leftarrow$) Since unobserved events cannot change the information state, we have that $IS_C^Z(s) = IS_C^Z(P_C(s))$. Thus, $P_C(s_Y) = P_C(s_N)$ implies $IS_C^Z(s_Y) = IS_C^Z(s_N) = z$. By definition, $g(s) \in I(IS_C^Z(s))$, for all s. Thus, $g(s_N) \in I(IS_C^Z(s_N)) = z$ and $g(s_Y) \in I(IS_C^Z(s_Y)) = z$ as well. We may therefore take $s = s_Y$, $u = g(s_N)$, and $v = g(s_Y)$.
($\Rightarrow$) Recall from Lem. 1 that $I(IS_C^Z(s)) = \{v \in X^+ : \exists s' \text{ s.t. } P_C(s) = P_C(s') \wedge v \in g(s')\}$. Then $u, v, \in I(z)$ implies that there exists s_1, s_2 such that $P_C(s_1) = P_C(s_2) = P_C(s)$, $u \in g(s_1)$, and $v \in g(s_2)$. We simply take $s_N = s_1$ and $s_Y = s_2$. □

Theorem 2 (Monotonicity Property for the Extended Specification). *Suppose that $(u, v) \in T^e_{spec}$. Then, for any v' such that $S(v') = S(v)$ and $N(v') > N(v)$, we also have $(u, v') \in T^e_{spec}$.*

Proof. Suppose that $(u, v) \in T^e_{\text{spec}}$. Then:

$$\exists s_1, s_2 : P(s_1) = P(s_2), \quad (g(u, s_1), g(v, s_2)) \in T_{\text{spec}} \ .$$

We know that $N(g(u, s_1)) = -1$ and $N(g(v, s_2)) = K + 1$.
Case 1 ($N(v) \geq 0$):
In this case, we know that $|s_2| = K + 1 - N(v)$. Let $v' \in X^+$ be such that $S(v') = S(v)$ and $N(v') = N(v) + c$, where $c \in \{1, \ldots, K + 1 - N(v)\}$. Then $N(g(v', s_2)) = K + 1 + c$. Then define s'_2 to be s_2, shortened by c events (this is possible since $c \leq K + 1 - N(v) = |s_2|$). Next, take s'_1 to be equal to s_1 shortened by an appropriate number of events such that $P(s'_1) = P(s'_2)$. Then

s_1' and s_2' satisfy the three conditions that $P(s_1') = P(s_2')$, $N(g(u, s_1')) = -1$, and $N(g(v', s_2')) = K + 1$. Thus, we can conclude that $(u, v') \in T^e_{\text{spec}}$.
Case 2 ($N(v) = -1$):
In this case, we know that $|s_2| \geq K + 2$ (s_2 is of the form $(E \setminus \{E_f\})^* E_f E^{K+1}$). We proceed as before. Let $v' \in X^+$ be such that $S(v') = S(v)$ and $N(v') = c$, where $c \in \{0, \ldots, K + 1\}$. Define s_2' to be the first $K + 1 - c$ events of s_2 (this is possible since $K + 1 - c \leq K + 1 < |s_2|$). Next, take s_1' to be equal to s_1 shortened by an appropriate number of events such that $P(s_1') = P(s_2')$. Then s_1' and s_2' satisfy the three conditions that $P(s_1') = P(s_2')$, $N(g(u, s_1')) = -1$, and $N(g(v', s_2')) = K + 1$. Thus, we can conclude that $(u, v') \in T^e_{\text{spec}}$. □

Corollary 1 (Reduced information state carries all necessary information in determining safety). *For any information state $i \in I$, $D^e_I(i) = D^e_i(R(i))$.*

Proof. Obviously, $R(i) \subseteq i$, so that $D^e_I(R(i)) \geq D^e_I(i), \quad \forall i \in I$. Now suppose to the contrary that for some $i \in I$, $D^e_I(i) \neq D^e_i(R(i))$. Then it must be that $D^e_i(R(i)) = 1$ and $D^e_I(i) = 0$. Thus, $\exists u, v \in i : (u, v) \in T^e_{\text{spec}}$. Since $N(u) = -1$, we know that $u \in R(i)$ as well. Then it must be that $v \notin R(i)$. Hence $N(v) \neq -1$ and $\exists v' \in X^+$ such that $S(v') = S(v)$ and $N(v') > N(v)$. Furthermore, there is one such v' that is also in $R(i)$. But by Thm. 2, we know that $(u, v') \in T^e_{\text{spec}}$, so that $D^e_i(R(i)) = 0$ also, a contradiction. □

Lemma 2. *Let $i \in I$ be any finite information state and let Q be any boolean-valued function of an augmented state that is monotonic in count. That is, $Q : X^+ \to \{\mathbf{T}, \mathbf{F}\}$ satisfies $Q(u) \Rightarrow Q(u')$, for all u' such that $S(u') = S(u)$ and $N(u') \geq N(u)$. Then the following equivalence holds:*

$$\exists u \in i : Q(u) \Leftrightarrow \exists u \in R(i) : Q(u) \tag{32}$$

Proof. ($\Leftarrow$) Clearly, $R(i) \subseteq i$, so that $u \in R(i) \Rightarrow u \in i$.
($\Rightarrow$) Suppose that $\exists u \in i : Q(u)$. Define u' by $S(u') = S(u)$ and $N(u') = \max_{v \in i : S(v) = S(u)} N(v)$. By the monotonicity property of Q, $Q(u')$ holds. But by Def. 20, $u' \in R(i)$ as well. □

Theorem 3 (There is no loss in applying h^R_{ZY} to a reduced Z state). *For any Z state z and event $e \in C(z)$, $h^R_{ZY}(z, e) = h^R_{ZY}(R(z), e)$.*

Proof. We prove this by showing that, for any Z state z and any $e \in E_o \cup E_s \supseteq C(z)$, $v \in h^R_{ZY}(z, e) \Leftrightarrow v \in h^R_{ZY}(R(z), e)$. We consider two cases, depending on whether v has a count of -1 or a maximal count:

$$\begin{aligned}
& v \in h^R_{ZY}(z, e) \wedge N(v) = -1 \\
\Leftrightarrow\ & v \in h_{ZY}(z, e) \wedge N(v) = -1 \\
\Leftrightarrow\ & \exists u \in I(z) : v = g(u, e) \wedge N(u) = -1 \\
\Leftrightarrow\ & \exists u \in I(R(z)) : v = g(u, e) \wedge N(u) = -1 \\
\Leftrightarrow\ & v \in h_{ZY}(R(z), e) \wedge N(v) = -1 \\
\Leftrightarrow\ & v \in h^R_{ZY}(R(z), e) \wedge N(v) = -1
\end{aligned}$$

The first, third, and fifth equivalences follow from Def. 20 whereas the second and fourth equivalences follow by taking u to be the predecessor of v (or v the successor of u).

$$\begin{aligned}
& v \in h^R_{ZY}(z,e) : \nexists v' \in h^R_{ZY}(z,e) \text{ s.t. } S(v') = S(v) \wedge N(v') > N(v) \\
\Leftrightarrow\ & v \in h_{ZY}(z,e) : \nexists v' \in h_{ZY}(z,e) \text{ s.t. } S(v') = S(v) \wedge N(v') > N(v) \\
\Leftrightarrow\ & \exists u \in I(z) : f(S(u),e) = S(v) \wedge \nexists u' \in I(z) \text{ s.t. } f(S(u'),e) = S(v) \wedge N(u') > N(u) \\
\Leftrightarrow\ & \exists u \in I(R(z)) : f(S(u),e) = S(v) \wedge \nexists u' \in I(R(z)) \text{ s.t. } f(S(u'),e) = S(v) \wedge N(u') > N(u) \\
\Leftrightarrow\ & v \in h_{ZY}(R(z),e) : \nexists v' \in h_{ZY}(R(z),e) \text{ s.t. } S(v') = S(v) \wedge N(v') > N(v) \\
\Leftrightarrow\ & v \in h^R_{ZY}(R(z),e) : \nexists v' \in h^R_{ZY}(R(z),e) \text{ s.t. } S(v') = S(v) \wedge N(v') > N(v)
\end{aligned}$$

The first and fifth equivalences follow from Def. 20, the second and fourth equivalences follow by taking u to be the predecessor of v (or v the successor of u), and the third equivalence follows by two applications of Lem. 2, first to the inner ($\nexists$) expression and then to the outer ($\exists$) expression. □

Theorem 4 (There is no loss in applying h^R_{YZ} to a reduced Y state). *For any Y state y and control decision $C(y)$, $h^R_{YZ}(y,C(y)) = h^R_{YZ}(R(y),C(y))$.*

Proof. We prove this by showing that, for any Y state y, $v \in RUR(y,C(y)) \Leftrightarrow v \in RUR(R(y),C(y))$. As before, We consider two cases, depending on whether v has a count of -1 or a maximal count:

$$\begin{aligned}
& v \in RUR(y,C(y)) \wedge N(v) = -1 \\
\Leftrightarrow\ & v \in UR(y,C(y)) \wedge N(v) = -1 \\
\Leftrightarrow\ & \exists u \in y, t \in (E \setminus (C(y) \cup E_f))^* : v = g(u,t) \wedge N(u) = -1 \\
\Leftrightarrow\ & \exists u \in R(y), t \in (E \setminus (C(y) \cup E_f))^* : v = g(u,t) \wedge N(u) = -1 \\
\Leftrightarrow\ & v \in UR(R(y),C(y)) \wedge N(v) = -1 \\
\Leftrightarrow\ & v \in RUR(R(y),C(y)) \wedge N(v) = -1
\end{aligned}$$

The first, third, and fifth equivalences follow from Def. 20 whereas the second and fourth equivalences follow by taking u to be the predecessor of v (or v the successor of u).

$$\begin{aligned}
& v \in RUR(y,C(y)) : \nexists v' \in RUR(y,C(y)) \text{ s.t. } S(v') = S(v) \wedge N(v') > N(v) \\
\Leftrightarrow\ & v \in UR(y,C(y)) : \nexists v' \in UR(y,C(y)) \text{ s.t. } S(v') = S(v) \wedge N(v') > N(v) \\
\Leftrightarrow\ & \exists u \in y, t \in (E \setminus C(y))^* : \begin{array}{c} f(S(u),t) = S(v) \wedge \nexists u' \in y, t' \in (E \setminus C(y))^* \\ \text{s.t. } f(S(u'),t') = S(v) \wedge N(g(u',t')) > N(g(u,t)) \end{array} \\
\Leftrightarrow\ & \exists u \in R(y), t \in (E \setminus C(y))^* : \begin{array}{c} f(S(u),t) = S(v) \wedge \nexists u' \in R(y), t' \in (E \setminus C(y))^* \\ \text{s.t. } f(S(u'),t') = S(v) \wedge N(g(u',t')) > N(g(u,t)) \end{array} \\
\Leftrightarrow\ & v \in UR(R(y),C(y)) : \nexists v' \in UR(R(y),C(y)) \text{ s.t. } S(v') = S(v) \wedge N(v') > N(v) \\
\Leftrightarrow\ & v \in RUR(R(y),C(y)) : \nexists v' \in RUR(R(y),C(y)) \text{ s.t. } S(v') = S(v) \wedge N(v') > N(v)
\end{aligned}$$

The first and fifth equivalences follow from Def. 20, the second and fourth equivalences follow by taking u to be the predecessor of v (or v the successor of u),

and the third equivalence follows by two applications of Lem. 2, first to the inner ($\nexists$) expression and then to the outer ($\exists$) expression. □

Proposition 1 (Running Time of Extended Specification Computation). *The running time of the procedure described in this section is $O(|X|^2|E|)$ if G is deterministic.*

Proof. The graph created in step 1 has $|V^e_{\text{spec}}| = 2|X|^2$ vertices. For a deterministic automaton, there is at most one transition defined for a given initial state and event. Thus, for each of the $2|X|^2$ vertices of G^e_{spec}, there is at most one outgoing edge for each event $e \in E_o \cup E_s$ (labeled (e, e)), at most two outgoing edges for each event $e \in E_{uo}$ (labeled either (e, ε) or (ε, e)), and at most one outgoing edge for each event $e \in E_f$ (labeled (ε, e)). Hence, the graph created in step 1 has $|A^e_{\text{spec}}| \leq [2|X|^2][2|E|] = 4|X|^2|E|$ edges. Step 2 consists only of labeling edges and is done at the same time as step 1. Finding the strongly connected components in step 3 can be done in time $O(|V^e_{\text{spec}}|+|A^e_{\text{spec}}|)$ [reference?]. Creating G^{SCC}_{spec} in step 4 can also be done in linear time, i.e., $O(|V^e_{\text{spec}}|+|A^e_{\text{spec}}|)$. Furthermore, $|V^{SCC}_{\text{spec}}| \leq |V^{SCC}_{\text{spec}}| = 2|X|^2$ and $|A^{SCC}_{\text{spec}}| = |V^{SCC}_{\text{spec}}| - 1 < 2|X|^2$ The first part of step 5 can be done with step 4 and thus takes no additional time. The second part of step 5 can be done through a single depth-first search on G^{SCC}_{spec}, and hence takes $O(|X|^2)$ time. For the last part of step 5, finding a topological sort can be done at the same time as determining strongly connected components. The remainder of the algorithm takes linear time in the size of G^{SCC}_{spec} by considering this graph's vertices in topologically sorted order, starting from sink nodes. Thus, the total running time is in $O(|X|^2|E|)$. □

Proposition 2 (Running time of DoDFS). *The running time of algorithm 1 is in $O([(2(K+2))^{|X|}][2^{|E_s|}][|X|^2])$.*

Proof. There are $(2(K+2))^{|X|}$ reduced information states. For each information state encountered, a maximum of $2^{|E_s|}$ control decisions are considered. Finally, for each control decision, it is necessary to compute $D^e_I(i)$, where $i = I(h^R_{YZ}(y, C(y)))$. By using an efficient encoding, this can be done in time $O(|X|^2)$. □

Proposition 3 (Running Time of Reduced Unobservable Reach Computation). *The running time of the procedure described in this section is $O(|X||E|)$ if G is deterministic.*

Proof. The graph created in step 1 has $|X|$ vertices and at most $|E \setminus (C(y) \cup E_f)| \leq |E_s| + |E_{uo}| \leq |E|$ outgoing edges per vertex and can therefore be constructed in time $O(|X||E|)$. The depth-first search in step 2 takes linear time in the size of the graph $G_{C(y)}$. Step 3 requires considering all fault transitions in the automaton. Since G is deterministic, and there is only one fault event, there are at most $|X|$ such transitions and hence this step is done in $O(|X|)$ time. Similarly to $G^{NF}_{C(y)}$, the graph $G^F_{C(y)}$ created at the beginning of step 4 has size $O(|X||E|)$ and can be computed in time $O(|X||E|)$. In step 5, computing

strongly connected components (SCCs) can be done in linear time in the size of the graph $G^F_{C(y)}$. Checking if there exists any $x \in X$ satisfying $(x, n) \in rur$ for some $n \geq 0$ and $|SCC_{C(y)}(x)| > 1$ takes time $O(|X|)$. Checking if there exists any singleton $S_1 = \{x_1\} \in V^{SCC}_{C(y)}$ satisfying $(x_1, n) \in rur$ for some $n \geq 0$ and there exists a path to some $S_2 \in V^{SCC}_{C(y)}$ such that $|S_2| > 1$ can be done through a depth-first search on $G^{SCC}_{C(y)}$. Topologically sorting vertices in step 6 can be done at the same time as finding strongly connected components. Finally, the computation of the M^{SCC} values can be achieved in linear time in the size of $G^{SCC}_{C(y)}$, by considering the nodes in topologically sorted order, starting from source nodes. The overall running time is therefore $O(|X||E|)$. □

Effectiveness of Transition Systems to Model Faults

Jingshu Chen and Sandeep Kulkarni

Michigan State University,
3115 Engineering Building, 48824 East Lansing, US
Email: {chenji15,sandeep}@cse.msu.edu
Web: http://www.cse.msu.edu/~{chenji15,sandeep}

Abstract. The goal of this paper is to bridge the gap between the theory and practice in fault-tolerant systems. Specifically, our goal is to model faults uniformly using transition systems so that techniques such as model checking and model revision can be applied effectively.
We begin with the taxonomy of faults that is based on the practitioner's view of fault-tolerant systems. For each of 31 categories in this taxonomy, we identify whether it is feasible to model faults in that category using transition systems. We argue that (1) such modeling is feasible and cost-effective for 18 categories; (2) Also, it is feasible but not cost-effective for 2 categories; (3) And, it is not feasible for the remaining 11 categories. The results in this paper provide an update on the (unproven) folk theorem about fault modeling while identifying its limitations.

Key words: Fault Modeling, Transition Systems, High Assurance, Fault Tolerance.

1 Introduction

The goal of this paper to evaluate the *effectiveness* of transition systems to model faults affecting different types of computer systems. Specifically, we find that there is a serious dichotomy about this topic among researchers as well as practitioners. In particular, some researchers take it as a folk theorem that any fault (whether a software fault or a hardware fault, transient fault or a permanent fault, detectable fault or an undetectable fault) could be modeled using transition systems. Regarding this aspect, some existing work (e.g., [5, 6, 10–12]) in literature has shown that used this approach for modeling certain specific types of faults. On the other hand, some researches view this folk theorem with skepticism. Some of the questions raised include the following

- How can physical faults be modeled by adding transitions? Faults such as stuck-at faults effectively remove transitions.
- While the use of transitions may be appropriate for hardware faults, how can it be used model faults such as buffer overflow?
- How can permanent faults be modeled as transitions? For example faults such as byzantine faults cannot be modeled using transitions since byzantine process may behave arbitrarily.
- How can faults such as failure of a processor (failstop) be modeled as transitions?
- Faults such as physical degradation are continuous in nature and, hence, not suitable for being modeled as *discrete* transition systems.

A second question regarding fault modeling is: *Even if it is* possible *to model faults using transition systems, is it* appropriate *and* reasonable. Specifically, this question focuses on the *complexity* of modeling faults in terms of transition systems. The complexity of modeling faults depends upon the desired objective in modeling faults. One of the main reasons for modeling faults is to enable verification and/or synthesis of fault-tolerant models/programs. Thus, there may be circumstances where modeling of faults in terms of transition systems may be feasible but expensive thereby making it difficult (or impossible) to utilize it.

A third question regarding fault modeling is: *How can we know if we have represented all types of faults that may affect a computer system?* This question is crucial since it characterizes the completeness of transitions systems to be used to model faults. While answering this question is beyond the scope of formal methods, we can consider this question in the context of a fault classification for practical systems.

In particular, we can begin with a classification of faults from the perspective of practitioners and then evaluate whether faults from those models can be effectively represented using transition systems.

Based on these questions, in this paper, we focus on the *feasibility* and *practicality* of modeling faults as transition systems. We begin with the third question. Specifically, we utilize the classification from seminal paper by Avizienis et al. [7]. This classification provides different causes of faults and their effects. Since this classification is based on practitioner's viewpoint, it provides the basis for answering the first two questions. Specifically, in this paper, we focus on how/if each category of fault identified here can be modeled in terms of transitions. From the point of view of formal methods, it is necessary to model the effect of the fault as opposed to the cause of the fault. Hence, our work will focus on how effect of faults from each category can be modeled.

Additionally, we also evaluate the complexity introduced by modeling faults as transition systems. We consider the complexity in terms of two objectives: *model checking* and *model revision*. Specifically, model checking [1, 2] is one of the most successful strategies for providing assurance for model of hardware and software design. It focuses on deciding whether a given model of system, say M satisfies the given property pr. Since model checking computes (directly or indirectly) all computations of M to determine whether pr is satisfied, it is especially useful in providing assurance about a system developed from that model. The related problem of model revision [3, 4] focuses on scenarios where model checking produces a counterexample or where an existing model needs to be revised to add new properties (such as safety, liveness and timing constraints). Thus, the goal in model revision is to modify the given model M so that it satisfies the given property pr. Since the revised model is correct by construction, it can assist us in obtaining a correct model of system when model checking ends up finding a counterexample.

We view the fault modeling in *online* setting where faults occur *during* system execution as opposed to *offline* setting where faults have already occurred at the beginning and the goal of the system is to provide acceptable service even in the presence of faults. This is due to the fact that in offline setting, often, there is no need to model faults explicitly; instead it suffices to model the system based on which components are still active. Hence, offline setting requires us to model a degraded version of the original system itself rather than modeling faults. In online setting however, system may initially execute without faults. Then, one or more faults could occur and change subsequent system behavior. Hence, in online setting, it is often necessary to model faults explicitly.

The main contributions of the paper are as follows:

- Of the 31 categories from [7], we show that faults from 20 categories can be modeled using transition systems. These faults include byzantine actions in a networked system, physical deterioration of brake in a braking control system and so on.
- We show that faults from 11 categories cannot (or should not be) represented using transition systems. These include faults such as buffer overflow, hardware errata and so on.
- We also describe the feasibility and practicability of modeling faults as transition systems. We show that (1) the modeling of faults from 18 categories as transition systems is practical and feasible and (2) the modeling of faults from 2 categories as transition systems is not practical although feasible.
- Besides, we discuss the relative completeness of the proposed approach with recent literature in the appendix.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we recall the classification of different categories of faults from [7]. In Section 3, we briefly define transition systems. In Section 4, we discuss feasibility of modeling of faults from different categories and its effect on model verification and revision. In Section 5, we identify practicability of modeling faults in terms of transition system. Related work is discussed in Section 6. Section 7 makes concluding remarks. Appendix discusses the relative completeness of the proposed approach with recent literature and identifies each fault with the corresponding approach.

2 A Taxonomy of Faults

In this section, we recall the terminology of fault classification from [7]. The classification is based on eight basic viewpoints about faults. Figure 1 illustrates this classification. This figure combines the classification

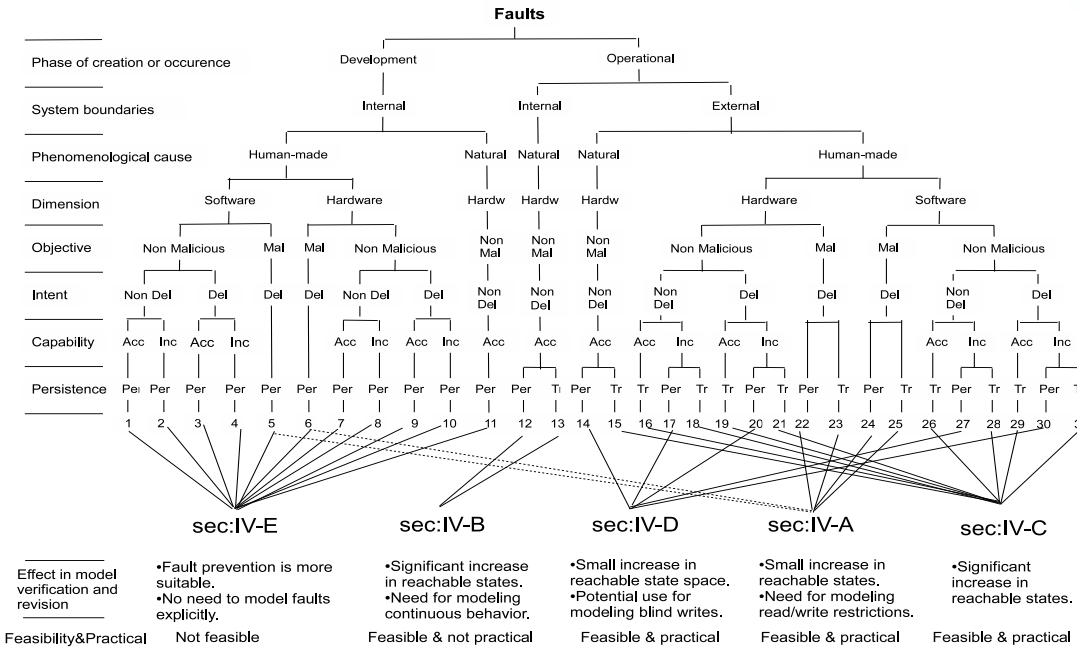

Fig. 1: Fault Categories

from [7] with results in this paper about the ability and effect of modeling faults from the respective category. In particular, for each category from [7], we discuss how to model it in Section 4. Then we discuss about the effect of such modeling in model checking and revision. We summarize the feasibility and practicability of modeling faults in transtion systems in 5.

Next, we briefly describe the viewpoints considered in this classification.

One viewpoint is based on how the fault occurs. In this viewpoint, there are two possibilities: **development faults** and **operational faults**. The former corresponds to the case where fault occurs due to mistakes during development. This category includes faults such as buffer overflows, incorrect results of certain floating point division caused by Pentium FDIV bug etc. The latter corresponds to the case where fault occurs while the system is deployed.

The second viewpoint is based on whether the fault occurs inside system boundary (**internal faults**) or whether it occurs outside system boundary, i.e., in the environment (**external faults**). The former corresponds to faults such as physical deterioration of brakes in the vehicle and logic bomb. And, the latter corresponds to faults such as bit flip in memory caused by cosmic ray and wrong parameter configuration.

The third viewpoint is based on the cause of the fault. In this viewpoint, a fault can be either **natural fault** or **human made fault**. The former corresponds to random events that may occur naturally. Examples of such faults include internet and telecoms connectivity disrupted by Taiwan earthquake. The latter corresponds to (intentional or otherwise) mistakes caused by humans. Examples of such faults include development failure of the AAS system [29].

The fourth viewpoint is based on how the fault affects the system. In this viewpoint, a fault can be either **hardware fault** or a **software fault**. Examples of former include loss of network switch and deflated car tie whereas examples of latter include Y2038 problem of software system and Trojan horses.

The fifth viewpoint is based on the objective of the fault. In this viewpoint, the fault can be either a **malicious fault** or a **non-malicious fault**. In the former case, the goal of the (human responsible for the) fault is to intentionally disrupt the system execution. Examples of such faults include attacker and worm. In the latter case, there is no malicious objective. Examples of such faults include physical deterioration and heating/cooling caused by natural environments.

The sixth viewpoint is based on in the intent of the fault. In this viewpoint, the fault can be either a **deliberate fault** or a **non-deliberate fault**. The former case is due to bad decisions. Examples of such faults include wrong configuration that can affect security, networking, storage, middleware, etc [30]. The latter one is caused by mistakes. Examples of such faults include software flaws and physical production defects.

The seventh viewpoint is based on whether the fault is caused **accidently** or due to **incompetence**. And, finally, the eighth viewpoint is based on whether the fault is a **transient fault** that occurs occasionally and does not persist for a long duration. Or a **persistent fault** where the fault persists for a long time (possibly forever).

3 Transition Systems

Since our goal is to demonstrate the use of transition systems to describe faults, we briefly define them next. Our formulation is based on the use of guarded commands [48] where a guarded command is of the form $g \longrightarrow st$ where g identifies constraints under which the guarded command can be executed and st identifies the effect of such execution. Thus, guarded commands are suitable for programs that are 'event-based' where g identifies the event and st identifies the action taken by the program while responding to the event. Guarded commands are also suitable for 'time-based' programs where g corresponds to a clock tick and st denotes the action taken by the program for that clock tick.

As mentioned in the Introduction, we chose this representation as it can be easily utilized in model checkers such as SPIN, SMV, PRISM, UPPAAL, etc [1, 2, 31, 32] as well as model revision tools such as SYCRAFT [4].

Given a program and a fault where both are specified in terms of one or more guarded commands, concurrent execution of the program and faults corresponds to interleaved execution guarded commands from both the program and faults. We do not make explicit assumptions about fairness of execution between program and fault actions. This fairness will depend upon the application at hand and can be modeled orthogonally. For example, in some applications, it is assumed that faults will eventually stop. In some cases, probabilities are assigned to execution of each action. Alternatively, in some cases, one may specify parameters such as priority of different actions. Depending upon the nature of such assumptions and the limitations of the tools being used, these fairness requirements can be modeled in an orthogonal manner. Likewise, in some systems (especially hardware-based), it is expected that all actions execute at a particular speed. One instance of this is maximum-parallelism semantics where in every step, all actions whose guard is true are executed. Again, modeling of such semantics is orthogonal to the issue of fault modeling considered in this paper.

4 Uniform Modeling of Faults

In this section, we discuss modeling of faults from different categories in Figure 1. Based on the characteristics of the faults, we partition our discussion among Sections 4.1-4.5. For each section, we identify the abstract model of fault from one category first. Then we introduce how to mapping to the abstract model with concrete example. We also discuss variations of the same fault that are considered in the literature and evaluate its effect on modeling that fault. Subsequently, we identify related fault categories, i.e., fault categories where the modeling would be similar. Finally, we identify the effect of such modeling in verification and revisions.

4.1 Operational, External, Human-made, Persistent, Malicious and Software Faults

In this section, we focus on faults that are operational, external, human made, persistent, malicious and software. (This corresponds to category 24 in Figure 1.) Thus, these faults occur during the system execution. Generally, they are caused by malicious users or attackers. And they persist permanently (or long enough time). It is expected that the system will continue to function even when the faults are present. While some attempts may be made to prevent such faults from occurring, it is possible that such preventions would not be fully successful. Hence, for assurance in the context of these faults, it is necessary that one considers

the system execution where the faults (exhibited in terms of compromised hosts, malicious users or attacks) continue to occur potentially frequently. Often, in such systems, assumptions are made about the number of faults that may exist at a given time. These assumptions ensure that sufficiently many 'good resources' are available to solve the problem at hand.

Abstract Model. To capture the impact that this type of fault has on the underlying system state, there are three types of abstract actions in the model for fault from this category.

– Access Actions. These actions allow user to get knowledge about the current system state. An example of access action is that an user get the data from remote server and make a local copy. We use variable v to denote user's copy of system state and variable s to denote the value of the current system state. Hence this type of actions can be modeled as follows:

ACCESS action:
$v := s$;

– Update Actions. These actions create a change to the system state. Examples of update actions include writing data to file in the server and updating value of a flag which is used to denote whether a file is changed. We use variable v to denote the system state and x to denote the value which is used to update system state by $UpdateActions$. Hence this type of actions can be modeled as follows:

Update action:
$v := x$;

– Fault Actions. These actions, (which may be conducted by malicious users or attackers), perturb the system to a random state. An example of a fault action is that a malicious user changes the value of accessible data randomly. We introduce variable v to denote the state which is corrupted by fault actions. We also introduce an extra variable m to model whether malicious user exists in the environment. The fault actions can be modeled as follows:

Fault action 1:
$true \longrightarrow m := true$;
Fault action 2:
$m \quad \longrightarrow v := random()$;

Mapping to Abstract Model - From a concrete example. Next we discuss how to mapping raw actions in the real example to the abstract actions in the model described above.

A typical example of the faults from this category occurs in the context of distributed system where the system is organized as a network of nodes and some of these nodes may not work as expected and behave arbitrarily because of some reasons, such as attackers. One such example is Farsite [26]. Farsite is a serverless distributed file system that provides centralized file-system service. Farsite aims at providing secure, scalable and strongly consistent file storage service. In Farsite, the concept of servers in the system is virtual and the system actually runs on a network of untrusted PCs some of which may be controlled by a malicious user. Hence, the whole infrastructure is highly susceptible to a fault that the virtual server behaves arbitrarily. This fact brings the challenge of guaranteeing the delivery of correct service in the presence of such faults.

Since our goal is to illustrate the modeling of such fault and not the details of Farsite, without loss of generality, we describe how one can model operations for a single file, say fl. In particular, labeling $UpdateAction$ and $AccessAction$ is straightforward. In this example, for these Read operations that allows users to get the data from servers, we mapping them as $AccessAction$ whereas we mapping Write operation that allows users to write data to servers to $UpdateAction$. In each operation, the client identifies the list of server nodes that contain (or are likely to contain) a copy of the file. We use index j to quantify over the list of such servers. Furthermore, we use the term $data.j$ to denote data maintained by the server j for file fl and the term $copy.j$ to denote local copy of file fl.

In read operation, the client obtains a copy of the file from all (or a subset of) servers. In write operation, the data is written to the respective servers. In order to capture system semantics of fault tolerance mechanism designed in this example, we make extend of $AccessAction$ in the basic abstract model. There are several copies of stored files. Since some of the copies may be incorrect due to malicious users, the client performs an agreement algorithm (e.g., majority computation) first, then return the agreed data to the user who requests file fl.

Thus, the operations in Farsite can be modeled as shown next.

Acess action:
$\forall j :: copy.j := data.j$;
$getResouce := agreement(copy.j)$;
return getResource;
Update action to write v to file:
$\forall j :: data.j := v$;

Obviously our above model in transition system can capture system semantic of this example.

Observe that the above model only describes the program actions. Next, we show that the actions of malicious users can also be modeled in terms of $faultaction$ in the abstract model. In particular, to denote whether process j is malicious, we introduce a variable $m.j$ that denotes whether j is malicious. The fault actions can be modeled as follows:

Fault action 1:
$true \longrightarrow m.j := true$;
Fault action 2:
$m.j \longrightarrow data.j := random()$;

Thus, Fault action 1 denotes that machine j is compromised. Fault action 2 denotes that the compromised machine can change the local data arbitrarily. Observe that along with the write action that assumes that data is written to all replicas, the fault action allows the malicious user to change the data stored in a compromised machine.

Completeness of this approach for modeling faults from this category. Next, we argue that this approach is generic for faults from this category. Specifically, the faults are operational in nature, i.e., at the time system is created (and deployed), there are no faults perturbing the system. (Note that this is not true for development faults.) Thus, faults become operational at some point after (possibly, before the system executes even one of its actions) the system is deployed. Thus, any modeling must include an action by which the malicious components/processes appear in the system. In other words, there must be one action (similar to Fault action 1) where some components/process becomes faulty. Moreover, since the faults are permanent, it is required that it must be possible for the fault to persist forever. In other words, the actions modeling faults should be such that they can execute forever (such as that indicated by the guard of Fault action 2). Finally, because the fault is human made and malicious, the faulty component/process can change the data that it controls arbitrarily (as in the statement of Fault action 2). Thus, any fault that is operational, human made, persistent and malicious, can be modeled using an approach similar to the one presented earlier. Additionally, we note that we have reviewed faults from recent publications in ICDCS and DSN and list these faults that can be modeled using this approach in Table 3.

Modeling variations of this fault. There are several possible variations that one may consider in this category of faults, as follows.

- A malicious user may intend to remain hidden. In this case, Fault action 2 will have to be changed so that instead of changing the data arbitrarily, it will change it only in a way that allows prevents it from being discovered.
- The system may have a mechanism to clean up affected nodes (e.g., through antivirus programs etc.) If such a mechanism exists then it can be modeled by the dual of Fault action 1 where $m.j$ is changed from $true$ to $false$.

– Also, often assumptions are made about the number of malicious users that can exist in the system at a given time. For example, a standard assumption is that the number of malicious replicas is less than a third of the total replicas. If such an assumption is desired, it can be achieved by changing Fault action 1 so that a node can become malicious only if the total number of malicious nodes will not exceed the bound. Note that this will require one to read the value m variable of all nodes. However, this is acceptable since we are simply recording the assumption.

Modeling faults from related categories. Approach similar to the one in this section also applies for faults from other categories. For example, the above modeling can be applied when the fault is caused by hardware. Also, if one were to model the fault which persists for some duration and then disappears, we only need to slightly modify the aboved modeling (the case where fault is permanent (long-lasting) in nature), by adding the modeling of another fault action (similar to Fault action 1) where $m.j$ changes from $true$ to $false$. Based on this discussion, we can model faults from category 22-25 (cf. Figure 1) using the approach in this section. A special case when the fault is transient and occurs only once, we can simplify the modeling of faults by the approach in Section 4.3.

Additionally, a similar approach could also be used for modeling malicious and deliberate development faults (category 5 and 6 in Figure 1). Examples of faults from these categories include logic bomb. For faults from these categories, the modeling in this section may be used if there are several designer teams producing different parts of the system (e.g., with N-version programming) and sufficient redundancy exists to deal with such faults. However, for the case where one design team is creating the given system or where sufficient redundancy is unavailable, the only suitable approach is to use fault prevention where the goal is to ensure that the fault cannot happen. And, when fault prevention is used, there is no need to model faults explicitly, as proving that fault prevention work requires one to only consider system behavior in the absence of faults.

Effect during verification of fault-tolerance. If one were to verify a program that models malicious users in the above fashion, we can observe that the introduction of the variable $m.j$ increases the state space of the program. Additionally, to ensure that the variable $m.j$ is not used improperly, we need to syntactically check the program; specifically, we need to ensure that actions at machine j do not reference variable $m.k$, where $j \neq k$.

Effect during revision for adding fault-tolerance. One important characteristic of above formulation is that variable $m.j$ is readable only to node j. Other nodes cannot read it. If model revision is used for a fault from this category, it is necessary that this restriction is continued to be satisfied in the revised model. Additionally, variable $m.j$ cannot be changed by any node; it may be changed only by a fault action. These restrictions have been shown to increase complexity from P to NP-complete in some instances [22].

4.2 Operational, Internal, Natural, Hardware, Non-malicious, Non-deliberate, Accidental and Persistent Faults

In this section, we focus on faults that are operational, internal, natural, hardware, non-malicious, non-deliberate, accidental and persistent. (This corresponds to category 12 (cf. Figure 1).)

Abstract Model. Obviously, the fault from this category occurs during the system execution. Generally, they occur due to hardware degradation over time. And, it is expected that they last for a significant duration of time. Similar to the faults from Section 4.1, it is expected that the system will continue to function even when the faults are present. Periodic maintenance would be used to replace the hardware as necessary to correct this fault.

Taking these impacts that this category of fault has on the underlying system state into account, we introduce g to denote the hardware state and then the corresponding abstract action in our model is as follows, where ϵ is a small real number.

Fault action 1:
$g \geq \epsilon \longrightarrow g := g - \epsilon;$

Mapping to abstract model - From a concrete example. A typical example from this category occurs in the context of a braking control system where the brake may wear out due to physical deterioration thus system failures may occur with accidents and severe damage and human injuries as consequences.

The basic modeling of braking system includes the speed of the vehicle, the status of brakes (e.g., applied or not) and their reliability.

One such fault in this system can be caused by the case where wear and tear on brakes (or other factors) reduces their effectiveness. Modeling such fault into abstract action is straightforward. If we use g to denote the specified level of performance and integrity of the brake, we can apply abstract action in the basic model directly. To capture system semantics of this example, we assume value of g is in the range $[0, 1]$ where $g = 1$ corresponds the ideal brake and $g = 0$ corresponds to a nonfunctioning brake in this case.

Besides, in order to model program actions, we use variable s to denote the speed of the vehicle. And, we use a Boolean variable b to denote whether the brake is pressed. (Similar to g, b could also be modeled as a continuous variable. However, this extension is straightforward and, hence omitted.) Thus, when the brake is pressed, the vehicle reduces the speed. We model this as a reduction in speed by a fixed value (denoted as C in the below program action) that depends on the reliability of brakes. Hence, one possible way to define such program action in this context is as follows:

Brake Action:
$b == true \longrightarrow s := MAX(s - g * C, 0);$

Completeness of this approach for modeling faults from this category. Next, we argue that this approach is generic for faults from this category. Since the faults are operational in nature, faults occur at some point after the system is deployed and thus any modeling must include some action where some component/process becomes faulty. Also considering the faults are internal, natural, non-malicious and non-deliberate, the fault action should not be triggered by outsider. In other words, the guard of the fault action should be some status of fault component or process itself, like Fault action 1. Moreover, since the faults are permanent, it is required that it must be possible for the fault to persist forever (until the faulty component loses its capability totally).

Thus, any fault that is operational, internal, natural, hardware, nonmalicious, non-deliberate, accidental and persistent Faults, can be modeled using an approach similar to the one presented earlier. Additionally, we review faults from recent relevant publications and list how these faults can be modeled using this approach in Table 3.

Modeling variations of this fault. The above modeling is discrete in nature. It models that when the brake is pressed, speed reduces by a certain amount. A more accurate model would be to utilize timing based information for modeling both programs as well as faults. Examples of such approach include the timed automata [37] that combines transition systems with time. In particular, with such an approach, we could model speed reducing at a particular rate depending upon the reliability of brakes and the pressure applied on brakes.

Modeling faults from related categories. Approach similar to the one in this section also applies for faults from other categories. The above modeling is for the case where fault is permanent in nature. If the faults persist for some duration and then disappears then we need to model another fault action (similar to the above fault action $g \geq \epsilon \longrightarrow g := g - \epsilon$) where g can be increased ϵ and reach at most 1 as range required. Based on this discussion, we can model faults from category 12-13 (cf. Figure 1) using the approach in this section.

However, similar to Section 4.1, for the case where physical elements play a sole role in the structure of the whole system, when that elements totally loss the functions, e.g. a deflated tire of car, there is no need to model the fault explicitly. And, the only suitable approach is to use fault prevention to ensure such faults cannot happen.

Effect during verification and revision of fault-tolerance. It is expected that the fault from this category will generally require one to consider hybrid models where both discrete and continuous variables are considered. The effect/effects of such hybrid model in verification is/are considered in [38]. Regarding revision, revising timed automata is considered in [36]. However, the cost of such revision is often higher (exponential in the constants involved in specifying timing constraints).

4.3 Operational, External, Natural, Hardware, Non-malicious, Non-deliberate, Accidental and Transient Faults

In this section, we focus on faults that are operational, external, natural, hardware, non-malicious, non-deliberate, accidental and transient. (This corresponds to category 15 in Figure 1.)

Abstract Model. Obviously, the fault from this category occurs during the system execution. Generally, they occur due to unexpected transient issues that are unlikely to happen again. They are not malicious in nature but may cause system to behave in an incorrect manner. One approach for dealing with such faults is self-stabilization [43] where the system is guaranteed to recover from an arbitrary state to a legitimate state.

Modeling of the fault from this category in terms of transition systems is straightforward since the fault occurs once (or rarely) and the impact is to change the system state into random value. Specifically, we introduce x to denote system state then the fault can be modeled by the following action:

$$true \longrightarrow x := random();$$

Mapping to Abstract Model- From a concrete example. A typical example from this category includes faults that cause memory to change to an arbitrary state. A possible reason why such errors occur is cosmic rays, for example. Typically, it is assumed that such faults are not detectable and, hence, the system continues to operate in spite of them. Another related example in this class includes the use of uninitialized variables. Such a fault will result in the program starting in an arbitrary state. Moreover, other types of faults such as crash and message loss can exhibit a behavior that is similar to a fault from this category. Specifically, in [42], authors have shown that in the presence of these faults, the program may be (essentially) perturbed to an arbitrary state.

Mapping the raw actions in the real example to the abstract action in the model is straightforward. First, we map the variable (denoted as v) changed by the raw actions into x in the abstract action first. Second, we map the actual way to change the value of v to $random()$ in the abstract action. By these two steps, we can apply the basic abstract model to model fault actions in the real example directly.

Completeness of this approach for modeling faults from this category. Next, we argue that this approach is generic for faults from this category. Consider the faults are operational in nature, there must be some fault action (similar to Fault action 1) occuring after the system is deployed. These faults may affect the entire system or a part of it. Since the faults are transient, this implies that their effect is temporary and the system continues to execute its actions after faults occur. Moreover, since the fault is accidental and not deliberate, the effect of fault on the affected part (like var x in Fault action 1) is random. Thus, any fault that is operational, human made, persistent and malicious, can be modeled using an approach similar to the one presented earlier. Additionally, In Table 3, we identify faults from recent literature in the aspect of fault tolerance that can be modeled using this approach.

Modeling variations of this fault. There are several possible variations that one may consider in this category. For example, the most common assumption used in the literature states that the fault will change the variable to only a value that is legitimate in some configuration. In other words, the fault will assign the variable a value that is from its domain; intuitively, this assumption is based on the fact that if the value assigned to the variable is outside the domain then it would be detected before the variable value is used. Also, often assumptions are made about inability of the fault to change the code itself, i.e., the fault only changes data. If faults are allowed to disrupt code then this can be modeled using the approach in Section 4.1. Alternatively, it could also be modeled using the approach in Section 4.4 if it is expected that the fault will render the corresponding component in a state from where it cannot continue its execution.

Modeling faults from related categories. Approach similar to the one in this section also applies for faults from other categories. For example, the above modeling can also be applied when the fault is caused by human being's non-malicious action. Also, the modeling can be applied where fault is caused by software. Hence, we can model faults from category 15-16, 18-19, 21, 26, 28-29, 31 (cf. Figure 1) using the approach in this section.

Effect during verification of fault-tolerance. If one were to verify a system that models transient faults in the above fashion, then one has to consider system execution from arbitrary (respectively, large number of) states. In particular, if the fault can corrupt all program variables then this corresponds to considering execution from arbitrary states. The topic of such verification is considered in the context of self-

stabilization; in [39], authors have shown the feasibility of verifying self-stabilization using model checker SMV [2].

Effect during revision for adding fault-tolerance. Other than the state space issue, this fault category does not introduce new difficulties in model revision. This is due to the fact that no auxiliary variables are needed to model this fault. An example where such fault modeling is used in model revision includes [40].

4.4 Operational, External, Hardware, Non-malicious and Persistent Faults

In this section, we focus on the class of faults that are operational, external, hardware, non-malicious and persistent. (They correspond to category 14, 17 and 20 (cf. Figure 1).) These categories differ in terms of whether the fault is human-made or natural as well as whether the fault is deliberate or non-deliberate. However, the exact cause is not important in modeling effect of such faults.

Abstract Model. Unlike the faults from Section 4.1, where faults are malicious in nature, the faults in category 14, 17 and 20 are non-malicious. It is expected that these faults are persistent in nature. Examples of such faults include failure of nodes, failure of channels etc.

Modeling of such fault in terms of transition is similar to that in Section 4.1. Specifically, since the faulty component is permanently 'killed' we can model it with an auxiliary variable C that denotes whether the given component is correct or whether it has failed. Also, let $data$ denote state of a node (including its buffered messages). Now, the fault action can be modeled as follows:

Fault action 1:
$C = true \longrightarrow C := false, data := empty;$

Additionally, all program actions would have to be modified so that they only execute when the switch is functioning correctly. In other words, all actions by which one node communicates with another would have to be modified so that it can occur only if the respective nodes have not failed.

Mapping to Abstract Model- From a concrete example. A typical example of the faults from this category occurs in context of networking systems where a fault may cause one or more nodes to fail in a manner where it completely stops working.

One example of such a system is from [33] where authors have focused on developing a failstop processor: A failstop processor works correctly before a fault occurs and it performs no actions when it fails. Moreover, data maintained at the failed processor is lost.

Another example is SafetyNet [27]. SafetyNet is a lightweight global checkpoint/recovery scheme. It aims at providing stable and reliable system services even in the situation of either dropped coherence messages or the loss of an interconnection network switch (and its buffered messages).

Mapping raw actions in the real example to abstract action in the model described above is very straightforward. The steps is similar with Section 4.3(, and hence omitted).

Completeness of this approach for modeling faults from this category. Next, we argue that this approach is generic for faults from this category. Specifically, considering the faults are operational, it is required to include in the modeling such an action to denote faulty components/processes appear in the system. Also, since faults of this category are persistent, it is required that it must be possible for the fault to persist forever (such as that indicated by Fault action 1). Since the fault is external and non-malicious, the occurrence of fault is due to some change of environment condition and hence there must be some variable to denote this change in the modeling (like var c in the Fault action 1). Thus, any fault that is operational, external, persistent, hardware and non-malicious, can be modeled using an approach similar to the one presented earlier. We review faults from recent literature in the area of fault tolerance, and find that how these faults identified in Table 3 can be modeled using this approach.

Modeling variations of this fault. There are several possible variations that one may consider in this category of faults. For example, a 2D torus topology is considered in [27] to prevent a single point-of-failure by splitting each switch into two half-switches. Execution may resume after reconfiguration to route around the lost switch [28] although at reduced bandwidth. In this case, we can use two variables to denote the link status and buffered data of each half-switch independently. The link status of the whole switch can be modeled as disjunction of the link status of each half element. Also often redundancy is used in the networking system. For example, the configuration of a specific point-to-point path may consist of several

available links. If such an assumption is desired, one can model the fault action for each available switch first. Then, the link status of the whole configuration can be modeled as disjunction value of these available switches.

The fault modeling from this section assumes that the fault is permanent, i.e., the failed node remains failed forever. A variation of this model is one where the faulty node is repaired and integrated in the system. In this case, a dual of the fault action where C is set to true must be added. Depending upon how such an action restores the node, the data associated with the node would change. For example, if the restore is equivalent to reboot where the node starts from some fixed state, data will be changed accordingly.

Modeling faults from related categories. Approach similar to the one in this section also applies for faults from other categories. For example, the above modeling can be used whether faults affect hardware or software. Hence, we can model faults from category 27 and 30 (cf. Figure 1) using the approach in this section.

Effect during verification and revision of fault-tolerance. The effect on verification and revision is similar to that in Section 4.1. There are some possible changes depending upon the assumptions about faults. In particular, variable C introduced in fault actions may or may not be readable by other processes depending upon whether one assumes that a fault is detectable or not. Furthermore, approaches such as failure detectors [41] could be used to model more fine tuned assumptions about delectability of the fault.

4.5 Development Faults

In this section, we focus on faults that occur during development. This corresponds to category 1-11 in the Figure 1. Typical examples of such faults cause software flaws, logic bombs and hardware errata, etc. If the developer(s) or operator(s) did not realize at the time that the consequence of their decision was a fault (or conceal faulty actions like logic bombs with the malicious purpose), and furthermore, design or decision is accepted for use, these faults can be treated as intrinsic nature of system and the occurrence of the faults is unavoidable.

A concrete example. A typical example from this class includes Pentium FDIV bug in the Intel P5 Pentium floating point unit(FPU) that was caused by few missing entries in the lookup table used by the divide operation. Another example is that of buffer overflows where one copies a longer string into a shorter string thereby affecting other parts of memory.

We argue that for such faults, formal methods for modeling faults explicitly are either undesirable or impossible. For such faults, a more practical approach is fault prevention where the goal is to ensure that the fault does not occur. For example, a thorough analysis of code could be useful to ensure that logic bombs do not occur. Likewise, analysis of Pentium FPU with theorem provers [34] has been successful in identifying the missing table entries in Pentium. Likewise, approaches such as [35] can be used to prevent buffer overflows.

In other words, for faults from this category, one needs to consider *fault-free execution* to show that the fault does not occur. Since this requires consideration of only *fault-free* execution, it does not require one to model faults explicitly.

That said, in certain instances, one may consider these faults explicitly and tolerate them, as preventing them may be impossible. In such cases, one needs to consider the *effect* of the development faults. We expect most such development faults to exhibit themselves as malicious faults (cf. Section 4.1) or as failstop (cf. Section 4.4) faults. In particular, we have discussed modeling of faults from category 5 and 6 in Section 4.1.

5 Practicability of Modeling during Verification and Revision

In Section 4, we considered the *feasibility* of modeling faults in terms of transition systems. In this section, we utilize those results in identifying the *practicability* of such modeling. Specifically, our goal is to evaluate the *cost* of such modeling in two contexts: *model-checking* and *model revision*. In case of model checking, we want to compare the cost of verifying a *fault-intolerant* program with that of verifying *fault-tolerant* program. And, in case of model revision, we want to compare the cost of *verifying fault-intolerant* program with that of *revising it to add fault-tolerance*. Both these tasks are feasible only if we can model the faults during the verification and/or revision process.

5.1 Cost of Modeling Faults during Model Checking

Model checking focuses on deciding whether a given model of system, say M satisfies the given property pr. While the cost of model checking depends upon several factors, one important factor is the state space of the resulting model. Since modeling of faults in terms of transitions has the potential to increase the state space of the program, we evaluate the cost in terms of the increased state space. Specifically, we consider the increased cost of modeling faults from Sections 4.1-4.4.

Observe that for modeling persistent and malicious faults, in Section 4.1, we needed to add a variable $m.j$ for every user. Essentially, this would double the state space of that user. Moreover, if there are n users in the system, then the total state space will be 2^n times more than that of the fault-intolerant system.

For faults from Section 4.2, the cost depends upon whether the physical degradation can be modeled using discrete values or whether continuous modeling is required. If the physical degradation is modeled using discrete values as in Section 4.2, the total state space will increase by a constant factor when compared with that of the fault-intolerant system. If the physical is modeled using continuous values, especially modeling in hybrid automata [38], the total reachability problem is potentially undecidable.

If one were to verify a system that models transient fault as in Section 4.3, then one has to consider all possible states that could be subtantially larger. However, the total state space will be unchanged compared with that of fault-intolerant system.

Since the modeling approach of faults from Section 4.4 is similar to that in Section 4.1, the increased state space on verification of modeling faults from Section 4.4 is similar to that of Section 4.1.

Based on the above discussion, we summarize the increased cost of modeling faults from Sections 4.1-4.4 in Table 1.

	Increased Cost of Modeling Faults during Model Checking
Faults from Section 4.1	Increased by a factor of 2^n.
Faults from Section 4.2	Increased by a constant factor if discrete degradation is considered. Potentially undecidable if continuous degradation is considered.
Faults from Section 4.3	Unchanged in statespace.
Faults from Section 4.4	Increased by a factor of 2^n.

Table 1: Cost of Modeling Faults during Model Checking *in terms of the Increased State Space*

According to the results in Table 1, we argue that the increased cost due to modeling faults in 18 categories (identifed in Section 4.1, 4.3 and 4.4) is reasonable. However, the increased cost due to modeling faults in 2 categories (identified in Section 4.2) is high.

5.2 Cost of Modeling Faults during Model Revision

Since model checking computes (directly or indirectly) all computations of M to determine whether pr is satisfied, it is especially useful in providing assurance about a system developed from that model. The related problem of model revision [3, 4] focuses on scenarios where model checking produces a counterexample or where an existing model needs to be revised to add new properties (such as safety, liveness and timing constraints). Thus, the goal in model revision is to modify the given model M so that it satisfies the given property pr. Since the revised model is correct by construction, it can assist us in obtaining a correct model of system when model checking ends up finding a counterexample.

Considering read-write restriction must be continued to be satisfied in the revised model, for modeling persistent and malicious faults in Section 4.1, variable $m.j$ is readable only to node j. Other nodes cannot read it. Besides, variable $m.j$ cannot be changed by any node; it may be changed only by a fault action. These restrictions have been shown to increase complexity from P to NP-complete in some instances [22].

Regarding revision for the modeling of faults from Section 4.2, revising timed automata is considered in [36]. However, the cost of such revision is often higher (exponential in the constants involved in specifying timing constraints. In some circumstance, the problem is even undecidable.

For modeling transient faults in Section 4.3, it does not introduce new difficulties in model revision. This is due to the fact that no auxiliary variables are needed to model this fault. An example where such fault modeling is used in model revision includes [40].

Similarly to modeling of faults from Section 4.1, the complexity of modeling faults from Section 4.4 is increased from P to NP-complte in certain circumstance.

As discussed above, we summarize the complexity issues caused by modeling faults from Sections 4.1-4.4 during model revision in Table 2.

	Complexity of Modeling Faults during Model Revision
Faults from Section 4.1	From P to NP-complete in size of statespace.
Faults from Section 4.2	Undecidable in certain circumstance.
Faults from Section 4.3	For centralized system, unchanged in complexity class. For distributed system, conjectured to NP-complete.
Faults from Section 4.4	From P to NP-complete in size of statespace.

Table 2: Complexity of Modeling Faults during Model Revision

Based on these results in Table 2, the complexity increases substantially for model revision. However, efficient heuristics have been found to mitigate the complexity for modeling faults identified in Section 4.1, 4.3 and 4.4. Hence, we argue that model revision for faults from these categories is practical. And for faults discussed in Section 4.2, the increased complexity may be too high.

6 Related work

Formally modeling of faults is studied from different perspectives. One approach focuses on representing the faults in a formal expression (e.g. process algebra, higher-order logic, atomic actions, etc) and thus facilitates the development of the fault-tolerant system. More specifically, modeling faults by process algebra represents fault as process as well as the system processes, and hence the system behavior in the presence of faults with this modeling approach can be a composition of processes. Subsequently, designer can utilize the existing verification techniques (e.g. model checking) to guarantee the correctness of design. The typical examples utilizing process algebra include [13, 19–21]. In [23], fault is modeled in a typed-system in higher-order logic and such a modeling approach can utilize the PVS system prover to specify and verify the design and implementation of the system. In [24, 25], authors model the fault as a set of atomic actions and design recovery actions from faults. Then, they show that the union of the program actions and the recovery actions meets system requirements in the presence of faults. However, these approaches seem tedious when one needs to differentiate and model the diversity of fault classes. By contrast, our approach will advance the applicability of the uniform modeling approach by allowing designers to model different classes of faults. Hence our approach is desirable when one needs to designing different levels of fault-tolerance to different classes of faults.

Our work is orthogonal to several existing approaches (e.g. [44–47]) for *identifying faults* in early stages of design and anticipating all possible faults that could occur in the system. The results in this paper are intended towards modeling faults identified using techniques such as those in [44–47] so that one can utilize techniques from model checking and revision.

Some formal specifications (e.g., in [14–18]) have addressed the problem of modeling only one specific class of fault. However, there are many situations where a system may be subject to multiple classes of faults. Our approach pursues to achieve uniform modeling of faults from different classes. The uniform formal modeling in our approach is necessary especially when we need to design or test the system where multiple classes of faults may occur.

7 Conclusion

This paper focused on bridging the gap between theory and practice of fault-tolerant systems by identifying a uniform model for different categories of faults. Towards this end, we began with the classification that

is based on practitioner's point of view [7] and for each category, either (1) identified how that fault can be modeled using transition systems or (2) argued that fault prevention techniques should be used for the corresponding fault thereby obviating the need for modeling that fault explicitly.

We show that (1) for 18 categories of the 31 catetories from [7], modeling faults using transition systems is practical and feasible; (2) for 2 categories, modeling faults using transition systems is feasible but not practical and (3) for 11 categories, modeling faults using transition system is not feasible.

Our approach for modeling faults is analogous with the approach for modeling fault-free behavior in model checking and the latter has been shown to be one of the most successful strategies for analyzing fault-free models. Also, our approach has been used in verifying and revising fault-tolerant programs in the context of specific instances of faults [4, 5, 25]. Hence, we expect the results in this paper to bridge the gap between theory and practice in providing assurance about fault-tolerant system design. Moreover, as the examples in Section 4 illustrate, our uniform fault modeling approach is beneficial in the situation where the system is subject to multiple faults from different classes (e.g., node crash vs message loss vs malicious attacker). Particularly, our uniform modeling of faults will assist in analyzing the system that is subject to multiple faults and furthermore utilizing the existing technologies (e.g., [8, 9]) to analyze and provide tolerance to such situations.

References

1. Holzmann, G.J.: The model checker SPIN, IEEE Trans. Software Eng., vol. 23, no.5, pp.279-295, May 1997.
2. McMillan, K. L.: Symbolic Model Checking. Kluwer Academic, 1993.
3. Borzoo Bonakdarpour and Sandeep S. Kulkarni: Exploiting Symbolic Techniques in Automated Synthesis of Distributed Programs. In IEEE International Conference on Distributed Computing Systems(ICDCS), pp. 3-10, 2007.
4. Borzoo Bonakdarpour and Sandeep S. Kulkarni, SYCRAFT: A Tool for Automated Synthesis of Fault-Tolerant Distributed Programs. In International Conference on Concurrency Theory (CONCUR), LNCS 5201, pp. 167-171,2008.
5. Felix C. Gartner: Specifications for Fault Tolerance: A Comedy of Failures, Technical Report TUD-BS-1998-03. Darmstadt University of Technology, Darmstadt, Germany, 1998.
6. M. Demirbas, A. Arora, V. Mittal and V. Kulathumani. A fault-local self-stabilizing clustering service for wireless ad hoc networks . IEEE Transactions on Parallel and Distributed Systems, Special issue on Localized Communication and Topology Protocols for Ad Hoc Networks, 17(4), 2006.
7. Avizienis, Algirdas and Laprie, Jean-Claude and Randell, Brian and Landwehr, Carl: Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Trans. Dependable Secur. Comput., 2004.
8. S. S. Kulkarni and A. Ebnenasir: Automated Synthesis of Multitolerance, International Conference on Dependable Systems and Networks, Palazzo dei Congressi, Florence, Italy, June 2004
9. Jingshu Chen and Sandeep S. Kulkarni: Complexity Analysis of Weak Multitolerance. In IEEE International Conference on Distributed Computing Systems(ICDCS) 2010, Genoa, Italy
10. Sukumar Ghosh, Arobinda Gupta, Ted Herman, Sriram Pemmaraju, Fault-containing Self-stabilizing Distributed Protocols, Distributed Computing, 20:53-73, 2007.
11. Yukiko Yamauchi, Toshimitsu Masuzawa, and Doina Bein: Preserving the Fault-Containment of Ring Protocols Executed on Trees, British Computer Journal, volume 52, number 4, pages 483-498, July 2009.
12. S. Kutten and D. Peleg: Fault-Local Mending, Journal of Algorithms, Vol. 30, No. 1, January 1999: 144-165. Also appeared in Proceedings of the Fourteenth Annual ACM Symposium on Principle of Distributed Computing (PODC 95) , Ottawa, Canada, August 1995: 20-27.
13. K. V. S. Prasad, Specification and proof of a simple fault tolerant system in CCS, Dep. Comput. Sci., Univ. Edinburgh, Int. Rep. CSR-178-84, 1984
14. Roxana Geambasu, Andrew Birrell, and John MacCormick: Experiences with Formal Specification of Fault-tolerant File Systems, In: Proceedings of the 38th Annual International Conference on Dependable Systems and Networks, 2008
15. J.F. Martins, P. J. Costa Branco, A.J. Pires, J.A. Dente: Fault Detection using Immune-Based Systems and Formal Language Algorithms,in IEEE Conference on Decision and Control (CDC2000).
16. Mahlstedt, U., Heinitz, M., Alt, J., Test Generation for IDDQ Testing and Leakage Fault Detection in CMOS Circuits, EURODAC 92, pp.486-491
17. Eldred, R.D., Test Routines Based on Symbolic Logical Statements, Journal ACM, vol.6, no.1, 1959, pp.33-36.
18. Su, Chauchin and Chiang, Shenshung and Jou, Shyh-Jye: Impulse response fault model and fault extraction for functional level analog circuit diagnosis,ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design, 1995.

19. Peleska, J. 1991. Design and verification of fault tolerant systems with csp. Distributed Computing 5, 95C106
20. Bernardeschi, C., Fantechi, A., and Simoncini, L. 2000. Formally verifying fault tolerant system designs. The computer journal 43, 3, 191C205
21. Gnesi, S., Lenzini, G., and Martinelli, F. 2005. Logical specification and analysis of fault tolerant systems through partial model checking. Electronic Notes in Theoretical Computer Science 118, 57-70
22. S. S. Kulkarni and A. Arora. Automating the Addition of Fault-Tolerance Formal Techniques in Real-Time and Fault Tolerant Systems, 2000.
23. Pike, L., Maddalon, J., Miner, P. S., and Geser, A. 2004. Abstractions for fault-tolerant distributed system verification. In 17th International Conference Theorem Proving in Higher Order Logics (TPHOLs). 257C27
24. Lin, F. and Wonham, W. M. 1990. Decentralized control and coordination of discrete-event systems with partial observation. IEEE Transactions On Automatic Control 35, 12 (December),1330-1337
25. Liu, Z. and Joseph, M. 1992. Transformation of programs for fault-tolerance. Formal Aspects of Computing 4, 5, 442C469
26. William J. Bolosky, John R. Douceur, and Jon Howell, The Farsite project: a retrospective, in ACM SIGOPS Operating Systems Review 41 (2), Association for Computing Machinery, Inc., April 2007.
27. Sorin, Daniel J. and Martin, Milo M. K. and Hill, Mark D. and Wood, David A.: SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery, ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture, 2002.
28. J. Duato, S. Yalamanchili, and L. Ni. Interconnection Network. IEEE Computer Society Press, 1997.
29. USA Department of Transportation, Office of Inspector General, Audit Report: Advance Automation System, Report Av-1998-113, Apr. 1998.
30. J. Gray: Functionality, Availability, Agility, Manageability, Scalability – The new priorities of application design, Proc. Int'l Workshop High Performance Trans. Systems, 2001.
31. Johan Bengtsson and Fredrik Larsson.: Uppaal a Tool for Automatic Verification of Real-Time Systems. DoCS Technical Report Nr 96/67, Uppsala University, ISSN 0283-0574, January 1996.
32. Hinton, A; Kwiatkowska, M; Norman, G; Parker, D. PRISM: A Tool for Automatic Verification of Probabilistic Systems. Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06), volume 3920 of Lecture Notes in Computer Science, pages 441-444, Springer. 2006.
33. Schlichting, Richard D. and Schneider, Fred B.: Fail-stop processors: an approach to designing fault-tolerant computing systems, ACM Trans. Comput. Syst.,1983.
34. Ruess, Harald and Shankar, Natarajan and Srivas, Mandayam K.: Modular Verification of SRT Division, Form. Methods Syst. Des., v.14, 1999.
35. Tuck, Nathan and Calder, Brad and Varghese, George, Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow, MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, 2004.
36. Borzoo Bonakdarpour: Automated Revision of Distributed and Real-Time Programs, PH.D. dissertation, Michigan State University, 2008.
37. R. Alur and D.L. Dill: A theory of timed automata. Theoretical Computer Science 126:183-235, 1994 (preliminary versions appeared in Proc. 17th ICALP, LNCS 443, 1990, and Real Time: Theory in Practice, LNCS 600, 1991).
38. R. Alur, C. Courcoubetis, T.A. Henzinger, P.-H. Ho. Hybrid Automata: An algorithmic approach to the specification and verification of hybrid systems. In Hybrid Systems, LNCS 736, pp. 209-229, 1993.
39. Tsuchiya, Tatsuhiro and Nagano, Shin'ichi and Paidi, Rohayu Bt and Kikuno, Tohru: Symbolic Model Checking for Self-Stabilizing Algorithms. IEEE Trans. Parallel Distrib. Syst. 12, 81–95 (2001).
40. Fuad Abujarad, and Sandeep S. Kulkarni. Constraint Based Automated Synthesis of Nonmasking and Stabilizing Fault-Tolerance. 28th International Symposium on Reliable Distributed Systems (SRDS), pp. 19-128, September-2009.
41. Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, 1996.
42. Jayaram, Mahesh and Varghese, George, Crash failures can drive protocols to arbitrary states, PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, 1996.
43. Dijkstra, Edsger W.: Self-stabilizing systems in spite of distributed control, Commun. ACM, 1974.
44. ARP-4761: Aerospace recommended practice: guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. 12th edition, Society of Automotive Engineers (SAE), 400 Commonwealth Drive Warrendale PA United States. 1996.
45. Vesely, W. Fault tree handbook. US Nuclear Regulatory Committee Report NUREG-0492, US NRC, Washington DC, United States. 1981.
46. Palady, P. Failure modes and effects analysis. PT Publications Inc. 1995.
47. Kletz, T. A. HAZOP and HAZAN: Identifying and assessing process industry standards. CRC (4 edition). 1999.
48. Anish Arora , Mohamed Gouda, Closure and Convergence: A Foundation of Fault-Tolerant Computing, IEEE Transactions on Software Engineering, v.19 n.11, 1992.

A Relative Completeness of this approach with Recent Literature

In this section, we evaluate relative completeness of modeling approach proposed in our paper with recent literature. In particular, we study the papers from two premier conferences in the area of fault tolerance: the International Conference On Distributed Computing Systems (ICDCS) and the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) since 2007.

Of 139 fault-tolerant relevant papers, we evaluate each paper whether modeling of those faults is feasible using the approaches mentioned in our paper. We find that faults mentioned from 51 papers can be modeled in the approach proposed in Section 4.1. Faults mentioned from 13 papers can be modeled in the approach proposed in Section 4.2. The fault model in the Section 4.3 can be applied to modeling faults from 40 papers. And the fault model in the Section 4.4 can be used in modeling faults from 31 papers. Besides, there are faults from 4 papers that can't be modeled in the approach proposed in our paper. We summarize our evaluation that how these faults can be modeled in the our proposed approaches in Table 3.

Appropriate Model	Faults proposed in the publications from DSN and ICDCS *since 2007*
Approach in Section 4.1	DDoS Attacks in [2], [23] Selfishness of Node/Host in distributed system in [4], [16], [17], [18], [21], [28], [50], [55], [56], [60], [70], [71], [74], [123], [135], [136], [137], [138], [139] Attacks in [11], [19], [20], [24], [25], [26], [58], [59], [61], [66], [67], [68], [76], [77], [96], [99], [100], [104], [105], [109], [110], [112], [117], [124], [127], [133], [134] Worms in [12], [111], [114],
Approach in Section 4.2	Power degradation of Battery in MANETs mentioned in [5], Energy consumption in [13], [35], [39], [40], [41], [42], [45], [52], [57], [62], [65], Aging-related bug in [129],
Approach in Section 4.3	Data Incoherency in [3], [8], [14], [27], [30], [32], [34], [37], [38], [43], [44], [46], [49], [72], [87], [116], [130], [131] Noise in WSN [10], User's misbehavior in [22], [67], [94], [106] Uncertain data in [53], [93], [103], [108], [122] Transient bugs in [64], [73], [75], [83], [84], [85], [89], [91], [107], [115], [120], [121], [128], [132]
Approach in Section 4.4	Message Loss in [1], [15], [80], [102] Crash Failure in [6], [7], [36], [47], [48], [51], [54], [63], [69], [78], [79], [81], [82], [86], [90], [92], [95], [113], [125], [126], [128] Disconnection in distributed system in [8], [29], [31], [97] [33], [98], [101].
No approach is feasible	code injection in [88], OS bugs in [118], Security flaw of web service in [119], program bugs in [140]

Table 3: Classification of Faults proposed in the publication of DSN and ICDCS *since 2007*

References

1. Yu Wang, Hongyi Wu, Feng Li and Nian-Feng Tzeng: Protocol Design and Optimization for Delay/Fault-Tolerant Mobile Sensor, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.

2. Minos Garofalakis, Rajeev Rastogi and Krishan Sabnani: Streaming Algorithms for Robust, Real-time Detection of DDoS Attacks, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
3. Manish Bhide, Krithi Ramamritham and Mukund Agrawal: Efficient Execution of Continuous Incoherency Bounded Queries over Multi-Source Streaming Data, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
4. Lin Chen and Jean Leneutre: Selfishness, Not Always A Nightmare: Modeling Selfish MAC Behaviors in Wireless Mobile Ad Hoc Networks,in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
5. Shan-Hung Wu, Chung-Min Chen and Ming-Syan Chen: An Asymmetric Quorum-base Power Saving Protocol for Clustered Ad Hoc Networks, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
6. Rachid Guerraoui, Dejan Kostic, Ron Levy and Vivien Quema: A High Throughput Atomic Storage Algorithm, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
7. Vijay K. Garg and Vinit Ogale: Fusible Data Structures for Fault-Tolerance,in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
8. Philippe Bergheaud, Dinesh Subhraveti and Marc Vertes: Fault Tolerance in Multiprocessor Systems via Application Cloning,in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
9. Sylvia Bianchi, Ajoy Datta, Pascal Felber and Maria Gradinariu: Stabilizing Peer-to-Peer Spatial Filters, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
10. Yongzhen Zhuang, Lei Chen, Xiaoyang, X. Sean Wang and Jie Lian: A Weighted Moving Average-Based Approach for Cleaning Sensor Data, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
11. Donggang Liu: Resilient Cluster Formation for Sensor Networks, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
12. Guanhua Yan and Stephan Eidenbenz: Modeling Propagation Dynamics of Bluetooth Worms, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
13. Cigdem Sengul and Robin Kravets: Heuristic Approaches to Energy-Efficient Network Design Problem, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
14. Yunfeng Lin, Baochun Li and Ben Liang: Differentiated Data Persistence with Priority Random Linear Codes, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
15. Mahmoud Elhaddad, Hammad Iqbal, Taieb Znati and Rami Melhem: Scheduling to minimize the worst-case loss rate, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
16. Ramses Morales and Indranil Gupta: AVMON: Optimal and Scalable Discovery of Consistent Availability Monitoring Overlays for Distributed Systems, in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
17. An Empirical Study of Collusion Behavior in the Maze P2P File-Sharing System, Qiao Lian, Zheng Zhang, Mao Yang, Ben Y. Zhao, Yafei Dai and Xiaoming Li: in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
18. Robust and Secure Interactions in Open Distributed Systems through Recovery of Trust Negotiations, Anna Cinzia Squicciarini, Alberto Trombetta and Elisa Bertino: in Proceeding ICDCS '07 Proceedings of the 27th International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2007.
19. Kai Xing, Fang Liu, Xiuzhen Cheng, David Hung-Chang Du: Real-Time Detection of Clone Attacks in Wireless Sensor Networks,28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
20. Wenjun Gu, Zhimin Yang, Can Que, Dong Xuan, Weijia Jia: On Security Vulnerabilities of Null Data Frames in IEEE 802.11 Based WLANs,28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
21. Radu Sion: Strong WORM, 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.

22. Linfeng Zhang, Yong Guan:Detecting Click Fraud in Pay-Per-Click Streams of Online Advertising Networks, 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
23. Muthusrinivasan Muthuprasanna, Govindarasu Manimaran: Distributed Divide-and-Conquer Techniques for Effective DDoS Attack Defenses, 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
24. P. Kumar Manna, Sanjay Ranka, Shigang Chen: Analysis of Maximum Executable Length for Detecting Text-Based Malware. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
25. Jose Carlos Brustoloni, David Kyle: Updates and Asynchronous Communication in Trusted Computing Systems. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
26. My T. Thai, Ying Xuan, Incheol Shin, Taieb Znati: On Detection of Malicious Users Using Group Testing Techniques. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
27. Xuejun Yang, Panfeng Wang, Hongyi Fu, Yunfei Du, Zhiyuan Wang, Jia Jia: Compiler-Assisted Application-Level Checkpointing for MPI Programs. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
28. Sajeeva L. Pallemulle, Haraldur D. Thorvaldsson, Kenneth J. Goldman: Byzantine Fault-Tolerant Web Services for n-Tier and Service Oriented Architectures. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
29. Michio Honda, Jin Nakazawa, Yoshifumi Nishida, Masahiro Kozuka, Hideyuki Tokuda: A Connectivity-Driven Retransmission Scheme Based On Transport Layer Readdressing. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
30. Qing Ye, Liang Cheng: DTP: Double-Pairwise Time Protocol for Disruption Tolerant Networks. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
31. Zhao Zhang, Weili Wu, Shashi Shekhar: Optimal Placements in Ring Network for Data Replicas in Distributed Database with MajorityVoting Protocol. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
32. Reza Curtmola, Osama Khan, Randal C. Burns, Giuseppe Ateniese: MR-PDP: Multiple-Replica Provable Data Possession. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
33. Guang Tan, Stephen A. Jarvis, Anne-Marie Kermarrec: Connectivity-Guaranteed and Obstacle-Adaptive Deployment Schemes for Mobile Sensor Networks. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
34. Yun Wang, Jie Wu: A Nonblocking Approach for Reaching an Agreement on Request Total Orders. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
35. Yingshu Li, Chunyu Ai, Wiwek P. Deshmukh, Yiwei Wu: Data Estimation in Sensor Networks Using Physical and Statistical Methodologies. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
36. Weijun Xiao, Qing Yang: Can We Really Recover Data if Storage Subsystem Fails? 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
37. Samer Al-Kiswany, Matei Ripeanu, Sudharshan S. Vazhkudai, Abdullah Gharaibeh: stdchk: A Checkpoint Storage System for Desktop Grid Computing.28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
38. Chang Xu, Shing-Chi Cheung, Wing Kwong Chan, Chunyang Ye: Heuristics-Based Strategies for Resolving Context Inconsistencies in Pervasive Computing Applications. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
39. Shan-Hung Wu, Ming-Syan Chen, Chung-Min Chen: Fully Adaptive Power Saving Protocols for Ad Hoc Networks Using the Hyper Quorum System. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
40. Dong Li, Yanmin Zhu, Li Cui, Lionel M. Ni: Hotness-Aware Sensor Networks. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
41. Cigdem Sengul, Mehedi Bakht, Albert F. Harris III, Tarek F. Abdelzaher, Robin Kravets: Improving Energy Conservation Using Bulk Transmission over High-Power Radios in Sensor Networks. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.

42. Jiayu Gong, Xiliang Zhong, Cheng-Zhong Xu: Energy and Timing Constrained System Reward Maximization on Wireless Networks. 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China. IEEE Computer Society 2008.
43. Maintaining Probabilistic Consistency for Frequently Offline Devices in Mobile Ad Hoc Networks, Wenzhong Li, Edward Chan, Daoxu Chen, Sanglu Lu. Published in ICDCS '09 Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2009.
44. Andrey Brito, Christof Fetzer, Pascal Felber: Minimizing Latency in Fault-Tolerant Distributed Stream Processing Systems. Published in ICDCS '09 Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2009.
45. Min Yeol Lim, Freeman L. Rawson III, Tyler K. Bletsch, Vincent W. Freeh: PADD: Power Aware Domain Distribution. Published in ICDCS '09 Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2009.
46. Weihan Wang, Cristiana Amza: On Optimal Concurrency Control for Optimistic Replication.Published in ICDCS '09 Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2009.
47. Carole Delporte-Gallet, Hugues Fauconnier, Andreas Tielmann: Fault-Tolerant Consensus in Unknown and Anonymous Networks. Published in ICDCS '09 Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2009.
48. Lanyue Lu, Prasenjit Sarkar, Dinesh Subhraveti, Soumitra Sarkar, Mark Seaman, Reshu Jain, Ahmed Bashir: CARP: Handling Silent Data Errors and Site Failures in an Integrated Program and Storage Replication Mechanism. Published in ICDCS '09 Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2009.
49. Stephane Weiss, Pascal Urso, Pascal Molli: Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. Published in ICDCS '09 Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2009.
50. Donggang Liu: Protecting Neighbor Discovery Against Node Compromises in Sensor Networks. Published in ICDCS '09 Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2009.
51. Tao Xie, Abhinav Sharma: Collaboration-Oriented Data Recovery for Mobile Disk Arrays. Published in ICDCS '09 Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems IEEE Computer Society Washington, DC, USA, 2009.
52. Gueyoung Jung, Matti A. Hiltunen, Kaustubh R. Joshi, Richard D. Schlichting, Calton Pu: Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
53. Xiaofeng Ding, Hai Jin: Efficient and Progressive Algorithms for Distributed Skyline Queries over Uncertain Data. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
54. Wenbo He, Xue Liu, Long Zheng, Hao Yang: Reliability Calculus: A Theoretical Framework to Analyze Communication Reliability. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
55. Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, Steve J. Chapin: ESCUDO: A Fine-Grained Protection Model for Web Browsers. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
56. Maxwell Young, Aniket Kate, Ian Goldberg, Martin Karsten: Practical Robust Communication in DHTs Tolerating a Byzantine Adversary. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
57. Yinliang Yue, Lei Tian, Hong Jiang, Fang Wang, Dan Feng, Quan Zhang, Pan Zeng: RoLo: A Rotated Logging Storage Architecture for Enterprise Data Centers. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
58. Jong Chun Park, Jedidiah R. Crandall: Empirical Study of a National-Scale Distributed Intrusion Detection System: Backbone-Level Filtering of HTML Responses in China. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
59. Soo Bum Lee, Virgil D. Gligor: FLoc : Dependable Link Access for Legitimate Traffic in Flooding Attacks. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
60. Marin Bertier, Anne-Marie Kermarrec, Guang Tan: Message-Efficient Byzantine Fault-Tolerant Broadcast in a Multi-hop Wireless Sensor Network. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
61. Qi Dong, Donggang Liu: Adaptive Jamming-Resistant Broadcast Systems with Partial Channel Sharing. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.

62. Bin Tong, Zi Li, Guiling Wang, Wensheng Zhang: How Wireless Power Charging Technology Affects Sensor Network Deployment and Routing. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
63. Taylor Johnson, Sayan Mitra, Karthik Manamcheri: Safe and Stabilizing Distributed Cellular Flows. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
64. Yangfan Zhou, Xinyu Chen, Michael R. Lyu, Jiangchuan Liu: Sentomist: Unveiling Transient Sensor Network Bugs via Symptom Mining. in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
65. Yu Gu, Tian He: Bounding Communication Delay in Energy Harvesting Sensor Networks.in Proceedings of 2010 International Conference on Distributed Computing Systems, Genova, Italy.
66. Yixin Shi, Gyungho Lee: Augmenting Branch Predictor to Secure Program Execution. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
67. Ying Zhang, Zhuoqing Morley Mao, Jia Wang: A Firewall for Routers: Protecting against Routing Misbehavior. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
68. Ryan Riley, Xuxian Jiang, Dongyan Xu: An Architectural Approach to Preventing Code Injection Attacks. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
69. Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, Andrew S. Tanenbaum: Failure Resilience for Device Drivers. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
70. Nicolas Salatge, Jean-Charles Fabre: Fault Tolerance Connectors for Unreliable Web Services. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
71. Piotr Zielinski: Automatic Verification and Discovery of Byzantine Consensus Protocols. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
72. Fabiola Greve, Sebastien Tixeuil: Knowledge Connectivity vs. Synchrony Requirements for Fault-Tolerant Agreement in Unknown Networks. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007
73. Martin Hutle, Andre Schiper: Communication Predicates: A High-Level Abstraction for Coping with Transient and Dynamic Faults. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
74. Josef Widder, Gunther Gridling, Bettina Weiss, Jean-Paul Blanquart: Synchronous Consensus with Mortal Byzantines. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
75. Ahmad Rahmati, Lin Zhong, Matti A. Hiltunen, Rittwik Jana: Reliability Techniques for RFID-Based Object Tracking Applications. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
76. Daniel Ramsbrock, Robin Berthier, Michel Cukier: Profiling Attacker Behavior Following SSH Compromises. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
77. Keith Harrison, Shouhuai Xu: Protecting Cryptographic Keys from Memory Disclosure Attacks. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
78. Marco Serafini, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstatter, Fulvio Tagliabo, Jens Koch: A Tunable Add-On Diagnostic Protocol for Time-Triggered Systems. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
79. Jon G. Elerath, Michael Pecht: Enhanced Reliability Modeling of RAID Storage Systems. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
80. James W. Mickens, Brian D. Noble: Concilium: Collaborative Diagnosis of Broken Overlay Routes. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
81. Shengchao Yu, Yanyong Zhang: R-Sentry: Providing Continuous Sensor Services against Random Node Failures. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.

82. Kenichi Kourai, Shigeru Chiba: A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
83. Teruaki Sakata, Teppei Hirotsu, Hiromichi Yamada, Takeshi Kataoka: A Cost-Effective Dependable Microcontroller Architecture with Instruction-Level Rollback for Soft Error Recovery. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
84. Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, Daniel A. Connors: Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
85. Qijun Zhu, Chun Yuan: A Reinforcement Learning Approach to Automatic Error Recovery. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
86. Tiejun Ma, Jane Hillston, Stuart Anderson: On the Quality of Service of Crash-Recovery Failure Detectors. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
87. Marcos Kawazoe Aguilera, Kimberly Keeton, Arif Merchant, Kiran-Kumar Muniswamy-Reddy, Mustafa Uysal: Improving Recoverability in Multi-tier Storage Systems. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
88. Ryan Riley, Xuxian Jiang, Dongyan Xu: An Architectural Approach to Preventing Code Injection Attacks. in The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings. IEEE Computer Society 2007.
89. Vimal K. Reddy, Eric Rotenberg: Coverage of a microarchitecture-level fault check regimen in a superscalar processor. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008
90. Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, Yuanyuan Zhou: Trace-based microarchitecture-level diagnosis of permanent hardware faults. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
91. Hisashige Ando, Ryuji Kan, Yoshiharu Tosaka, Keiji Takahisa, Kichiji Hatanaka: Validation of hardware error recovery mechanisms for the SPARC64 V microprocessor. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
92. Albert Meixner, Daniel J. Sorin: Detouring: Translating software to circumvent hard faults in simple cores. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
93. Cristian Constantinescu, Ishwar Parulkar, R. Harper, Sarah Michalak: Silent Data Corruption - Myth or reality? The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
94. Lorenzo Keller, Prasang Upadhyaya, George Candea: ConfErr: A tool for assessing resilience to human configuration errors.The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
95. Manish Marwah, Shivakant Mishra, Christof Fetzer: Enhanced server fault-tolerance for improved user experience. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
96. Yair Amir, Brian A. Coan, Jonathan Kirsch, John Lane:Byzantine replication under attack. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
97. Nicolas Schiper, Sam Toueg: A robust and lightweight stable leader election service for dynamic systems. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
98. Yawei Li, Zhiling Lan: A fast restart mechanism for checkpoint/recovery protocols in networked environments. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
99. Jose Fonseca, Marco Vieira: Mapping software faults with web security vulnerabilities. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.

100. Wei Yu, Nan Zhang, Xinwen Fu, Riccardo Bettati, Wei Zhao: On localization attacks to Internet Threat Monitors: An information-theoretic framework. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
101. Poul E. Heegaard, Kishor S. Trivedi: Survivability quantification of communication services. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
102. Ningfang Mi, Alma Riska, Evgenia Smirni, Erik Riedel: Enhancing data availability in disk drives through background activities. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
103. Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift: Analyzing the effects of disk-pointer corruption. The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings. IEEE Computer Society 2008.
104. Roberto Perdisci, Manos Antonakakis, Xiapu Luo, Wenke Lee: WSEC DNS: Protecting recursive DNS resolvers from poisoning attacks. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
105. Yao Zhao, Sagar Vemuri, Jiazhen Chen, Yan Chen, Hai Zhou, Zhi Fu: Exception triggered DoS attacks on wireless networks. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
106. Ying Zhang, Zheng Zhang, Zhuoqing Morley Mao, Y. Charlie Hu: HC-BGP: A light-weight and flexible scheme for securing prefix ownership. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
107. Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, Andrew S. Tanenbaum: Fault isolation for device drivers. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
108. Eric Rozier, Wendy Belluomini, Veera Deenadhayalan, Jim Hafner, K. K. Rao, Pin Zhou: Evaluating the impact of Undetected Disk Errors in RAID systems. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
109. Hari Kannan, Michael Dalton, Christos Kozyrakis: Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
110. Kevin S. Killourhy, Roy A. Maxion: Comparing anomaly-detection algorithms for keystroke dynamics. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
111. Filipe Freitas, Edgar Marques, Rodrigo Rodrigues, Carlos Ribeiro, Paulo Ferreira, Lus Rodrigues: Verme: Worm containment in overlay networks. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
112. Paulo Sousa, Alysson Neves Bessani, Wagner Saback Dantas, Fabio Souto, Miguel Correia, Nuno Ferreira Neves: Intrusion-tolerant self-healing devices for critical infrastructure protection. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
113. Luiz Eduardo Buzato, Gustavo M. D. Vieira, Willy Zwaenepoel: Dynamic content web applications: Crash, failover, and recovery analysis. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
114. Guanhua Yan, Leticia Cuellar, Stephan Eidenbenz, Nicolas W. Hengartner: Blue-Watchdog: Detecting Bluetooth worm propagation in public areas. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
115. Derek Graham, Per Strid, Scott Roy, Fernando Rodriguez: A low-tech solution to avoid the severe impact of transient errors on the IP interconnect. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
116. Jon G. Elerath: A simple equation for estimating reliability of an N+1 redundant array of independent disks (RAID). in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
117. Christian Cachin, Idit Keidar, Alexander Shraer: Fail-Aware Untrusted Storage. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
118. Lawall, J.L.;Brunel, J.;Palix, N.; Hansen, R.R.; Stuart, H.; Muller, G.;: A declarative approach to finding API protocols and bugs in Linux code. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.

119. Marco Vieira, Nuno Antunes, and Henrique Madeir;: using web security scanners to detect vulnerabilities in web services. in Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE 2009.
120. Towards Understanding the Effects of Intermittent Hardware Faults on Programs, Layali Rashid, Karthik Pattabiraman, Sathish Gopalakrishnan,in: Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on June 28 2010-July 1 2010.
121. Gate Input Reconfiguration for Combating Soft Errors in Combinational Circuits, Warin Sootkaneung, Kewal K. Saluja,in: Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on June 28 2010-July 1 2010.
122. A Concept of a Trust Management Architecture to Increase the Robustness of Nano Age Devices Thilo Pionteck, University of Lbeck; Werner Brockmann,in: Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on June 28 2010-July 1 2010.
123. Detecting selfish carrier-sense behavior in WiFi networks by passive monitoring,Paul, U., Das, S.R., Maheshwari, R., in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
124. Detecting Sybil Nodes in Wireless Networks with Physical Layer Network Coding Weichao Wang (University of North Carolina, Charlotte), Di Pu, Alex Wyglinski (Worcester Polytechnic Institute) , in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
125. Code-M: A non-MDS erasure code scheme to support fast recovery from up to two-disk failures in storage systems, Shenggang Wan,Qiang Cao,Changsheng Xie,Eckart, B.,Xubin He; in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
126. Decoding STAR Code for Tolerating Simultaneous Disk Failure and Silent Errors, Jianqiang Luo (Wayne State University), Cheng Huang (Microsoft Research), Lihao Xu (Wayne State University), in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
127. Data Recovery for Web Applications, Istemi Ekin Akkus, Ashvin Goel (University of Toronto), in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
128. Experimental Validation of a Fault Tolerant Microcomputer System against Intermittent Faults. J. Gracia-Moran, D. Gil-Tomas, L. J. Saiz-Adalid, J. C. Baraza, P. J. Gil-Vicente (Universidad Politcnica de Valencia, in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
129. An Empirical Investigation of Fault Types in Space Mission System Software, Michael Grottke (University of Erlangen-Nuremberg), Allen P. Nikora (California Institute of Technology), Kishor S. Trivedi (Duke University) in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
130. Efficient Eventual Consistency in Pahoehoe, an Erasure-Coded Key-Blob Archive ,Eric Anderson, Xiaozhou Li, Arif Merchant, Mehul A. Shah, Kevin Smathers, Joseph Tucek, Mustafa Uysal, Jay J. Wylie (Hewlett-Packard Laboratories), in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
131. Checkpointing Orchestration for Performance Improvement ,Hui Jin (Illinois Institute of Technology), in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
132. Transient Fault Models and AVF Estimation Revisited, Nishant J. George, Carl R. Elks, Barry W. Johnson, John Lach (University of Virginia), in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
133. DataGuard: Dynamic Data Attestation in Wireless Sensor Networks ,Dazhi Zhang, Donggang Liu (The University of Texas at Arlington), in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
134. Making Hadoop MapReduce Byzantine Fault-Tolerant ,Alysson N. Bessani, Vinicius V. Cogo, Miguel Correia, Pedro Costa, Marcelo Pasin, Fabricio Silva; Universidade de Lisboa, Faculdade de Ciencias, LASIGE Lisboa, Portugal Luciana Arantes, Olivier Marin, Pierre Sens, Julien Sopena; LIP6, Universite de Paris 6, INRIA Rocquencourt Paris, France, in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
135. Turquois: Byzantine Consensus in Wireless Ad Hoc Networks, Henrique Moniz, Nuno Ferreira Neves, Miguel Correia (University of Lisboa), in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
136. Generic Construction of Consensus Algorithms for Benign and Byzantine Faults Olivier Rutti, Zarko Milosevic, Andre Schiper (Ecole Polytechnique Federale de Lausanne) , in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
137. Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas , Marco Serafini, Peter Bokor, Dan Dobre, Matthias Majuntke, Neeraj Suri (Technische Universitat Darmstadt) , in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.

138. Zzyzx: Scalable Fault Tolerance through Byzantine Locking, James Hendricks, Shafeeq Sinnamohideen, Gregory R. Ganger (Carnegie Mellon University), Michael K. Reiter (University of North Carolina at Chapel Hill), , in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
139. Doubly-Expedited One-Step Byzantine Consensus , Nazreen Banu, Taisuke Izumi, Koichi Wada (Nagoya Institute of Technology),, in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.
140. Detecting vulnerabilities in C programs using trace-based testing , DAzhi Zhang, Donggang liu, Yu Lei, David Kung, Christoph Csallner, Wenhua Wang, in: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on June 28 2010-July 1 2010.

Power and Limits of Distributed Computing Shared Memory Models

Sergio Rajsbaum† Michel Raynal⋆,‡

† Instituto de Matemáticas, UNAM, Mexico City, D.F. 04510, Mexico
⋆Institut Universitaire de France
‡IRISA, Université de Rennes 1, France
rajsbaum@math.unam.mx, raynal@irisa.fr

Abstract. Due to the advent of multicore machines, shared memory distributed computing models taking into account asynchrony and process crashes are becoming more and more important. This paper visits some of the models for these systems, and analyses their properties from a computability point of view. Among them, the snapshot model and the iterated model are particularly investigated. The paper visits also several approaches that have been proposed to model crash failures. Among them, the *wait-free* case where any number of processes can crash is fundamental. The paper also considers models where up to t processes can crash, and where the crashes are not independent. The aim of this survey is to help the reader to better understand recent advances on what is known about the power and limits of distributed computing shared memory models and their underlying mathematics.

Keywords: Adversary, Agreement, Asynchronous system, Borowsky-Gafni's simulation, Concurrency, Core, Crash failure, Distributed computability, Distributed computing model, Fault-Tolerance, Iterated model, Liveness, Model equivalence, Obstruction-freedom, Progress condition, Recursion, Resilience, Shared memory system, Snapshot, Survivor set, Task, Topology, Wait-freedom.

1 Introduction

Sequential computing vs distributed computing Modern computer science was born with the discovery of the Turing machine model, that captures the nature and the power of sequential computing, and with the proofs of equivalence of this model with all other known models of a computer (e.g., Post systems, Church's lambda calculus, etc.). This means that the *functions* that can be computed in one model are exactly the same that the ones that can be computed in another model: these sequential computing models are defined by the same set of computable functions.

An asynchronous distributed computing model consists of a set of processes (individual state machines) that communicate through some communication medium and satisfy some failure assumptions. *Asynchronous* means that the speed of processes is entirely arbitrary: each one proceeds at its own speed which can vary and is always independent of the speed of other processes. Other timing assumptions are also of interest. In a *synchronous* model, processes progress in a lock-step manner, while in a *partially synchronous* system the speed of processes is not as tightly related.

If the components (processes and communication media) cannot fail, and each process is a Turing machine, then the distributed system is equivalent to a sequential Turing machine, from the computability point of view. Namely, processes can communicate to each other everything they know, and they compute locally any (Turing-computable) function. In this sense the power of failure-free distributed computing is the same as the one of sequential computing. Unfortunately the situation is different when processes are prone to failures.

Asynchronous distributed computing in presence of failures We consider here the case of the most benign process failure model, namely, the crash failure model. This means that, in addition to proceeding asynchronously, a process may crash in an unpredictable way (premature stop). Moreover, crashes are stable: a crashed process does not recover.

The net effect of asynchrony and process crashes gives rise to a fundamental feature of distributed computing: a process may always have uncertainty about the state of other processes. Processes cannot compute the same global state of the system, to simulate a sequential computation. Actually, in distributed computing we are interested in focusing on *distributed* aspects of computation, and thus we eliminate any restrictions on local, sequential computation. That is, when studying distributed computability (and disregard complexity issues), we model each process by an infinite state machine. We get models whose power is orthogonal to the power of a Turing machine. Namely, each process can compute functions that are not Turing-computable, but the system as a whole cannot solve problems that are easily solvable by a Turing machine (in a centralized manner).

The decision problems encountered in distributed systems, called *tasks*, are indeed *distributed*: each process has only part of the input to the problem. After communicating with each other, each process computes a part of the solution to the problem. A task specifies the possible inputs, and which part of the input gets each process. The input/output relation of the task, specifies the legal outputs for each input, and which part of the output can be produced by each process. From a computability point of view, a distributed system where even a single process may crash cannot solve tasks that can be computed by a Turing machine.

The multiplicity of distributed computing models In this paper we consider the simplest case, where processes can fail only by crashing. Even in the case of crash failures, several models have been considered in the past, by specifying how many processes can fail, if these failures are independent or not, and if the shared memory can also fail or not. The underlying communication model can also take many forms. The most basic is when processes communicate by reading and writing to a shared memory. However, stronger communication objects are needed to be able to compute certain tasks. Also, some systems are better modeled by message passing channels. Plenty of distributed computing models are encountered in the literature, with combination of these and other assumptions. A “holy grail” quest is the discovery of a basic distributed model that could be used to study essential computability properties, and then generalize or extrapolate results to other models, by systematic reductions and simulations. This would be great because, we would be able to completely depart from the situation of early distributed computing research, and instead of working on specific results suited to par-

ticular models only, a basic model allow us to have more general positive (algorithms) or negative (lower bounds and impossibility) results.

There is evidence that the basic model where processes communicate by atomically reading and writing a shared memory, and any number of them can crash, is fundamental. This paper considers this base, *wait-free* model, motivated by the following reasons. First, the asynchronous read/write communication model is the least powerful non-trivial shared memory model. Second, it is possible to simulate an atomic read/write register on top of a message-passing system as soon as less than half of the processes may crash [6,55] (but if more than half of the processes may crash, a message passing system is less powerful). Third, it has been observed that techniques used to analyze the read/write model can be extended to analyze models with more powerful shared objects (e.g., [32]), and that results about task computability when bounds on the number of failures are known can be reduced to the wait-free case via simulations e.g. [13].

Safety and liveness properties As far as safety an liveness properties are concerned, the paper considers mainly linearizability and wait-freedom. *Linearizability* means that the shared memory operations appear as if they have been executed sequentially, each operation appearing as being executed between its start event and its end event [36] (linearizability generalizes the atomicity notion introduced for shared read/write registers in [43] to any object type). *Wait-freedom* means that any operation on a shared object invoked by a non-faulty process (a process that does not crash) does terminate whatever the behavior of the other processes, i.e., whatever their asynchrony and failure pattern [29] (wait-freedom can be seen as starvation-freedom despite any number of process crashes).

Content of the paper This survey is on the power and limits of shared memory distributed computing models to solve tasks, in environments where processes can fail by crashing. It takes the approach that the write-snapshot wait-free iterated model is at the center of distributed computing theory. Results and techniques about this model can be extrapolated to the usual read/write models (where registers can be accessed many times) and message passing modes. Also, they can be extrapolated to models where bounds on the number of failures are known, or where failures are correlated.

It first defines what is a task (the distributed counterpart of a function in sequential computing) in Section 2. Then, Section 3 presents and investigates the base asynchronous read/write distributed computing model and its associated snapshot abstraction that makes programs easier to write, analyze and prove. Next, Section 4 considers the iterated write-snapshot model that is more structured than the base write/snapshot model. Interestingly, this model has a nice mathematical structure that makes it very attractive to study properties of shared memory-based distributed computing.

Section 5 considers the case where the previous models are enriched with failure detectors. It shows that there is a tradeoff between the computational structure of the model and the power added by a failure detector. Section 6 considers the case of a very general failure model, namely the adversary failure model. Section 7 discusses the BG simulation (and its variants) that reduces questions about task solvability under other adversaries, to the simplest adversary, namely to the wait-free case. The BG simulation relies on a new object type that is called *safe agreement*. Section 8 presents a new

implementation of the safe agreement object type based on the iterated model. Finally, Section 9 concludes the paper.

2 What is a task?

As already indicated, a *task* is the distributed counterpart of the notion of a *function* encountered in sequential computing. In a task T, each of the n processes starts with an input value and each process that does not crash has to decide on an output value such the set of output values has to be permitted by the task specification. More formally we have the following where all vectors are n-dimensional (n being the number of processes) [34].

Definition A task T is a triple $(\mathcal{I}, \mathcal{O}, \Delta)$ where $\mathcal{I}$ is a set of input vectors, $\mathcal{O}$ is a set of output vectors, and Δ is a relation that associates with each $I \in \mathcal{I}$ at least one $O \in \mathcal{O}$.

The vector $I \in \mathcal{I}$ is the input vector where, for each entry i, $I[i]$ is the private input of process p_i. Similarly O describes the output vector where $O[i]$ is the output that should be produced by process p_i. $\Delta(I)$ defines which are the output vectors legal for the input vector I.

Solving a task Roughly speaking, an algorithm $\mathcal{A}$ wait-free solves a task T if the following holds. In any run of $\mathcal{A}$, each process p_i starts with an input value in_i such that $\exists I \in \mathcal{I}$ with $I[i] = in_i$ (we say "p_i proposes in_i") and each non-faulty process p_j eventually computes an output value out_j (we say "p_j decides out_j") such that $\exists O \in \Delta(I)$ with $O[j] = out_j$ for all processes p_j that have computed an output value.

Examples of tasks The most famous task is *consensus* [20]. Each input vector I defines the values proposed by the processes. An output vector O is a vector whose entries contain the same value and Δ is such that $\Delta(I)$ contains all vectors whose single value is a value of I. The k-set agreement task relaxes consensus allowing up to k different values to be decided [18]. Other examples of tasks are renaming [7] and weak symmetry breaking (see [15] for an introductory survey), and k-simultaneous consensus [4].

3 Base shared memory models

3.1 The wait-free read/write model

Base wait-free model This computational model is defined by n sequential asynchronous processes p_1, ..., p_n that communicate by reading and writing one-writer/multi-reader (1WMR) reliable atomic registers. Moreover up to $n - 1$ processes may crash. Given a run of an algorithm, a process that crashes is *faulty* in that run, otherwise it is *non-faulty* (or *correct*).

This is the well-know *wait-free* shared memory distributed model. As processes are asynchronous and the only means they have to communicate is reading and writing atomic registers, it follows that the main feature of this model is the impossibility for a process p_i to know if another process p_j is slow or has crashed. This "indistinguishability" feature lies at the source of several impossibility results (e.g., the consensus impossibility [44]).

The case of an unreliable shared memory The previous model assumes that the atomic registers are *reliable*: a read or a write always returns and the atomicity behavior is always provided.

In a real distributed system potentially any of its components could fail. As we are interested in benign crash failures (the simplest kind of failures), we might also consider register crash failures, i.e., the case where registers stop working. Two types of such failures can be distinguished: responsive and non-responsive. In the responsive type, a register fails if it behaves correctly until some time, after which every read or write operation returns the default value $\perp$. hence, the register behaves correctly until it crashes (if it ever crashes) and then the failure can be detected. Responsive crash is sometimes called *fail-stop*. In the non-responsive type, after a register has crashed, its read and write operations never terminate, they remain pending forever. Non-responsive crash is sometimes called *fail-silent*.

It is possible to build reliable atomic registers on top of crash-prone base atomic registers. More precisely, let us assume that we want to cope with the crash of up to t unreliable base registers (t-register-resilience). There are t-register-resilient wait-free algorithms that build an atomic register [27]. If failure are responsive (resp., non-responsive) $t+1$ (resp., $2t+1$) base registers are necessary and sufficient. As register crash failures can be overcome, we consider only reliable registers in the rest of the paper.

3.2 The snapshot abstraction

Designing correct distributed algorithms is hard. Thus, it is interesting to construct out of read/write registers communication abstractions of higher level. A very useful abstraction (which can be efficiently constructed from read/write registers) is a *snapshot* object [1] (more developments on snapshot objects can be found in [5,8,37]).

A snapshot abstracts an array of 1WMR atomic registers with one entry per process and provides them with two operations denoted $X.\mathsf{write}(v)$ and $X.\mathsf{snapshot}()$ where X is the corresponding snapshot object [1]. The former assigns v to $X[i]$ (and is consequently also denoted $X[i] \leftarrow v$). Only p_i can write $X[i]$. The latter operation, $X.\mathsf{snapshot}()$, returns to the invoking process p_i the current value of the whole array X. The fundamental property of a snapshot object is that all write and snapshot operations appear as if they have been executed atomically, which means that a snapshot object is linearizable [36]. These operations can be wait-free built on top of atomic read/write registers (the best implementation known so far has $O(n \log n)$ time complexity [3]). Hence, a snapshot object provides the programmer with a high level shared memory abstraction but does not provide her/him with additional computational power.

3.3 A progress condition weaker than wait-freedom

As already indicated, the progress condition associated with an algorithm solving a task in the base shared memory distributed computing model is wait-freedom (any correct process has to decide has a value whatever the behavior of the other processes). This is the strongest progress condition one can think of but is not the only one. We present here a weaker progress condition.

Obstruction-freedom Wait-freedom is independent of the concurrency pattern. Differently, obstruction-freedom involves the concurrency pattern. It states that, if a correct process executes alone during a long enough period, it has to decide a value [30]. The words "long enough period" are due to asynchrony, they capture the fact that a process needs time to execute the algorithm.

As we can see, there are concurrency patterns in which no process is required to decide when we consider the obstruction-freedom progress condition. The important point to notice is that any algorithm that solves a task with the obstruction-freedom progress condition has to always ensure the task safety property: if processes decide, the decided values have to be correct.

Obstruction-freedom vs wait-freedom Obstruction-freedom is a progress condition that is trivially weaker (i.e., less constraining) than wait-freedom. This has a consequence on task computability. As an example, while it is impossible to wait-free solves the consensus problem in the base read/write (or snapshot) shared memory model, it is possible to solve it in the same model when the wait-freedom requirement is replaced by obstruction-freedom.

More generally, when conflicts are rare, obstruction-freedom can be used instead of wait-freedom.

4 The iterated write-snapshot model

4.1 The iterated write-snapshot model

Attempts at unifying different read/write distributed computing models have restricted their attention to a subset of *round-based* executions e.g. [14,33,46]. The approach introduced in [12] proposes an *iterated* model in which processes execute an infinite sequence of rounds, and in each round communicate through a specific object called *one-shot write-snapshot* object. This section presents this shared memory distributed computing model [49].

One-shot write-snapshot object A *one-shot write-snapshot* object abstracts an array $WS[1..n]$ that can be accessed by a single operation denoted write_snapshot() that each process invokes at most once. That operation pieces together the write() and snapshot() operations presented previously [11]. Intuitively, when a process p_i invokes write_snapshot(v) it is as if it instantaneously executes a write $WS[i] \leftarrow v$ operation followed by an WS.snapshot() operation. If several IS.write_snapshot() operations are executed simultaneously, then their corresponding writes are executed concurrently, and then their corresponding snapshots are also executed concurrently (each of the concurrent operations sees the values written by the other concurrent operations): they are set-linearizable [48]. $WS[1..n]$ is initialized to $[\perp, \ldots, \perp]$.

When invoked by a process p_i, the semantics of the write_snapshot() operation is characterized by the following properties, where v_i is the value written by p_i and sm_i, the value (or *view*) it gets back from the operation. A view sm_i is a set of pairs (k, v_k), where v_k corresponds to the value in p_k's entry of the array. If $WS[k] = \perp$, the pair

$(k, \perp)$ is not placed in sm_i. Moreover, we assume that $sm_i = \emptyset$, if the process p_i never invokes WS.write_snapshot(). These properties are:

- Self-inclusion. $\forall i : (i, v_i) \in sm_i$.
- Containment. $\forall i, j : sm_i \subseteq sm_j \ \vee \ sm_j \subseteq sm_i$.
- Immediacy. $\forall i, j : [(i, v_i) \in sm_j \ \wedge \ (j, v_j) \in sm_i] \ \Rightarrow \ (sm_i = sm_j)$.
- Termination. Any invocation of WS.write_snapshot() by a correct process terminates.

The self-inclusion property states that a process sees its write, while the containment properties states that the views obtained by processes are totally ordered. Finally, the immediacy property states that if two processes "see each other", they obtain the same view (the size of which corresponds to the *concurrency* degree of the corresponding write_snapshot() invocations).

The iterated model In the *iterated write-snapshot model* (IWS) the shared memory is made up of an infinite number of one-shot write-snapshot objects $WS[1], WS[2], \ldots$ These objects are accessed sequentially and asynchronously by each process, according to the round-based pattern described in Figure 1 where r_i is p_i's current round number.

```
r_i ← 0;
loop forever r_i ← r_i + 1;
             local computations; compute v_i;
             sm_i ← WS[r_i].write_snapshot(v_i);
             local computations
end loop.
```

Fig. 1. Generic algorithm for the iterated write-snapshot model

A fundamental result Let us observe that the IWS model requires each correct process to execute an infinite number of rounds. However, it is possible that a correct process p_1 is unable to receive information from another correct process p_2. Consider a run where both execute an infinite number of rounds, but p_1 is scheduled before p_2 in every round. Thus, p_1 never reads a value written to a write-snapshot object by p_2. Of course, in the usual (non-iterated read/write shared memory) asynchronous model, two correct processes can always eventually communicate with each other. Thus, at first glance, one could intuitively think that the base read/write model and the IWS model have different computability power. The fundamental result associated with the IWS model is captured by the following theorem that shows that the previous intuition is incorrect.

Definition 1. *A task is* bounded *if its set of input vectors $\mathcal{I}$ is finite.*

Theorem 1. [12] *A bounded task can be wait-free solved in the* 1WMR *shared memory model if and only if it can be wait-free solved in the* IWS *model.*

Why the IWS model? The interest of the IWS model comes from its elegant and simple round-by-round iterative structure. It restricts the set of interleavings of the shared memory model without restricting the power of the model. Its runs have an elegant recursive structure: the structure of the global state after $r+1$ rounds is easily obtained from the structure of the global state after r rounds. This implies a strong correlation with topology (see the next section) which allows for an easier analysis of wait-free asynchronous computations to prove impossibility results, e.g. [31,32]. The recursive structure of runs also facilitates the design and analysis of algorithms (see Section 7 for an example), e.g. [25].

4.2 A mathematical view

The properties that characterize the write_snapshot() operation are represented in Figure 2 for the case of three processes. In the topology parlance, this picture represents a *simplicial complex,*, i.e., a family of sets closed under containment. Each set, which is called a *simplex*, represents the views of the processes after accessing the one-shot write-snapshot object associated with the corresponding round. The *vertices* are 0-simplexes (size one); edges are 1-simplexes (size two); triangles are 2-simplexes (size three) and so on. Each vertex is associated with a process p_i and labeled with its name.

The highlighted 2-simplex on the picture at the left represents a run where p_1 and p_3 access the object concurrently, both get the same views seeing each other, but not seeing p_2, which accesses the object later, and gets back a view with the 3 values written to the object. But p_2 cannot tell the order in which p_1 and p_3 access the object; the other two runs are indistinguishable to p_2, where p_1 accesses the object before p_3 and hence gets back only its own value or the opposite. These two runs are represented by the 2-simplexes at the bottom corners of the left picture. Thus, the vertices at the corners of the complex represents the runs where only one process p_i accesses the object, and the vertices in the edges connecting the corners represent runs where only two processes access the object. The triangle in the center of the complex represents the run where all three processes access the object concurrently, and get back the same view.

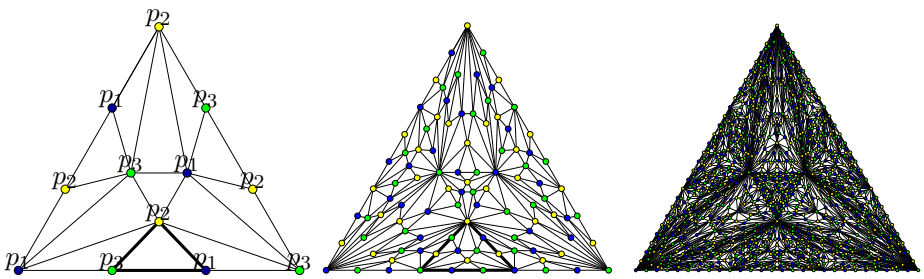

Fig. 2. One, two and three rounds in the iterated write-snapshot (IWS) model

Hence, the state of an execution after the first round (with which is associated the write-snapshot object $WS[1]$) is represented by one of the internal triangles of the left picture (e.g., the one discussed previously that is represented by the bold triangle in the pictures). Then, the state of that execution after the second round (with which is associated the write-snapshot object $WS[2]$) is represented by one of the small triangles inside the bold triangle in the picture in the middle. Etc. More generally, as shown in Figure 2, one can see that, in the write-snapshot iterated model, at every round, a new complex is constructed recursively by replacing each simplex by a one-round complex.

4.3 A recursive write-snapshot algorithm

Figure 3 presents a read/write algorithm that implements the write_snapshot() operation. Interestingly, this algorithm is recursive [15,25]. A proof can be found in [15]. To allow for a recursive formulation, an additional recursion parameter is used. More precisely, in a round r, a process invokes SM.write_snapshot(n, v) where the initial value of the recursion parameter is n and SM stands for $WS[r]$.

SM is a shared array of size n (initialized to $[\bot, \ldots, \bot]$ and such that each $SM[x]$ is an array of n 1WnR atomic registers. The atomic register $SM[x][i]$ can be read by all processes but written only by p_i.

Let us consider the invocation SM.write_snapshot(x, v) issued by p_i. Process p_i first writes $SM[x][i]$ and reads (not atomically) the array $SM[x][1..n]$ that is associated with the recursion parameter x (lines 01-02). Then, p_i computes the set of processes that have already attained the recursion level x (line 03; let us note that recursion levels are decreasing from n to $n-1$, etc.). If the set of processes that have attained the recursion level x (from p_i's point of view) contains exactly x processes, p_i returns this set as a result (lines 04-05). Otherwise less than x processes have attained the recursion level x. In that case, p_i recursively invokes SM.write_snapshot$(x-1)$ (line 06) in order to attain and stop at the recursion level y attained by exactly y processes.

```
operation SM.write_snapshot(x, v):
        % x (n ≥ x ≥ 1) is the recursion parameter %
(01) SM[x][i] ← v;
(02) for 1 ≤ j ≤ n do aux_i[j] ← SM[x][j] end for ;
(03) pairs_i ← {(j, v') | ∃j such that aux_i[j] = v' ≠ ⊥};
(04) if (|pairs_i| = x)
(05)       then sm_i ← pairs_i
(06)       else sm_i ← SM.write_snapshot(x − 1, v)
(07) end if;
(08) return(sm_i).
```

Fig. 3. A recursive write-snapshot algorithm (code for p_i)

The cost of a shared memory distributed algorithm is usually measured by the number of shared memory accesses, called *step complexity*. The step complexity of p_i's invocation is $O(n(n - |sm_i| + 1))$.

4.4 Iterative model vs recursive algorithm

It is interesting to observe that the iterative structure that defines the IWS model and the recursion-based formulation of the previous algorithm are closely related notions. In one case iterations are at the core of the model while in the other case recursion is only an algorithmic tool. However, the runs of a recursion-based algorithm are of an iterated nature: in each iteration only one array of registers is accessed, and the array is accessed only in this iteration.

5 Enriching a system with a failure detector

5.1 Adding failure detectors to the IWS model

The concept of a failure detector This concept has been introduced by Chandra and Toueg [17] (see [53] for an introductory survey). Informally, a failure detector is a device that provides each process p_i with information about process failures, through a local variable fd_i that p_i can only read. Several classes of failure detectors can be defined according to the kind and the quality of the information on failures that has to be delivered to the processes.

Of course, a non-trivial failure detector requires that the system satisfies additional behavioral assumptions in order to be implemented. The interested reader will find such additional behavioral assumptions and corresponding algorithms implementing failure detectors of several classes in chapter 7 of [55].

An example One of the most known failure detectors is the eventual leader failure detector denoted Ω [16]. This failure detector is fundamental because it encapsulates the weakest information on failures that allows consensus to solved in a base read/write asynchronous system.

The output provided by Ω to each (non crashed) process p_i is such that fd_i always contains a process identity (validity). Moreover, there is a finite time τ after which all local failure detector outputs fd_i contains forever the same process identity and it is the identity of a correct process (eventual leadership). The time τ is never explicitly know by the processes. Before τ, there is an anarchy period during which the local failure detector outputs can be arbitrary.

A result As indicated, the consensus problem cannot be solved in the base read read/write system [44] but can be solved as soon as this system is enriched with Ω.

On another side (see Theorem 1), the base shared memory model and the IWS model have the same wait-free computability power for bounded tasks. Hence a natural question: Is this computability power equivalence preserved when both models are enriched with the same failure detector? Somehow surprisingly, the answer to this question is negative. More precisely, we have the following.

Theorem 2. [51] *For any failure detector FD and bounded task T, if T is wait-free solvable in the model* IWS *enriched with FD, then T is wait-free solvable in the base shared memory model without failure detector.*

Intuitively, this negative result is due to the fact that the IWS model is too much structured to benefit from the help of a failure detector.

5.2 How to circumvent the previous negative result

A way to circumvent the previous negative result consists in "embedding" the failure detector inside the write_snapshot() operation. More precisely, the infinite sequence of invocations $WS[1]$.write_snapshot(), $WS[2]$.write_snapshot(), etc., issued by any process p_i has to satisfy an additional property that depends on the corresponding failure detector. This approach has given rise to the IRIS model described in [52].

The property to obtain the power of Ω Let us enrich the IWS model with the following property

$$PR(\Omega) \equiv \big(\exists \ell, \exists r : \forall r' \geq r : sm_\ell^{r'} = \{\ell\}\big).$$

$PR(\Omega)$ means that there is a round r and a process p_ℓ such that, at every round $r' \geq r$, any process p_i that executes $sm_i \leftarrow IS[r']$.write_snapshot() sees ℓ in its view sm_i (i.e., $\ell \in sm_i$). Said differently, whatever the concurrency degree among the $IS[r']$.write_snapshot() invocations issued by processes during r', the invocation issued by p_ℓ is always set-linearized alone and before the other invocations. So, p_ℓ always obtains the same view that contains only itself, namely, $\forall r' \geq r$: $sm_\ell^{r'} = sm_\ell^r = \{\ell\}$. It then follows from the containment property of the immediate snapshot operation, that the view $sm_j^{r'}$ obtained by any process p_j that executes a round $r' \geq r$ is such that $\ell \in sm_j^{r'}$.

The IWS model enriched with $PR(\Omega)$ at work It is shown in [50,52] that the base reader/write (snapshot) model enriched with Ω and the IWS model enriched with $PR(\Omega)$ have the same computability power for bounded tasks. In order to illustrate this computability equivalence, let us consider the well-known consensus problem. Let us remember that, in this decision problem, each process proposes a value, and each correct process has to decide a value (termination) such that a decided value is a proposed value (validity) and no two processes decide different values (agreement).

Several distributed algorithms solving consensus in the base read/write model enriched with Ω are described in the literature (e.g., [9,28]). A consensus algorithm suited to the IWS model enriched with $PR(\Omega)$ is described in Figure 4 (this algorithm is from [50]).

The value proposed by p_i is v_i and $\perp$ is a default value that cannot be proposed by a process. In addition to r_i (the current round number), a process and manages four local variables:

- The local variables est_i and dec_i are directly related to the decision value: est_i (initialized to v_i) contains p_i's current estimate of the decision value, while dec_i is a write-once local variable (initialized to $\perp$) that eventually contains the value decided by p_i.
- sm_i and tm_i are two local variables used to contain the snapshot value returned by the invocations to the operation write_snapshot() at the odd and even round numbers, respectively. The variable sm_i contains a set of triples, while tm_i contains a set of set of triples (i.e., a set of tm_j values).

```
r_i ← 0; est_i ← v_i; dec_i ← ⊥;
(01) loop forever
(02)      r_i ← r_i + 1;
(03)      sm_i ← WS[r_i].write_snapshot(⟨i, est_i, dec_i⟩);
(04)      r_i ← r_i + 1;
(05)      tm_i ← WS[r_i].write_snapshot(sm_i);
(06)      if (∃sm : (sm ∈ tm_i) ∧ (⟨−, −, dec⟩ ∈ sm with dec ≠ ⊥))
(07)        then if dec_i = ⊥ then est_i ← dec; dec_i ← dec end if
(08)        else if (∃sm : (sm ∈ tm_i) ∧ (sm = {⟨−, est, −⟩}))
(09)                then est_i ← est;
(10)                     if tm_i = {sm} ∧ sm = {⟨i, est, −⟩} then dec_i ← est_i end if
(11)             end if
(12)      end if
(13) end loop.
```

Fig. 4. A consensus algorithm for the IWS model enriched with $PR(\Omega)$ (code for p_i) [50]

A process p_i executes a sequence of pairs of rounds, namely, $(1, 2)$, then $(3, 4)$, etc. During the first round $(r - 1)$), p_i writes the triple $\langle i, est_i, dec_i\rangle$ in the one-shot immediate snapshot object $WS[r - 1]$, from which it obtains a set of such triples (lines 02-03). During the second round, p_i writes into $WS[r]$, the set of triples sm_i it has just obtained, and obtains a corresponding set of set of triples tm_i (lines 04-05).

Then, p_i considers the values it has obtained from the one-shot immediate snapshot objects $WS[r - 1]$ and $WS[r]$. If p_i sees that a value (dec) has already been decided (line 06) while itself has not yet decided, it decides that value (line 07). Otherwise, if the set of set of triples tm_i it has obtained from $WS[r]$ contains a set sm with a single triple (line 08), p_i adopts the estimate value of that triple sm (line 09) as its new estimate. Moreover, if additionally, tm_i contains a single triple and that triple is from p_i itself ($tm_i = \{\{\langle i, est_i, -\rangle\}\}$, line 10), then p_i decides its current estimate. Let us observe that $tm_i = \{\{\langle i, est_i, -\rangle\}\}$ means that p_i was the only "winner" of both the rounds $r - 1$ and r (the invocations $WS[r - 1]$.write_snapshot() and $WS[r]$.restricited_w_snapshot() issued by p_i have been set-linearized before the invocations from the other processes).

6 From the wait-free model to the adversary model

Adversaries are a very useful abstraction to represent subsets of executions of a distributed system. The idea is that, if one restricts the set of possible executions, the system should be able to compute more tasks. For example, the *condition based approach* [47] restricts the set of inputs of a task (and hence the corresponding executions), and allows to solve more tasks, or to solve tasks faster. Various adversaries have been considered in the past to model failure restrictions, as we shall now describe.

6.1 The notion of an adversary

In the wait-free model, any number of process can crash. In practice one sometimes estimates a bound t on how many processes can be expected to crash. However, often the crashes are not independent, due to processes running on the same core, or on the same subnetwork, for example.

Wait-freedom It is easy to see that wait-freedom is the least restrictive adversary, i.e., the adversary that contains all the (non-empty) subsets of processes, namely, the sets of processes that may be alive in some execution. Hence, a wait-free algorithm has to work whatever the number of process crashes.

t-Faulty process resilience The *t-faulty process resilient* failure model (also called *t-threshold* model) considers that, in any run, at most t processes may crash. Hence, the corresponding adversary is the set of all the sets of $(n - t)$ processes plus all their supersets: for each such set, there are executions where the processes that do not crash consist of this set.

Cores and survivor sets The notion of t-process resilience is not suited to capture the case where processes fail in a dependent way. This has motivated the introduction of the notions of *core* and *survivor set* [42].

A core C is a minimal set of processes such that, in any run, some process in C does not fail. A survivor set S is a minimal set of processes such that there is a run in which the set of non-faulty processes is exactly S. Let us observe that cores and survivor sets are dual notions (any of them can be obtained from the other one). As an example let us consider a system of 4 processes p_1,p_2, p_3 and p_4 where two cores are defined, namely $\{p_1, p_2\}$ and $\{p_3, p_4\}$. The corresponding survivor sets are $\{p_1, p_3\}$, $\{p_1, p_4\}$, $\{p_2, p_3\}$ and $\{p, p_4\}$.[1].

Computability results associated with the set agreement problem have been generalized from t-resilience to cores in [31]. A connection relating any adversary A (see the definition below) that is superset-closed (i.e., $(s \in A) \Rightarrow (\forall s' : s \subseteq s' : s' \in A)$) and wait-freedom is presented in [24]. It is easy to see that the notion of survivor sets is more general than the notion of t-threshold resilience. It is also possible to see that more general notions failures are possible as shown below.

Adversaries The most general notion of an adversary with respect to failure dependence has been introduced in [19]. An *adversary* A is a set of sets of processes. It states that an algorithm solving a task must terminate in all the runs whose the corresponding set of correct processes is (exactly) a set of A.

As an example, Let us considers a system with four processes denoted p_1, ..., p_4. The set $A = \{\{p_1, p_2\},\{p_1, p_4\},\{p_1, p_3, p_4\}\}$ defines an adversary. An algorithm A-resiliently solves a task if it terminates in all the runs where the set of correct processes

[1] Borrowing the quorum terminology and considering cores as *quorums*, the corresponding survivor sets are then their *antiquorums* [10].

is exactly either $\{p_1, p_2\}$ or $\{p_1, p_4\}$ or $\{p_1, p_3, p_4\}$. This means that an A-resilient algorithm is not required to terminate in an execution in which the set of correct processes is exactly the set $\{p_3, p_4\}$ or the set $\{p_1, p_2, p_3\}$.

Adversaries are more general than the notion of survivor sets (this is because when we build the adversary corresponding to a set of survivor sets, due the "minimality" feature of each survivor set, we have to include all its supersets in the corresponding adversary).

On progress conditions It is easy to see that an adversary can be viewed as a liveness property that specifies the crash patterns in which the correct processes must progress. The interested reader will find more developments on progress conditions in [23,24,41,58].

6.2 Simulating the snapshot model in the iterated model

The iterated write-snapshot model (IWS) has been presented in Section 4 where we have seen that the base read/write (or snapshot) shared memory model and the IWS model are equivalent from a wait-free (bounded) task solvability point of view [12].

This section shows that this equivalence remains true when wait-freedom is replaced by an adversary-based progress condition. To that end, this section presents the simulation of the base write/snapshot memory model in a simple variant of the IWS model where wait-freedom is replaced by the adversary defined by survivor sets. This simulation is from [26].

The snapshot-based algorithm Let $\mathcal{A}$ be the snapshot-based algorithm we want to simulate. We assume without loss of generality that it is a *full information* algorithm, i.e., each time a process writes, it writes its full local state. Initially, each process p_i has an input value denoted $input_i$. Then, each process alternates between writing its state in the shared memory, taking a snapshot of the shared memory and taking this snapshot as its new local state.

The algorithm $\mathcal{A}$ solves a task with respect to an adversary A if each correct process decides a value based on its current local state in every run that is *fair* with respect to A (which means that each time a process takes a snapshot it reads new values written by processes of a survivor set of the adversary A). This means that as in [26], given a snapshot object WS, a process writes it only once but can repeatedly invokes $WS.\mathsf{snapshot}()$ until it sees new values from a survivor set of A.

Given a local state of a process, we assume that the algorithm $\mathcal{A}$ has a predicate $\mathsf{undecided}(state)$ and a decision function $\mathsf{decide}(state)$. Once a process has decided, its predicate $\mathsf{undecided}(state)$ remains forever false. The kth snapshot issued by p_i is denoted $snapshot(k, i)$ and its kth write is denoted $write(k, i)$.

As it does not lead to confusion, we use "p_i" to denote both the simulated process and the process that simulates it.

Simulation: the operation $WS[r].\mathsf{write\_\&\_snapshot}(v)$ This is the simulation operation that allows processes to coordinate and communicate. When a process p_i invokes it, it does the following. The value v is written in $WS[r][i]$ and p_i waits until the set of

processes that have written into $WS[r]$ contains a survivor set of A. When this happens, a snapshot of $WS[r]$ is taken and returned. To simplify the presentation we consider that the snapshot value that is returned is an array $sm[1..n]$.

```
(01) r_i ← 0; state_i.clock ← [0, ... , 0]; state_i[i] ← (1, input_i);
(02) loop forever
(03)   r_i ← r_i + 1;
(04)   sm_i[1..n] ← WS[r_i].write_&_snapshot(state_i);
(05)   foreach j do
(06)     state_i[j] ← sm[x][j] such that ∀y : sm[x][j].clock ≥ sm[y][j].clock;
(07)   end for;
(08)   if (Σ_{1≤j≤n} state_i[j].clock) = r_i then
         % simulation of snapshot(k, i) (with k = state_i[i].clock) which returns state_i.val
(09)     if undecided(state_i.val)
(10)       then state_i[i] ← (state_i[i].val, state_i[i].clock + 1)
                 % simul. of write(k, i) (with k = state_i[i].clock) which writes state_i[i].val
(11)       else decide(state_i.val) if not yet done
(12)     end if
(13)   end if
(14) end loop.
```

Fig. 5. The iterated simulation for adversary A (code for p_i)

Simulation: local variables A process p_i manages the following local variables.

- r_i is the local current round number of the iterated model (initially 0).
- $state_i[1..n]$ is an array of pairs such that $state_i[x].clock$ is a clock value (integer) that measures the progress of p_x as know by p_i and $state_i[x].val$ is the corresponding local state of p_x; $state_i[i].val$ is initialized to $input_i$ (the input value of p_i), the initial values of the other $state_i[j].val$ are irrelevant. The notation $state_i.clock$ is used to denote the array $[state_i[1].clock, \ldots, state_i[n].clock]$ and similarly for $state_i.val$.

Simulation: behavior of a process During a round r, the behavior of a process p_i is made up of two parts.

- First p_i writes its current local state $state_i$ in $WS[r]$ and saves in sm_i the view of the global state it obtains for that round (line 04). Then (lines 05-07) p_i computes its new local state as follows: it saves in $state_i[j]$ the most recent local state of p_j it knows ("most recent" refers to the clock values that measures p_j progress).
- Then the simulation p_i strives for making the simulation of the simulated process to progress.
  The quantity $\Sigma_{1\leq j\leq n} state_i[j].clock$ represents the current date of the global simulation from p_i's point of view. If this date is different from p_i's current round number r, then p_i is late (as far as its round number is concerned) and it consequently

proceeds to the next simulation round in order to catch up. (The proof shows that the predicate $\Sigma_{1\leq j\leq n} state_i[j].clock \leq r$ remains invariant.)
If $\Sigma_{1\leq j\leq n} state_i[j].clock = r$, the round number is OK for p_i to simulate the invocation $snapshot(k,i)$ of the simulated process where $k = state_i[i].clock$. The corresponding value that is returned is $state_i.val$.
Then, if its state is undecided (line 09), p_i makes the simulation progress by simulating $write(k,i)$ where $k = state_i[i].clock+1$ and the value written is $state_i[i].val$ (line 10).
If the state of p_i can allow for a decision, p_i decides if not yet done (line 11). Let us notice that, as soon as a process p_i has decided, it continues looping (this is necessary to prevent permanent blocking if p_i belongs to survivor sets) but its local clock is no longer increased in order to allow the correct processes to decide.

A proof of this simulation can be found in [26]. It is based on the observation that if for two processes the round number is OK, then theirs $state$ variables agree.

7 Simulating adversary models in the wait-free model: the BG simulation

7.1 The base BG simulation

The simulation of Section 6.2 showed that the same tasks can be solved in the iterated write-snapshot (IWS) model and in the base read/write (or snapshot) shared memory model, in the presence of failures. More precisely, considering adversaries defined by survivor sets, a task can be solved in the IWS model under some adversary if and only if it can be solved in the base model under the same adversary. As we have seen, the simplest adversary is the wait-free adversary. So it would be nice to reduce questions about task solvability under other adversaries to the wait-free case. This is exactly what the BG *simulation* [13] and its variants (e.g. [22,38,39]) do when considering the set of adversaries defined by t-resilience.

BG simulation: motivation and aim Let us consider an algorithm $\mathcal{A}$ that is assumed to solve a task T in an asynchronous read/write shared memory system made up of n processes, and where any subset of at most t processes may crash. Given algorithm $\mathcal{A}$ as input, the BG simulation is an algorithm that solves T in an asynchronous read/write system made up of $t+1$ processes, where up to t processes may crash. Hence, the BG simulation is a wait-free algorithm.

The BG simulation has been used to prove solvability and unsolvability results in crash-prone read/write shared memory systems. It works only for a particular class of tasks called *colorless* tasks. These are the tasks where, if a process decides a value, any other process is allowed to decide the very same value and, if a process has an input value v, then any other processes can exchange its own input by v. Thus, for colorless tasks, the BG simulation characterizes t-resilience in terms of wait-freedom, and it is not hard to see that the same holds for any other adversary defined by survivor sets.

As an example, let us assume that $\mathcal{A}$ solves consensus, despite up to $t = 1$ crash, among n processes in a read/write shared memory system. Taking $\mathcal{A}$ as input, the BG

simulation builds a $(t+1)$-process (i.e., 2-process) algorithm $\mathcal{A}'$ that solves consensus despite $t = 1$ crash, i.e., wait-free. But, we know that consensus cannot be wait-free solved in a crash-prone asynchronous system where processes communicate by accessing shared read/write registers only [20,29,44] in particular if it is made up of only two processes. It then follows that, whatever the number n of processes the system is made up of, there is no 1-resilient consensus algorithm.

The BG simulation algorithm has been extended to work with general tasks (called *colored* tasks) [22,38] and for algorithms that have access to more powerful communication objects (e.g., [39] that extends the BG simulation to objects with any consensus number x).

BG simulation: how does it work? Let $\mathcal{A}$ be an algorithm that solves a colorless decision task in the t-resilient model for n processes. The basic aim is to design a wait-free algorithm $\mathcal{A}'$ that simulates $\mathcal{A}$ in a model with $t+1$ processes. A simulated process is denoted p_j, while a simulator process is denoted q_j, with $1 \leq j \leq n$.

Each simulator q_j is given the code of every simulated process $p_1, \ldots, p_n$. It manages n threads, each one associated with a simulated process, and locally executes these threads in a fair way (e.g., using a round-robin mechanism). It also manages a local copy mem_i of the snapshot memory mem shared by the simulated processes. The code of a simulated process p_j contains invocations of $mem[j].\mathsf{write}()$ and of $mem.\mathsf{snapshot}()$. These are the only operations used by the processes $p_1, \ldots, p_n$ to cooperate. So, the core of the simulation is the design of algorithms that describe how a simulator q_i simulates these operations. These simulation algorithms are denoted $\mathsf{sim\_write}_{i,j}()$, and $\mathsf{sim\_snapshot}_{i,j}()$ whose implement-ion relies on the following object type.

The safe agreement object type This type is at the core of the BG simulation. It provides each simulator q_i with two operations, denoted $\mathsf{sa\_propose}(v)$ and $\mathsf{sa\_decide}()$, that q_i can invoke at most once, and in that order. The operation $\mathsf{sa\_propose}(v)$ allows q_i to propose a value v while $\mathsf{sa\_decide}()$ allows it to decide a value. The properties satisfied by an object of the type safe agreement are the following.

- Termination. If no simulator crashes while executing $\mathsf{sa\_propose}(v)$, then any correct simulator that invokes $\mathsf{sa\_decide}()$ returns from that invocation.
- Agreement. At most one value is decided.
- Validity. A decided value is a proposed value.

7.2 The BG simulation when base objects are stronger than read/write registers

The BG simulation considers that processes communicate through base atomic read/write registers. Hence, the question: What does happen when processes can communicate with objects that are more powerful than base read/write registers?

Consensus number of a concurrent object The synchronization/coordination power of a concurrent object in presence of asynchrony and process failures can be characterized by its *consensus number* [29]. More precisely, the consensus number of an object O is the greatest integer x such that consensus can be solved from objects O and atomic

read/write registers among x processes. If there is no such finite x, the consensus number of object O is $+\infty$.

The consensus of read/write objects is 1, the one of $Test\&Set$ objects, $Fetch\&Add$ objects, shared queues is 2, etc., until $Compare\&Swap$ or LL/SC objects whose consensus number is $+\infty$. (The interested reader will find a description of these objects in several books and articles, e.g., [2,35,45,54,56,57]). More generally, consensus numbers define an infinite hierarchy of concurrent objects such that objects at level x can solve consensus among up to x processes but not $s+1$.

The multiplicative power of consensus numbers Let $\mathcal{ASM}(n,t,x)$ denote an asynchronous system in which up to t processes may crash and where the processes can communicate through objects whose consensus number is x where $1 \leq x \leq t < n$.

The base BG simulation shows that the systems models $\mathcal{ASM}(n,t,1)$ and $\mathcal{ASM}(t+1,t,x)$ have the same computability power (for bounded tasks). A generalization of the BG simulation is presented in [39] where it is shown that the system models $\mathcal{ASM}(n,t_1,x_1)$ and $\mathcal{ASM}(n,t_2,x_2)$ have the same computability power if and only if $\lfloor \frac{t_2}{x_2} \rfloor = \lfloor \frac{t_1}{x_1} \rfloor$. This simulation shows that consensus numbers have a multiplicative power with respect to process failures, namely, $\mathcal{ASM}(n,t',x)$ and $\mathcal{ASM}(n,t,1)$ are equivalent if and only if $(t \times x) \leq t' \leq (t \times x) + x - 1$.

8 Safe agreement object type and the iterated model

Simple implementations of the safe agreement object type are described in [13,40]. We introduce here an alternative implementation that works in the iterated write-snapshot model (IWS). Actually, the proposed implementation of the operations sa_propose() and sa_decide() associated with the safe agreement type needs two iterations of the IWS model. As the safe agreement object is at the core of the BG simulation, doing so connects the BG simulation with the "iterated model" research line.

The two underlying write-snapshot objects used to implement a safe agreement object are denoted $WS[1]$ and $WS[2]$. As seen in Section 4.1 the semantics of the associated write_snapshot() operation is defined by the properties Self-inclusion, Containment, Immediacy, and Termination which have been described in that section.

operation sa_propose(v_i):
(01) $sm_i^1 \leftarrow WS[1]$.write_snapshot(v_i);
(02) $sm_i^2 \leftarrow WS[2]$.write_snapshot(sm_i^1).

Fig. 6. Operation sa_propose(v_i) in the iterated model (code for q_i)

Implementing the operation sa_propose() An algorithm implementing the safe agreement operation sa_propose() is described in Figure 6. When a simulator process q_i invokes sa_propose(v_i), where v_i is the value it proposes, it writes v_i into the first

write-snapshot object $WS[1]$ and stores the set obtained from $WS[1]$ into a local variable sm_i^1 (line 01). It then writes this result sm_i^1 into the second write-snapshot object $WS[2]$ and stores the value obtained from $WS[2]$ into its local variable sm_i^2 (line 02).

An important point here is that q_i writes atomically into $WS[2]$ (let us notice that the value that it obtains from $WS[2]$ and saves in sm_i^2 is not used). Let us also insist on the fact that sm_i^1 is a set of pairs (k, v_k) (where v_k is the value proposed by q_k to the safe agreement object) that contains at least the pair (i, v_i) written by q_i. Differently, $WS[2]$ is a set of pairs $(x, view_x)$ where $view_x$ is the value of sm_x^1, i.e., the set of pairs obtained from $WS[1]$ by q_x. Let us also remark that, after $WS[2]$.write_snapshot() has been invoked by q_i, $WS[2]$ contains at least the pair (i, sm_i^1).

```
operation sa_decide():
(03)  repeat sm_i^3 ← WS[2].scan()
(04)       until (∀ k : (k, −) ∈ sm_i^1 ⇒ (k, view_k) ∈ sm_i^3)
(05)  end repeat;
(06)  sm_i^3 ← {(k, view_k) ∈ sm_i^3 | (k, −) ∈ sm_i^1};
(07)  (−, min_view_i) ← (k, view_k) ∈ sm_i^3 such that ∀(x, view_x) ∈ sm_i^3 : |view_k| ≤ |view_x|;
(08)  let dec_i = min{v_x | (x, v_x) ∈ min_view_i}
(09)  return(dec_i).
```

Fig. 7. Operation sa_decide() in the iterated model (code for q_i)

Implementing the operation sa_decide() An algorithm implementing the second operation of the safe agreement object type, namely sa_decide(), is described in Figure 7. It is made up of two parts.

- In the first part (lines 03-05), q_i repeatedly reads $WS[2]$ until some closure property is satisfied[2]). The value read from the write-snapshot object $WS[2]$ is saved in the local variable sm_i^3. Hence, sm_i^3 is a set of pairs $(x, view_x)$ containing at least the pair (i, sm_i^1) (and in turn sm_i^1 contains at least the pair (i, v_i)).
  The closure property that allows q_i to exit the loop is the following: for any pair $(k, v_k) \in (i, sm_i^1)$ we have $(k, view_k) \in sm_i^3$. This means that, from q_i's point of view, each simulator process q_k that has written into $WS[1]$ has also written into $WS[2]$ (more precisely, it has written the view sm_k^1 that it has obtained from $WS[1]$). (Let us observe that, interestingly, sm_i^1 is used by q_i as a survivor set.)
- The simulator process q_i considers then the value of sm_i^3 that allowed it to exit the loop (line 06) and selects from it the view that has the smallest size, i.e., the view $view_k$ such that $\forall (x, view_x) \in sm_i^3$ we have $|view_k| \leq |view_x|$ (line 07).
  As we will see in the proof, for any two simulator processes q_i and q_j which exit the loop we have $min\_view_i = min\_view_j$. Consequently, any deterministic rule

[2] Recall that, as in the simulation of Section 6.2, we are assuming that, although writing a write-snapshot object is done only once, it can be read many times. Hence, we assume here an operation $WS[2]$.scan() that read non-atomically $WS[2]$.

(e.g., $\min()$) that extracts a decided value from such a set of pairs (x, v_x) can be used to compute the value dec_i decided by the safe agreement object (lines 08-09).

Theorem 3. *The algorithms described in Figure 6 for the operation* sa_propose() *and in Figure 7 for the operation* sa_decide() *are a correct implementation of the safe agreement object type.*

Proof Let us first recall that it is assumed that a process q_i that invokes sa_decide() has previously invoked sa_propose().

Proof of the termination property. We have to show that, if no simulator process crashes while executing sa_propose(), then any correct simulator that invokes sa_decide() returns from its invocation.

If no process crashes while executing sa_propose(), it follows that any simulator process that executes this operation executes both $WS[1]$.write_snapshot() and $WS[2]$.write_snapshot(). Hence, as any simulator process q_i that executes sa_decide() repeatedly reads $WS[2]$, we eventually have $\forall\ (k, -) \in sm_i^1 : (k, view_k) \in sm_i^3$ (this is because $(k, -) \in sm_i^1$ means that p_k has invoked $WS[1]$.write_snapshot() and as p_k eventually invokes $WS[2]$.write_snapshot(), we eventually have $(k, view_k) \in sm_i^3$). When this happens q_i stops looping. Moreover, as -at least- $(i, sm_i^1) \in sm_i^3$ and $(i, v_i) \in sm_i^1$ it follows that the minimum operations of lines 07 and 09 are well defined and do terminate. Consequently q_i terminates its invocation of sa_decide() which concludes the proof of the termination property.

Proof of the validity property. We have to show that a decided value is a proposed value. Let v be the value decided by a simulator process q_i and $view = min\_view_i$ (computed at line 06). It follows from lines 07-08 that $\exists (x, v) \in min\_view_i = view$ and $(-, view) \in sm_i^3$, from which we conclude that some simulator process q_j has executed $WS[2]$.write_snapshot(sm_j^1) where $sm_j^1 = view$. Hence, q_j has obtained $sm_j^1 = view$ from its invocation $WS[1]$.write_snapshot(). It then follows from the validity property of the write-snapshot object $WS[1]$ and the fact that $(x, v) \in view$, that the value v has been proposed by some simulator process which concludes the proof of the validity property.

Proof of the agreement property. We have to show that no two processes decide different values. Let q_i and q_j be two simulator processes that decide. We show that the sets min_view_i and min_view_j of pairs computed at line 07 are equal (from which follows the agreement property).

Let us first observe that, due to containment property of the write-snapshot object $WS[1]$, any two pairs (x, sm_x^1) and (y, sm_y^1) written in $WS[2]$ (line 02) are such that $sm_x^1 \subseteq sm_y^1 \vee sm_y^1 \subseteq sm_x^1$ (Observation O1). Moreover, we can conclude from the self-inclusion and containment properties of $WS[1]$, that $((x, v_x) \notin sm_y^1) \Rightarrow (sm_y^1 \subsetneq sm_x^1)$ (Observation O2).

Let sm_i^3 and sm_j^3 denote the last values of the corresponding local variables obtained at line 06. As a simulator writes only once in both $WS[1]$ and $WS[2]$, we have $sm_i^3 = \{(k, view_k) \mid (k, -) \in sm_i^1\}$ and $sm_j^3 = \{(k', view_{k'}) \mid (k', -) \in sm_j^1\}$. If

$sm_i^1 = sm_j^1$ we have $sm_i^3 = sm_j^3$ which implies $min\_view_i = min\_view_j$ and agreement follows. Hence, due to observation O1, the remaining case is $sm_i^1 \subsetneq sm_j^1$ or, equivalently, $sm_j^1 \subsetneq sm_i^1$. Without loss of generality let us assume $sm_i^1 \subsetneq sm_j^1$, from which we have $sm_i^3 \subsetneq sm_j^3$. To show $min\_view_i = min\_view_j$ when $sm_i^1 \subsetneq sm_j^1$, we show that $\forall\ (\ell, view_\ell) = (\ell, sm_\ell^1) \in (sm_j^3 \setminus sm_i^3)$ it exists $(\ell', view_{\ell'}) = (\ell', sm_{\ell'}^1) \in sm_j^3$ such that $sm_{\ell'}^1 \subsetneq sm_\ell^1$. Then we will have $|sm_{\ell'}^1| < |sm_\ell^1|$ and, as $sm_i^3 \subsetneq sm_j^3$, it will follow that the smallest view in sm_j^3 is the smallest view in sm_i^3.

To show that there is ℓ' such that $(\ell', sm_{\ell'}^1) \in sm_j^3$ and $sm_{\ell'}^1 \subsetneq sm_\ell^1$, let us consider $\ell' = i$. As $(\ell, sm_\ell^1) \in (sm_j^3 \setminus sm_i^3)$ we have $(\ell, v_\ell) \in sm_j^1 \setminus sm_i^1$, i.e., $(\ell, v_\ell) \notin sm_i^1$. It then follows from Observation O2 that $sm_i^1 \subsetneq sm_\ell^1$ from which we have $|sm_i^1| < |sm_\ell^1|$ and concludes the proof of the agreement property. $\square_{Theorem\ 3}$

9 Conclusion

This paper has presented an introductory survey of recent advances in asynchronous shared memory models where processes can commit unexpected crash failures. To that end the base snapshot model and iterated models have been presented. As far as resilience is concerned, the wait-free model and the adversary model have been discussed. Moreover, the essence of the Borowsky-Gafni's simulation has been described and a new implementation of the safe agreement object type (that lies at the core of the BG simulation) has been presented. It is hoped that this introductory survey will help a larger audience of the distributed computing community to understand the power, subtleties and limits of crash-prone asynchronous shared memory models.

References

1. Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.
2. Afek Y., Weisberger E. and Weisman H., A Completeness Theorem for a Class of Synchronization Objects. *Proc. 12th Int'l ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 159-168, 1993.
3. Attiya H., Rachman O., Atomic Snapshots in $O(n \log n)$ Operations. *SIAM Journal of Computing*, 27(2): 319–340, 1998.
4. Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., The k-Simultaneous Consensus Problem. *Distributed Computing*, 22(3):185-195, 2010.
5. Anderson J., Multi-writer Composite Registers. *Distributed Computing*, 7(4):175-195, 1994.
6. Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, 42(1):121-132, 1995.
7. Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524-548, 1990.
8. Attiya H., Guerraoui R. and Ruppert E., Partial Snapshot Objects. *Proc. 20th ACM Symposium on Parallel Architectures and Algorithms (SPAA'08)*, ACM Press, pp. 336-343, 2008.
9. Attiya H. and Welch J., Distributed Computing: Fundamentals, Simulations and Advanced Topics (2d Edition), *Wiley-Interscience*, 414 pages, 2004.

10. Barbara D. and Garcia Molina H., Mutual Exclusion in Partitioned Distributed Systems. *Distributed Computing*, 1:119-132, 1986.
11. Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-51, 1993.
12. Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-free Computations. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 189-198, 1997.
13. Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG Distributed Simulation Algorithm. *Distributed Computing*, 14(3):127-146, 2001.
14. Charron-Bost B. and Schiper A., The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1), 49–71, 2009.
15. Castañeda A., Rajsbaum S. and Raynal M., The Renaming Problem in Shared Memory Systems: an Introduction. *Computer Science Review*, to appear, 2011.
16. Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.
17. Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
18. Chaudhuri S., More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132-158, 1993.
19. Delporte-Gallet C., Fauconnier H., Guerraoui R. and Tielmann A., The Disagreement Power of an Adversary. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer-Verlag LNCS 5805, pp. 8-21, 2009.
20. Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.
21. Gafni E., The 01-Exclusion Families of Tasks. *Proc. 12th Int'l Conference on Principles of Distributed Systems (OPODIS'08*, Springer Verlag LNCS 5401, pp. 246-258, 2008.
22. Gafni E., The Extended BG Simulation and the Characterization of t-Resiliency. *Proc. 41th ACM Symposium on Theory of Computing (STOC'09)*, ACM Press, pp. 85-92, 2009.
23. Gafni E. and Kuznetsov P., Turning Adversaries into Friends: Simplified, Made Constructive and Extended. *Proc. 14th Int'l Conference on Principles of Distributed Systems (OPODIS'10)*, Springer-Verlag, LNCS 6490, pp. 380-394, 2010.
24. Gafni E. and Kuznetsov P., Relating $\mathcal{L}$-Resilience and Wait-freedom with Hitting Sets. *Proc. 12th Int'l Conference on Distributed Computing and Networking (ICDCN'11)*, Springer-Verlag, LNCS 6522, pp. 191-202, 2011.
25. Gafni E. and Rajsbaum S., Recursion in Distributed Computing. *Proc. 12th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, Springer-Verlag, LNCS 6366, pp. 362-376, 2010.
26. Gafni E. and Rajsbaum S., Distributed Programming with Tasks. *Proc. 14th Int'l Conference on Principles of Distributed Systems (OPODIS'10)*, Springer-Verlag, LNCS 6490, pp. 205-218, 2010.
27. Guerraoui R. and Raynal M., From Unreliable Objects to Reliable Objects: the Case of atomic Registers and Consensus. *9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, Springer Verlag LNCS LNCS 4671, pp. 47-61, 2007.
28. Guerraoui R. and Raynal M., The Alpha of Indulgent Consensus. *The Computer Journal*, 50(1):53-67, 2007.
29. Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
30. Herlihy M.P., Luchangco V. and Moir M., Obstruction-free Synchronization: Double-ended Queues as an Example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, pp. 522-529, 2003.

31. Herlihy M. P. and Rajsbaum S., The Topology of Shared Memory Adversaries. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 105-113, 2010.
32. Herlihy M.P. and Rajsbaum S., Concurrent Computing and Shellable Complexes. *Proc. 24th Int'l Symposium on Distributed Computing (DISC'10)*, Springer Verlag LNCS 6343, pp. 109-123, 2010.
33. Herlihy M.P., Rajsbaum S., and Tuttle, M., Unifying Synchronous and Asynchronous Message-Passing Models. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 133–142, 1998.
34. Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.
35. Herlihy M.P. and Shavit N., The Art of Multiprocessor Programming, *Morgan Kaufman Pub.*, San Francisco (CA), 508 pages, 2008.
36. Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.
37. Imbs D. and Raynal M., Help when Needed, but no More: Efficient Read/Write Partial Snapshot. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer-Verlag LNCS 5805, pp. 142-156, 2009.
38. Imbs D. and Raynal M., Visiting Gafni's Reduction Land: from the BG Simulation to the Extended BG Simulation. *Proc. 11th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'09)*, Springer-Verlag LNCS 5873, pp. 369-383, 2009.
39. Imbs D. and Raynal M., The Multiplicative Power of Consensus Numbers. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 26-35, 2010.
40. Imbs D. and Raynal M., A Liveness Condition for Concurrent Objects: x-Wait-freedom. To appear in *Concurrency and Computation: Practice and experience*, 2011.
41. Imbs D., Raynal M. and Taubenfeld G., On Asymmetric Progress Conditions. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 55-64, 2010.
42. Junqueira F. and Marzullo K., Designing Algorithms for Dependent Process Failures. *Future Directions in Distributed Computing*, Springer-Verlag, LNCS 2584, pp. 24-28, 2003.
43. Lamport. L., On Interprocess Communication, Part 1: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77-101,1986.
44. Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Par. and Distributed Computing: vol. 4 of Advances in Comp. Research*, JAI Press, 4:163-183, 1987.
45. Moir M., Practical Implementation of Non-Blocking Synchronization Primitives. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 219-228, 1997.
46. Moses Y. and Rajsbaum S., A Layered Analysis of Consensus. *SIAM Journal Computing* 31(4): 989-1021, 2002.
47. Mostéfaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Journal of the ACM*, 50(6):922-954, 2003.
48. Neiger G., Set Linearizability. *Brief Announcement, Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, ACM Press, pp. 396, 1994.
49. Rajsbaum S., Iterated Shared Memory Models. *Proc. 9th Latin American Symposium Theoretical Informatics (LATIN'10)*, Springer Verlag LNCS 6034, pp. 407-416, 2010.
50. Rajsbaum S., Raynal M. and Travers C., The Iterated Restricted Immediate Snapshot Model. *Tech Report #1874*, 21 pages, IRISA, Université de Rennes (France), 2007.
51. Rajsbaum S., Raynal M. and Travers C., An Impossibility about Failure Detectors in the Iterated Immediate Snapshot Model. *Information Processing Letters*, 108(3):160-164, 2008.

52. Rajsbaum S., Raynal M. and Travers C., The Iterated Restricted Immediate Snapshot (IRIS) Model. *14th Int'l Computing and Combinatorics Conference (COCOON'08)*, Springer-Verlag LNCS 5092, pp.487-496, 2008.
53. Raynal M., Failure Detectors for Asynchronous Distributed Systems: an Introduction. *Wiley Encyclopedia of Computer Science and Engineering*, Vol. 2, pp. 1181-1191, 2009.
54. Raynal M., Shared Memory Synchronization in Presence of Failures: an Exercise-based Introduction. *IEEE Int'l Conference on Complex, Intelligent and Software Intensive Systems (CISIS'09)*, pp. 9-18, IEEE Press, New York, 2009.
55. Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. *Morgan & Claypool Publishers*, 251 pages, 2010 (ISBN 978-1-60845-293-4).
56. Raynal M., On the Implementation of Concurrent Objects. *Proc. of the Conference dedicated to Brian Randell's 75th Birtday*, Springer Verlag LNCS series, 2011.
57. Taubenfeld G., Synchronization Algorithms and Concurrent Programming. *Pearson Prentice-Hall*, 423 pages ISBN 0-131-97259-6, 2006.
58. Taubenfeld G., The Computational Structure of Progress Conditions. *Proc. 24th Int'l Symposium on Distributed Computing (DISC'10)*, Springer Verlag LNCS 6343, pp. 221-235, 2010.

Fault Tolerant Clock Synchronization for Arbitrary Start-Up Conditions

Natasha Neogi

National Institute for Aerospace

Abstract. The issue of self stabilization in clock synchronization protocols for arbitrary initial conditions is examined in this paper, in the context of the fault tolerant digital clocking system implemented by Daly, Hopkins and McKenna in 1973 at the C. S. Draper Laboratory [1]. We develop formal hybrid input-output automata (HIOA) models of the clocks, communications channels and logical decision-making (voting) protocols housed on each processor; given their implementation description and logical circuit diagrams. We then analyze these models under composition for arbitrary initial conditions to evaluate whether the non-faulty processors all achieve correct and valid clock synchronization. Using a simulation relation between the hybrid dynamical system representation of the hybrid automata and the actual composed HIOA, under restricted Lipschitz conditions for their evolving dynamics, we are able to demonstrate the convergence of the protocol under realistic arbitrary starting conditions.

1 Introduction

Distributed systems consisting of a set of processors that communicate by message transmission and that do not have access to a central clock are prevalent in safety critical applications, such as aircraft, spacecraft, military applications and enterprise systems. These systems must possess stringent guarantees with respect to timing properties and invariant properties related to safety, security and availability. It is oftentimes necessary that these systems possess a common notion of time, be it an integer-valued counter or a real-valued continuum. The ability of these processors to obtain and maintain a common notion of time, within a given bound of error, is traditionally know as clock synchronization.

We wish to prove that, for a given clock synchronization protocol structure and an arbitrary start up state, for a given number of clocks (of which a number are faulty), that the entire system will converge into a synchronous notion of time, modulo a fixed band of error, within a pre-defined bounded period of time. This will require us to address the issues of both convergence of the protocol on each non-faulty node to a common notion of time, as well as closure of the protocol such that all good nodes do not diverge from that common notion of time after achieving convergence, barring any illegal global system state that would necessitate a restart of the entire protocol.

Given the breath and scope of clock synchronization algorithms currently available, we will attempt to generalize the basic components of the problem by specifying the common elements in a formal language. We choose to utilize the Hybrid Input Output Automata (HIOA) formalism for this task. A formal description of the clock and communication channel models is given in the HIOA language, and the fault models utilized

are outlined. An equivalent dynamical systems representation for the clock automaton is derived, as is a graph theoretic model for the communications network. Validity, agreement and termination properties necessary for the convergence of clock synchronization are stated in this context. The decision logic of the Daly, Hopkins and McKenna [1] algorithm is outlined, and a dynamical system representation with Lipschitz ordinary differential equations (ODEs) is constructed in order to prove the convergence of the protocol under realistic arbitrary start-up conditions. The result is placed in context of current work, and future directions are outlined. The next section provides a brief overview of the clock synchronization problem, and its many solutions. After that, the hybrid input/output automata formalism is introduced in Section 3, as are the failure models. Formal HIOA templates of a clock with drift, and a timed indexed channel are provided in Section 4. We then develop a dynamical system of Lipschitz ordinary differential equations in Section 5 to express a clock with drift, and prove that it is equivalent to the clock HIOA in section 4. This enables us to derive sufficient constraints, in Section 7 on a clock synchronization voting algorithm to ensure that the composed system, under stable network assumptions discussed in Section 6, will be self-stabilizing under arbitrary startup, given a set of initial assumptions. We evaluate these conditions in the context of the Daly, Hopkins and McKenna [1] fault tolerant digital clocking system, discuss implications and avenues for future investigation in Section 8.

2 Clock Synchronization Algorithms in Literature

The main goal of a clock synchronization algorithm is to ensure that the clocks of non faulty processors never divert by more than some fixed amount, usually referred to as γ, and is called the *precision* or *agreement condition*. Another condition often imposed is the *accuracy* or *validity condition*, which requires that the clocks stay close to real time, i.e. that the drift of the clocks away from real time be limited. Yet another common goal is that of minimizing the number of messages exchanged during synchronization.

The algorithms in [2],[3],[4],[5],[6], [7] , handle Byzantine (i.e. arbitrary) processor faults, as long as the number of nodes N is greater than three times the number of faulty nodes F i.e. $N > 3F$ (except where noted). They also all require that the processors be initially synchronized and that there be known bounds on the message delays and clock drift. Finally, they all run in rounds, or successive periods of resynchronization activity (necessitated by clock drift).

At every round of the interactive convergence algorithm of [2], each processor obtains a value for each of the other processors' clocks, and sets its clock to the average of those values that are not too different from its own. The closeness of synchronization achieved is about $2N\delta_{latency}$ (where $\delta_{latency}$ is the uncertainty in the message delay). Accuracy is close to that of the underlying hardware clocks (although it is not explicitly discussed). The size of the adjustment is about $(2N + 1)\delta_{latency}$. Reintegration and initialization are not discussed in [2]. The algorithm in [3] also collects clock values at each round, but they are averaged using a fault-tolerant averaging function based on those in [8] to calculate an adjustment. It first throws out the F highest and F lowest values, and then takes the midpoint of the range of the remaining values. Clocks stay synchronized to within about $4\delta_{latency}$. The synchronized clock's rate of drift does not

exceed the drift of the underlying hardware clocks by overmuch. The size of the adjustment at each round is about $5\delta_{latency}$. Superficially this performance looks better than [2]; however in converting between the different models, it may be the case that $\delta_{latency}$ in the [3] model equals $N\delta_{latency}$ in the [2] model. The reason is that in the [2] algorithm a processor can obtain another processor's clock value by sending the other processor a request, and is busy waiting until that processor replies, whereas in the [3] algorithm a processor can receive a clock value from any processor during an interval, necessitating the processor to cycle through polling N queues for incoming messages (this argument is expanded on in [2]). This is an example of the many pitfalls encountered in comparing clock synchronization algorithms. The algorithms of [5] are also based on the interactive convergence algorithm of [2]. At each round, clock values are exchanged. All values that are not close enough to $N - F$ other values (thus are clearly faulty) are discarded, and the remaining values are averaged. However, the performance is analyzed in different terms, with more emphasis on how the clock values are related before and after a single round, so agreement, accuracy, and adjustment size values are not readily available. Reintegration and initialization are not discussed. A useful aspect of this algorithm is that it degrades gracefully if more than a third of the processors fail.

The next set of algorithms (those in [9], [4], and [7]) do not require a fully connected network. Again, every processor communicates with all its neighbors at each round, but since the network is not necessarily fully connected, the message complexity per round could be less than $O(N^2)$. The estimates of agreement, accuracy, and adjustment size presented in the rest of this section for these algorithms are made assuming $N = 3F+1$, and a fully connected network with no link failures, in order to facilitate comparison although, as mentioned above, the algorithms do not require that these conditions hold.

Marzullo and Owicki [10] extended their previous algorithm to handle Byzantine faults without authentication by calculating the new interval in a more complicated, and thus fault-tolerant, manner, and altering the clock rates, in addition to the clock times. Since the algorithm's performance is analyzed probabilistically, assuming various probability distributions for the clock rates over time, it is difficult to compare results with the analyses of the other algorithms, which make worst-case assumptions.

The algorithm of Halpern et al. [4] can tolerate any number of processor and link failures as long as the non faulty processors can still communicate. However, the price paid for this extra fault tolerance is that authentication is needed. When a processor's clock reaches the next in a series of values (decided on in advance), the processor begins the next round by broadcasting that value. If the processor receives a message containing the value not too long before its clock reaches that value, it updates its clock to the value and relays the message. The closeness of synchronization achievable is about $(\delta_{latency} + \rho_{max})$, where ρ_{max} is the maximum drift rate. By sending messages too early, the faulty processors can cause the non-faulty ones to speed up their clocks, and the slope of the synchronized clocks can exceed 1 by an amount that increases as F increases. The size of the adjustment is about $(F + 1)(\delta_{latency} + \rho_{max})$, again depending on F. An algorithm to reintegrate a repaired processor is mentioned; although it is complicated, it has the property of not forcing the processor to wait until the next resynchronization, but instead starting as soon as the processor requests it. However, no overall system initialization is discussed. In the revised version of their paper [11], they

present a simpler reintegration algorithm that joins processors at predetermined fixed times that occur with much greater frequency than the predetermined fixed standard synchronization times.

The algorithm of Srikanth and Toueg [7] is very similar to that of [4], but only handles fewer than $N = 2$ processor failures and does not handle link failures. However, they can relax the necessity of authentication (if $N > 3F$). Agreement, as in [4] is about $(\delta_{latency} + \rho_{max})$. Accuracy is optimal, in that it is provided by the underlying hardware clocks. The size of the adjustment is about $3(\delta_{latency} + \rho_{max})$. There are twice as many messages per round as in [4] when digital signatures are not used. Reintegration is based on the method in [3]. A simple modification to the algorithm gives an elegant algorithm for initially synchronizing the clocks.

Given the breath and scope of clock synchronization algorithms currently available, we will attempt to generalize the basic components of the problem by specifying the common elements in a formal language. We choose to utilize the Hybrid Input Output Automata formalism for this task.

3 Hybrid Input Output Automata (HIOA) Formalism

A HIOA is a formalism for modeling state machines that evolve both discretely and continuously with time [12]. For the ease of presentation we describe the essential concepts in this framework ignoring some of the technical details.

The state of a system is captured by valuations of variables. For a variable v, its type, denoted by $type(v)$, is the set of values that v can take. For a set of variables V, a *valuation* $\mathbf{v}$ is a function that maps each $v \in V$ to a point $type(v)$. The set of all valuations for V is denoted by $val(V)$. A *trajectory* for V models continuous evolution of the values of the variables over a interval of time. A variable is said to be *continuous* if all its trajectories are piece-wise continuous. A *discrete variable* is a special type of continuous variable whose trajectories are piece-wise constant. Typically, continuous variables are used to model physical state such as time, position, velocity, and orientation, while discrete variables are used to model software or program state.

Definition 1 (Hybrid Automaton) *A hybrid automaton $\mathcal{A}$ is a tuple $(V, \Theta, A, \mathcal{D}, \mathcal{T})$ where (a) V is a set of discrete and continuous variables and $val(V)$ is called the* state space. *(b) $\Theta \subseteq val(V)$ is a set of* start states*; (c) A is a set of* actions*, (d) $\mathcal{D} \subseteq val(V) \times A \times val(V)$ is a set of* discrete transitions*. A transition $(\mathbf{v}, a, \mathbf{v}') \in \mathcal{D}$ is written in short as $\mathbf{v} \xrightarrow{a}_{\mathcal{A}} \mathbf{v}'$ and we say that action a is* enabled *at $\mathbf{v}$. (e) $\mathcal{T}$ is set of* trajectories *for V that satisfies certain basic axioms about closure, continuity, and admissibility.*

An execution of $\mathcal{A}$ records the evolution of all the variables along a *particular* run of the modeled system. Formally, its a (possibly infinite) alternating sequence of trajectories and actions $\tau_0 a_1 \tau_1 a_2 \ldots$, such that the first state of $\tau(0)$ is in Θ and for all i in the sequence, $a_i \in A$, $\tau_i \in \mathcal{T}$, and there exists a transition labeled by a_i from the last state of τ_i to the first state of τ_{i+1}. A state $\mathbf{v} \in val(V)$ is *reachable* if it is the last state of some finite execution. We denote the set of executions and reachable states of $\mathcal{A}$ by

$\mathsf{Execs}_{\mathcal{A}}$ and $\mathsf{Reach}_{\mathcal{A}}$. Properties of interest for $\mathcal{A}$ such as safety, stability, progress and timeliness are specified in terms of predicates on $\mathsf{Execs}_{\mathcal{A}}$ and $\mathsf{Reach}_{\mathcal{A}}$. Specifically, a *safety property* is stated as a predicate S of $val(V)$ and $\mathcal{A}$ is said to be safe with respect to S if $\mathsf{Reach}_{\mathcal{A}} \subseteq S$. The notion of stability applies closely to the idea of convergence around a bounded interval for a given set of dynamics, or in this case, differing clocks. Thus, if an idea of convergence of all non-faulty processors to a common notion of real time can be thought of as the convergence of all clocks to a bounded interval about the actual global time, then all good nodes must converge to within a bounded distance of the formal analytical description of the evolution of real time.

3.1 Fault Model Description

In order to self-stabilize and tolerate Byzantine failures, it is essential to assume that eventually the bound on the permanent number of Byzantine failures is strictly less than one third of the number of processors in the network. Formally, that is, after any start/restart state, by some bounded time C there will be at most F Byzantine faults, where $N = 3F + 1$, in the system of N nodes. We consider a component being labeled as faulty if it sends differing incorrect messages to some subset of the receiving parties. Note that this behavior degenerates to a purely Byzantine case, where any message can be sent to any receiver at will; with no correct messages being sent. Clearly, these faults are the most difficult to handle in the context of clock synchronization. In our modeling formalism, this is represented by the sending party i sending a possibly different message (or no message at all) to each receiving party j. This requires additional rounds of communication, as well as a more complex decision making function in order to determine when all non-faulty clocks have converged to within the desired tolerance, enabling the transition to the closure mode of protocol operation.

4 Formal Description of Generic Components Common to Most Clock Synchronization Algorithms

Generically speaking, every clock synchronization protocol will have certain key elements in common. A series of clocks must start up, and then evolve according to given rate. Processors which possess evolving clocks then communicate amongst one another, **exchanging adjusted clock signals**. A protocol for logical decision making, regarding the synchronization of clocks, must be present on each functioning processor. This protocol then evaluates the messages received to decide which clocks fall within the required tolerance to be regarded as synchronous. This framework can be broken down into a set of clock automata, a set of timed channel automata, and a set of uniform (logical) decision making automata. This is a useful abstraction, and will allow for many protocols to be represented in the same general framework. Additionally, as we formally model these elements, we must consider the fashion in which faults will manifest in each component, both at startup and during the evolution of the system. Figure 1 illustrates the major components.

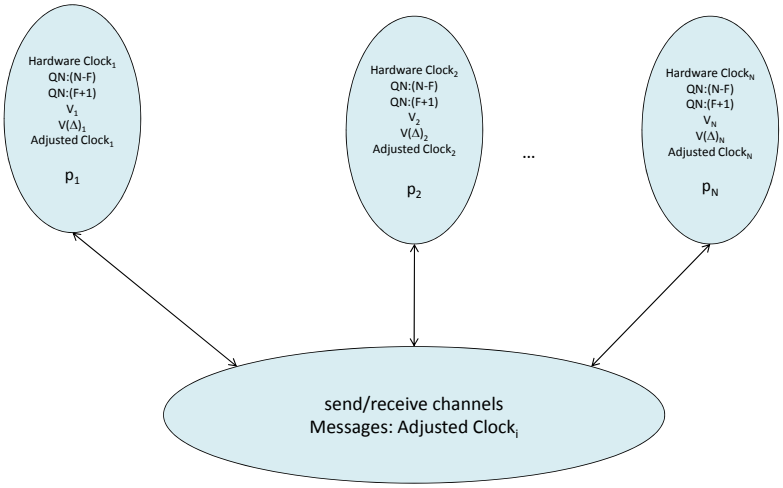

Fig. 1. Components for Clock Synchronization

4.1 Clock Automaton

The clock automaton is a standard physical clock, which can have an arbitrary start-up state, and evolves with a constant bounded drift rate. The clock outputs the time (in an hour-minute format, which is an arbitrary choice). The clock can send messages of its output to all other processors, and does on a fixed periodical schedule. There is the option of having the clock adjusted by an external input (i.e. a voting function). The description is shown in the formal HIOA language in Table 4.1.

4.2 Indexed Channel Automaton

The indexed channel automaton is a an indexed channel (i, j) going from the sending process i to the receiving process j. It can have an arbitrary start-up state, and possesses timing properties. It evolves time forward at a constant rate and is capable of timing out a message that has been in its queue too long (i.e for more that $\delta_{latency}$. It delivers messages in a first in, first out manner, as long as a message has not been timed out (note that dropped messages can be simulated by incorrect timeouts). Its invariant preserves the ordering of messages in this FIFO manner, while respecting latency issues. The description in the formal HIOA language can be seen in Table 4.2.

4.3 Logical Decision Making Automaton

The logical decision making automaton is the main part of the clock synchronization protocol which manipulates the messages its receives from clocks in order to converge to a coarse degree of synchrony. When it receives a message from a clock, it is able to identify the sending clock (due to the indexed channel), and can determine whether the message meets the valid latency conditions associated with that clock. It then uses the time-stamped message received to logically determine the state of the system with respect to clock synchrony. By repeated message passing between clocks and processors, and the application of the logical decision making function (such as a majority voting or quorum function), a correct node, possessing a correct clock and correct voter, is able to

synchronize itself with all other correct nodes, within a bounded time period, dependent on the fault hypothesis. The constraints of the decision logic can be derived such that, when composed with valid clock and channel automata, they will yield a stabilizing clock synchronization solution in the face of arbitrary start-up conditions. The logic of the particular decision making protocol under study in this paper, referred to in short as the Draper Protocol [1] is discussed in further detail in Subsection 7.1.

4.4 Proof Sketch

We wish to compose the clock automata, channel automata and voting protocol automata into a single system. Composing a clock automata $\mathcal{A}$ with a voting automata $\mathcal{B}$ to create the set of processor execution traces $\mathsf{Execs}_{\mathcal{A}\|\mathcal{B}}$ and reachable states $\mathsf{Reach}_{\mathcal{A}\|\mathcal{B}}$ can be considered notionally as follows. The continuous functions describing the execution traces and reachable states of the voting automata are further discretized by the clock transitions. Thus, for every clock tick, an action is inserted into the execution traces of the voting automata. This is as simple as imagining a continuous function $f(x)$, which has finitely many discontinuities, having further discontinuities inserted along the integer axis at each value. The only possible point of confusion can arise when the inserted action for the clock tick arrives at the same time as an action of the actual voting automaton. At this point, there are two possible values for the composed function at that instant, depending on whether we process the voter's action first, and then discretize or vice versa. Without loss of generality, we create a policy for resolving this issue by always processing the clock tick prior to the voter's action. Thus, the voting system will always start the new time interval with the new dynamical equations (if the voter action resulted in a mode switch). The evolution of the execution traces is then governed by the set of differential equations which describes the dynamics of the (clock, voter) system.

If we then compose N of these (clock,voter) processors in the context of a stable communications network, wherein at most F processors are faulty, the set of execution traces for the whole system can be constructed by taking the union of the execution traces over each of the N individual processors. These execution traces can then be described as the union over the differential dynamics governing each (clock, voter) automaton. If this system is described by a set of ordinary differential equations that possess a bounded evolution over time (i.e. they are Lipschitz), then the composed system is guaranteed to possess a unique solution, which implies the convergence of all execution traces. In the next section, we will construct the dynamical system which possess Lipschitz ordinary differential equations that describes the clock automata, and illustrate what conditions will be needed to be imposed on the voter in order to guarantee this unique solution, for our assumed fault and network model.

5 Asynchronous Oscillators as Clocks Under Arbitrary Startup

In order to illustrate the notion of self stabilization of clock synchronization under arbitrary startup, we must first be able to implement the notion of precise timing pulses

in a hybrid automaton, under arbitrary initial reset conditions. This allows for the simulation of powerful differential equations through the composition of the hybrid clock automaton under arbitrary startup, with other hybrid automata with complex governing dynamics (such as dynamic communication channels or voting protocols).

5.1 Clocks and Dynamical Systems

To this end, we use the dynamical system definition of the notion of the hybrid clock automaton as an exact clock [13].

Definition 2 *A dynamical system is defined as $\mathcal{F} = \{X, \Gamma, \phi\}$, where X is an arbitrary topological space (and the state space of $\mathcal{F}$) defined on $\mathbb{R}^n$ (the real space of dimension n), Γ is a topological semigroup, and ϕ is the extended transition map where $\phi : X \times \Gamma \to X$ which satisfies the identity, semigroup and continuity properties.*

Definition 3 *An exact clock is a function S: $\mathbb{R}_+ \to \mathbb{Z}$ and is an exact m-ary clock with pulse width 2Δ, or simply an (m,2Δ)-clock, if:*

1. It is piecewise continuous with finite image Q = $\{0, 1, ..., m-1\}, m \geq 2$.

2. $\forall t \in (2k\Delta, 2(k+1)\Delta)$, $S(t) = i$ if $k \equiv i$ (mod m).

The idea of turning on and off separate systems of differential equations (ODEs) is the key to this definition. We can then implement a $(2, 2\Delta)$-clock (an oscillator of pulse width Δ) with a single ordinary differential equation (ODE), which has dimension $n = 1$.

Theorem 1 *The hybrid automaton $\mathcal{H}$ that implements an exact (2,2Δ)-clock S and allows for the arbitrary resetting of continuous valued parameters on clock edges (i.e. the pasting of an initial condition into a new set of dynamical equations), has an equivalent formulation as a discrete dynamical system $\mathcal{F}$ with continuous ODEs on $\mathbb{R}^{2n}$. Thus the hybrid automaton $\mathcal{H}$ can be simulated by the discrete dynamical system F.*

Proof 1 *By construction.*

Define the set of differential equations that govern the trajectories of H as $G \equiv F(X, \psi, 1)$, that is, the dynamical system with the identity transition map. Both systems are initialized at $t = 0$ with $c = x(0) = x_0$, $x_0 \in domain(G)$.

1. Initialize the function z(t) as $z(0) = x_0$.

Use:

$$\dot{x}(t) = (2\Delta)^{-1}[\tilde{G}(z) - z](1 - S(t)), \tag{1}$$

$$\dot{z}(t) = (2\Delta)^{-1}[\tilde{G}(x - c)]S(t). \tag{2}$$

where $\tilde{G}$ is the continuous extension of $\mathcal{F}$, that is, where every continuous interval ends with the copying of the final conditions into a legal starting state of the next interval (and its governing differential equations). The constant c is set to z when $t = 2k\Delta$, for k odd. So, we have that $x(4k\Delta) = z(4k\Delta) = G^k(x_0)$. Choose $\psi(x, z) = x$ for $x = z$, $x \in domain(G)$.

2. Use:

$$\dot{x}(t) = (2\Delta)^{-1}[\tilde{G}(c) - c](1 - S(t)). \tag{3}$$

The constant c is set to x when $t = 2k\Delta$, for k even. So, we have that $x(4k\Delta) = G^k(x_0)$. Choose $\psi(x, z) = x, x \in domain(G), \forall z$.

□

This theorem allows us to demonstrate the exact equivalence of the formal HIOA clock specification in Table 4.1 with drift rate set to zero, to the constructed dynamical system $\mathcal{F}$. Note that the continuity property of the transition map refers to the fact that ϕ is continuous in both arguments simultaneously, i.e. for any neighborhood W of the point $\phi(g)$, with $g \in \Gamma$, there exist neighborhoods U and V of the point x and the element g respectively such that $\phi(U, V) \subset W$. Note that if $\mathcal{F}$ is not invertible, forward trajectories of the above system of equations may merge. To allow for clocks with drift, we define a **valid inexact** (2,2Δ)-clock as follows.

Definition 4 *A* **valid inexact oscillatory clock** $S_{1,2}(\tau)$ *is described by defining $\dot{\tau}(t) = 1/(2\Delta)$, initialized at $\tau(0) = 0$. Now, define*

$$S_{1,2}(\tau) = h_{\pm}[sin(\pi\tau)], \tag{4}$$

where

$$h_+(l) = \begin{cases} 0, l \leq \eta/2, \\ 2r/\eta - 1, \eta/2 < r \leq \eta, \\ 1, \eta < r, \end{cases} \tag{5}$$

and $h_-(l) = h_+(-l)$, and $0 < \eta < \frac{\sqrt{2}}{2}$.

Note that η and r are parameters which are used to characterize the drift rate ρ. Thus, one switches between two different systems of ODEs with Lipschitz continuous functions of the state of another Lipschitz ODE. This requires $(2n + 1)$ ODEs to simulate a hybrid automaton which has n different sets of governing equations. In this case, there are 3 different sets of governing dynamical equations, which can be switched between depending on the evolution of τ and value of $\rho(\delta, r)$. In order to ensure the boundedness of the differential equations, we impose the notions of **finite gain** and **non degeneracy**. As before, n is the dimension of the topological space X, while N is the number of clocks (and processors). For exact clocks, the composed topological space for all N clocks would be of dimension $n = N$, but for inexact clocks, this becomes $n = 3N$.

Definition 5 (Non-Degeneracy and Finite Gain) *A function $f : \mathbb{R}^n \to \mathbb{R}^n$, is non-degenerate and possesses a finite gain if there exists constants $\beta \geq 0, M \geq 0$, such that*

$$\|x\| \leq M\|f(x)\| + \beta, \forall x \in X \tag{6}$$

$$\|f(x)\| \leq M\|x\| + \beta, \forall x \in X \tag{7}$$

We can now introduce our main result for hybrid automaton composition using inexact oscillatory clocks.

Theorem 2 *Every discrete dynamical system $\mathcal{F}$, which simulates a hybrid automaton $\mathcal{H}$ that is governed by a system of continuous Lipschitz ODEs in $\mathbb{R}^n$, and is defined on $Y \subset \mathbb{Z}^N$ can be: (1) composed with a* **valid inexact oscillatory clock** *automaton $S_{1,2}$, where (2) $\mathcal{F}$ is finite gain and non-degenerate and is governed by a system of continuous Lipschitz ODEs in $\mathbb{R}^{(2n+1)}$, and (3) Y is bounded.*

Note that the theorem is divided into three major conditionals, and its proof is divided thusly.

Proof 2 *By construction. Let $G \equiv \mathcal{F}(X, \psi, 1)$ and $0 < \zeta < 1/3$. $S_{1,2}$ and η are as defined in the description of the inexact oscillatory clock. For each $y \in Y$, define the set:*

$$H_y = \{(x, z, \tau) \mid \|x - y\|_\infty < \zeta, \|z - y\|_\infty < \zeta, \\ \sin(\pi\tau) < \eta/2, \tau mod_2 < 1/2\},$$

for the continuous mod function.

Set $\psi(x, z, \tau) = \Pi(z) = y$ if $(x, z, \tau) \in H_y$. This means that $\psi^{-1}(y) = H_y$ are open and disjoint. Initialize $x(0), z(0), \tau(0)$ in $\psi^{-1}(y), y \in Y$.

1. Choose:

$$\dot{x} = -\zeta^{-2}[x - \tilde{G}(\Pi(z))]^3 S_1(\tau), \tag{8}$$

$$\dot{z} = -\zeta^{-2}[z - (\Pi(z))]^3 S_2(\tau), \tag{9}$$

$$\dot{\tau} = 1. \tag{10}$$

This selection yields $\Pi(z(4k\Delta)) = G^k(y), k \in \mathbb{Z}_+$, for $\mid \tau \mid < \pm\pi^{-1} \sin^{-1}(\eta/2)$.

2. Let α and P be the finite gain constants of G, and β and M be the non-degeneracy constants of G under the infinity norm. Choose

$$\dot{x} = -2\zeta^{-1}[x - \tilde{G}(\Pi(z))]S_1(\tau), \tag{11}$$

$$\dot{z} = -2\zeta^{-1}[z - (\Pi(z))]_2^S(\tau), \tag{12}$$

$$\dot{\tau} = 1/[1 + (P + 1)\|z\|_\infty + \alpha + (M + 1)\|x\|_\infty + \beta]. \tag{13}$$

This selection yields $\Pi(z(t)) = G^k(y)$ on an interval about the time t_k where $\tau(t_k) = 2k, k \in \mathbb{Z}_+$.

3. Let $\beta = \max\{\|i - j\|\}_\infty \mid i, j \in Y\}$. Choose

$$\dot{x} = -2\beta\zeta^{-1}[x - \tilde{G}(\Pi(z))]S_1(\tau), \tag{14}$$

$$\dot{z} = -2\beta\zeta^{-1}[z - (\Pi(z))]S_2(\tau), \tag{15}$$

$$\dot{\tau} = 1. \tag{16}$$

This selection yields $\Pi(z(4k\Delta)) = G^k(y), k \in \mathbb{Z}_+$, for $\mid \tau \mid < \pm\pi^{-1} \sin^{-1}(\eta/2)$.

□

Note that this rather complicated mathematical exercise is merely a proof that the pasting policy described in Subsection 4.4 will yield a set of Lipzchitz ODEs for the composed system of the inexact clock and any dynamical system $\mathcal{F}$ that can be expressed as a set of Lipschitz ODEs. The non-degeneracy and finite gain of the continuous extension of G do not need to hold for points not in Y. There is a bounded neighborhood of initial conditions that will lead to a bounded reachable set of behaviors. The import of condition 2 is that if $G \equiv \mathcal{F}(.,1)$ is non degenerate and may be extended to a Lipschitz function, then the ODEs that govern both the dynamics of the hybrid automaton $\mathcal{H}$ as well as its discrete transitions are Lipschitz. That is, the ODEs that switch between the vector fields described by each mode governed by continuous Lipschitz ODE dynamics, are also Lipschitz [13]. So, G is notionally defined as a switching map that determines the change in dynamics from each set of ordinary differential equations that characterize the dynamics of $\mathcal{F}$. For instance, since an inexact clock has three different sets of dynamical equations, G will govern the switching between them, as the system can only evolve according to one set at a time. Note that this holds for any dynamical system defined on $Y \subset \mathbb{R}^n$ such that there is some minimum separation between any two distinct points of Y, where Y is the domain on which the dynamical system is defined.

This powerful result allows us to compose N diverse inexact oscillatory clocks each with copies of the same hybrid automaton described by a set of $2n+1$ Lipschitz ODEs, and define the global dynamics of the system using $(2n+1)^{N(N-1)}$ Lipschitz ODEs. While this is exponential in the number of clocks, if the system starts within the bounded set of acceptable initial conditions defined in Y, then the system can be guaranteed to be stable. That is, if the composed decision-making protocol and channel automata are described by Lipschitz ODEs, the composed system will be stable, under bounded start conditions. So, it only remains to demonstrate that a given voting protocol can be represented by a set of Lipschitz ODEs (for a stable connected network configuration) in order to assure that the composed system converges to synchrony. Obviously, the stability of the hybrid automaton decision making protocol is impacted by the communication between clocks and decision making protocols. The network topology and each communication channel must also be stable in composition with the inexact clocks and decision making protocol.

6 Network Stabilization: Channels and System Reset

If a bound on the physical channel is known, a finite state self stabilizing protocol may be feasible [14]. We assume that we are given indexed input/output communication channels that can store at most one outstanding message in the event of a system wide communication reset. If we consider the wires connecting the components as channels, this is a reasonable approximation of the behaviour. We assume a message can remain in a channel for at most a duration of $\delta_{latency}$, which is much smaller than the network stabilization time P. We assume that at startup or reset, the network can experience Byzantine behavior, but that after a predetermined period of time, all faulty behavior can be predicted (until the advent of reset). For $3F+1$ nodes, F of which may be

faulty, there must be at least $2F(2F+1)$ correctly operating indexed channels in order to tolerate Byzantine behavior in the F faulty nodes.

The strategy of many protocols (including the Draper protocol) is to evaluate each message in the channel to see if it is a valid value, and if this is not the case, the value is re-written or $reset$ to a valid value by the processor. For a non-faulty channel, we assume that after a time period P, the channel stabilizes, that is, for $e = (i, j)$ the message m sent from node i to j is delivered. The behavior of the channel is regarded as correct when it exhibits the following characteristics.

Definition 6 (Correctness) *Using regular expression operators, the behavior of a correct channel must be able to be broken into segments of the form*

$$\alpha = empty^* send(m, i, j) receive(m, i, j) empty^* \tag{17}$$

within a finite execution time bounded above by $\delta_{latency} \ll P$ for $e = (i, j)$.

We assume all correct channels will exhibit this behavior after a predefined time P from startup/reset. We wish to guarantee that, after all topological changes cease, if one of the valid nodes attains an invalid state, and no valid node attains an invalid state infinitely often, then we can assure the notions of **validity**, and **termination** for a given node running a clock synchronization protocol.

Corollary 1 (Validity). *If $e = (i, j)$ is a valid indexed channel in the final topology, then the execution sequence α of Send(m, i, j) input at i after time $C + P$ is identical to the execution sequence $\tilde{\alpha}$ of Receive(m, i, j) output at j.*

Corollary 2 (Termination). *All valid nodes will terminate a correct clock synchronization protocol if: 1. In finite time all of the valid nodes with valid channels will receive all messages from every other valid node. 2. No valid node receives infinitely many invalid messages.*

6.1 Startup and Reset

Given the perspective of a non faulty processor during clock synchronization, we assume that it is able to communication with all other valid inexact clocks and all other non faulty processors through a correct channel (from definitions 1-2).

Theorem 3 *Assume that there is a set S of $(2F + 1)$* **valid** *and* **terminating** *decision making protocol hybrid automata in the network of at least $3F + 1$ nodes. Each valid indexed channel is a link whose* **correctness** *is checked at minimum consistent intervals P, and each decision making protocol hybrid automaton has stabilization time C. The projected system state $[s]_e, \forall i, j \in S$ from time $\max(C, P)$ onwards, is correct unless the network topology is reset.*

Proof 3 *By contradiction. Suppose not. This implies that there exists a correct link between two valid nodes (i, j), which possesses an execution trace not of the form*

$$\alpha = empty^* send(m, i, j) receive(m, i, j) empty^* \tag{18}$$

over an interval P for $e = (i, j)$. This can occur in one of two ways.

Case 1: A valid message m, sent from the valid node $i \in S$ was not received at a valid node $j \in S$ after the interval P. This implies that the message is still in a valid channel. However, since messages can dwell for at most $\delta_{latency}$ in valid given channel, and $\delta_{latency} \ll P$, we have a contradiction.

Case 2: A valid message m, received by the valid node $j \in S$ was not sent from a valid node $i \in S$ within the prior interval P. This implies that either the valid message m did not originate from any valid node $i \in S$, in which case we have a contradiction, or the valid message m was in the channel for a period longer than P, which contradicts the assumption that all valid messages are delivered by the maximum latency time $\delta_{latency}$.

□

Thus, once the network topology has stabilized, and each valid inexact clock is composed with a logical decision making hybrid automaton, the condition remaining to ensure stabilization in the face of arbitrary startup condition reduces to demonstrating that the hybrid automaton representing the decision making protocol can be represented by a discrete dynamical system of Lipschitz ODEs.

7 Decision Making Protocol

The difficulty arises in the determination of the initial conditions used during the first iteration of the decision making protocol. Without sufficient boundedness arguments, it becomes impossible to limit the reachable set of states for the system. As long as the hybrid automaton $\mathcal{H}$ containing the decision protocol for synchronization, associated with a given clock, receives a value picked from a bounded initial set, then the hybrid automaton will propagate according to its logical execution. For an arbitrary start up value of that clock, or the channel that communicates that clock value to the decision making automaton, it becomes necessary to replace arbitrary initial values of the initial states for the clocks with an acceptable value (i.e. a 0 or a 1). This requires the introduction of a device, which can select the requisite value to replace an unacceptable initial clock value in the decision protocol. Unfortunately, an undefined input can be interpreted under disparate fashions by each protocol, that is, some may interpret it as high (1), some may interpret it as low (0), or some may interpret it as undefined.

Each valid inexact clock evolves according to a constant rate in addition to a bounded drift rate of ρ, with respect to actual time. Before convergence is achieved, the system may behave arbitrarily, as it can start up from an unknown state. However, after a bounded time $\max(C + P)$, there can be at most $(N - 1)/3$ permanent faults in the system. Furthermore, it is assumed that the logical execution of the implemented decision making protocols cannot be affected by faults from other nodes, for all non faulty nodes. Synchronization is defined as being the state in which all good nodes have their clock automata within a pre-defined tolerance, γ. We define a non-faulty node:

Definition 7 (Non-Faulty Node) *Given two hybrid automata defined as the inexact logical clock $\mathcal{A} = S_{1,2}$ and a logical decision making automaton $\mathcal{B} = (V', \Theta', A, \mathcal{D}', \mathcal{T}')$,*

their composition $\mathcal{A}\|\mathcal{B} = (V, \Theta, A, \mathcal{D}, \mathcal{T})$ *is regarded as being non-faulty at times when it displays the following properties: (a) The clock automaton* $S_{1,2}$ *obeys a global bound on the drift rate* ρ*, defined as* $0 \leq \rho \leq 1$*, such that for every closed continuous time interval* $[t_1, t_2]$*,* $(1-\rho)(t_2 - t_1) \leq val(V(t_2)) - val(V(t_1)) \leq (1+\rho)(t_2 - t_1)$ *(b) The logical decision making automaton* $\mathcal{B}$ *executes correctly. (c) The node must process all messages received from all other non-faulty nodes within a bounded time* $t_{processing}$*.*

The composed system $\mathcal{A}\|\mathcal{B}$ is considered faulty if it violates any of the above conditions. The automata $\mathcal{A}$ and $\mathcal{B}$ are disjoint in their internal actions, and cannot block each other's progress. For a fuller description of the mathematical details, we refer the reader to [12]. Note that a faulty node may recover and begin behaving correctly at any time; however, it will not be regarded as being correct until at least $\max(C, P)$ time has elapsed and the necessary number of communications rounds for that node have been established as being passed by another correct node. Furthermore, a faulty node which begins behaving correctly can be considered correct by other correct nodes only after it has demonstrated that it is in synchrony with all other correct nodes. That is, it must have undergone the process of a node that, from an arbitrary start up state, manages to synchronize with all other correct nodes.

Definition 8 (Convergence) *The system is in a synchronized clock state at the real time* t *if for all correct nodes* $(\mathcal{A}\|\mathcal{B})_i$*:* $\mid val(\mathcal{A}\|\mathcal{B})_i(t) - val(\mathcal{A}\|\mathcal{B})_j(t) \mid \leq \gamma$*.*

7.1 Fault Tolerant Digital Clocking System (Draper Protocol)

The fault tolerant digital clocking system implemented in [1] assumes that a central array of $N = 3F + 1$ clock elements interact with one another to produce $N = 3F + 1$ mutually synchronized adjusted processor clock signals, AC_i, for each valid i^{th} node. Each of these valid clock signals maintained at each processor can be synchronized with all other processors except those clock signals that have failed. Failures during the course of operation are assumed to be arbitrary and variable, with any time dependence whatsoever. While every node can start up in an arbitrary state, the $2F + 1$ nodes must possess valid hardware clocks (whose initial state may be arbitrary, but who behave correctly for all time afterward). The phase relation between them is determined by the maximum differential in the desired frequencies (i.e. from the difference between ω_{max} and ω_{min}), which is specified via hardware implementation, and is fixed. A description of the implemented system, including its voting logic (referred to as the Draper Protocol) follows.

Each of the N processors of the central array receives the $N-1$ other adjusted clock signals (ACs) for synchronization purposes. Intrinsic synchronization and valid clock signal generation at the i^{th} node are achieved as follows. Once the adjusted clock AC_i has changed state, two time delay mechanisms are triggered: (a) No further change of state can occur before at least some minimum time t_{min} has elapsed and, (b) If a change of state has not occurred by some maximum time t_{max}, then the state is changed at that time irrespective of the other elements. Note that if $F + 1$ other processors' adjusted

clock signals change state (after t_{min} and before t_{max}), then the current processor also changes the state of its adjusted clock signal.

Thus, the logical decision making protocol on the processor changes state in the following fashion: (a) The output of the protocol V_i is set to 1 when all but F of the N clock elements are 1. (b) The output of the protocol V_i is set to 0 when all but F of the N clock elements are 0. (c) Otherwise, the protocol output V_i retains its previous value, until t_max time has elapsed, and a state change is forced. This protocol output V_i is delayed by the desired half period ($\Delta = 1/2\pi\omega$) to form the signal $V_i(\Delta)$, which serves as the local clock signal on the processor, AC_i, and is synchronized with all other non-faulty processors. **Thus, the processors exchange as messages the value AC_i, which is (upon network stabilization) a delayed version of the voter output V_i.** This procedure can be encapsulated using a quorum function, defined over N elements Q_i^N, such that Q_i^N has a value of 1 if i or more inputs are 1, and otherwise has the value 0. The quorum function *evaluates based on the exchanged values AC_i.* V_i **is set to 1 if Q_{N-F}^N at the processor changes from 0 to 1;** V_i **is set to 0 if Q_{F+1}^N changes from 1 to 0**.

Sample Execution If we consider the case where there are $N = 4$ nodes, of which $F = 1$ are Byzantine faulty, we can define the relevant quorum functions Q_3^4 and Q_2^4. If we consider the condition where a valid node j has its own clock reading 1, and has received messages from the three other clocks of $(1, 1, 0.4)$ (recall one is Byzantine faulty), we evaluate $Q_3^4 = 1$ and $Q_2^4 = 1$. Thus, its own clock output retains the value $V_j = 1$. Now, after some time $t_1 < t_{max}$ has passed, one of the other valid clock outputs switch to zero, and the message set becomes $(1, 0, 0.4)$, with $Q_3^4 = 0$ and $Q_2^4 = 1$. The node maintains its output $V_j = 1$. After some further time another valid clock switches its output, and the message set becomes $(0, 0, 0.4)$ with $Q_3^4 = 0$ and $Q_2^4 = 0$. The falling edge of Q_2^4 triggers the switch to low, so $V_j = 0$. After the minimum settling time has passed, and the message sets $(1, 0, 0.4)$ (leading edge of $Q_2^4 = 1$) and $(1, 1, 0.4)$ (leading edge of $Q_3^4 = 1$) have been received, the clock output returns to high on the leading edge of Q_3^4, with $V_j = 1$ (see Figure 7.1). Note that the failed clock has been modeled as being "stuck at" a particular value. The value of the failed clock can be dynamic and influence the quorum functions; however, the set to high on the rising edge of Q_3^4 (and the set to low on the falling edge of Q_2^4) require that *at least one valid clock has changed its state AC_i to high (or low)*, respectively.

A valid adjusted clock signal is then formed at each processor via the described hysteresis voting protocol. A sketch (for brevity) of the hybrid input output automata used to model the protocol can be seen in Table 7.1. We are primarily concerned with the behavior of the decision making logic when at least one inexact valid clock starts with an arbitrary value (and all state variables are arbitrary at start-up).

Consider the initial conditions such that each good clock component starts with a valid state, except for one, labeled k, which starts from an arbitrary state, that is $V_k = a$. That is, each good i^{th} clock excepting V_k starts at 0 or 1, and each i^{th} voter is initialized to an arbitrary value V_i. As described above, the signal V_i is delayed by Δ_i, which is the desired half period for the signal V_i. Then, we have that the clock output AC_i will go to 1 on the leading edge of Q_{F+1}^N or the falling edge of V_i (modulo the delay Δ_i). Similarly, the AC_i goes low on the falling edge of Q_{N-F}^N or the leading edge of V_i.

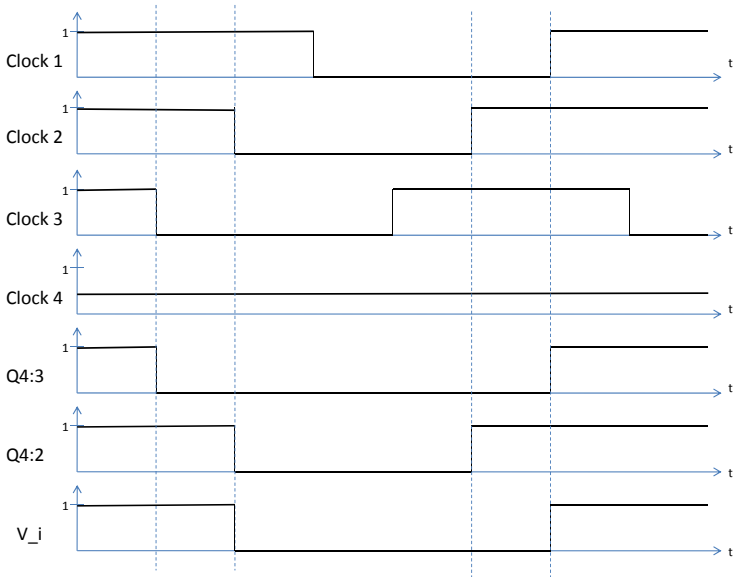

Fig. 2. Example Execution Trace of the Draper Protocol with N=4, F=1

In order to ensure that all valid adjusted clocks AC_i eventually change their state due to a change in a quorum function (and not just due to timeout parameters), which is based on the change in at least one other valid adjusted clock , each valid processor must be able to *arbitrarily distinguish and assign the value of the clock V_k into a leading or falling edge*. With the introduction of a threshold function:

$$val(AC_k) = \begin{cases} 1, AC_k < Threshold, \\ 0, otherwise \end{cases} \tag{19}$$

for each processor, it is possible that this condition is probabilistically achieved. The threshold function can be implemented in hardware as a Schmitt trigger. Thus for any number of arbitrary non-valid initial clock states, which eventually after C become valid, each successive non-valid initial clock message received by a non faulty processor is arbitrarily assigned a valid message value of 0 or 1. Thus, after a time $\max(C, P)$ and at least $2F + 1$ valid messages from distinct processors, will have a state change of the adjusted clock AC_i via a change in a quorum function. This boundedness condition on the AC_i input messages to a valid processor guarantees that the values of the quorum functions and voter outputs $(V_i, V_i(\Delta), AC_i)$ are always bounded after receiving at most $3F + 1$ distinct processor messages.

Theorem 4 *A node $S_{1,2} \| \mathcal{B}$ composed of a clock automaton $S_{1,2}$ and a logical decision making automaton $\mathcal{B} = (V', \Theta', A, \mathcal{D}', \mathcal{T}')$ is self-stabilizing under arbitrary startup*

conditions in the context of a global system S of $N = 3F + 1$ such automata (of which at most F are Byzantine faulty) connected by correct indexed channels (whose topology is described by $[s]_e, \forall i, j \in S$) if the hybrid decision making automaton $\mathcal{B}$ can be represented by a discrete dynamical system $\mathcal{F}$ which is Lipschitz.

Proof 4 *By construction. For all non-faulty processors in the set $I \subset S$ to eventually converge to clock synchronization under arbitrary state start-up conditions, the values for $\mid val(\mathcal{B}_i)(t) - val(\mathcal{B}_j)(t) \mid \leq \gamma, \forall i, j \in I$.*

If the quorum function (modulo a delay Δ, and the maximum and minimum dwell times), determines the output value of V_i, then a bound on the quorum function would naturally result in a bound on the output V_i. Given that all invalid values V_j sent from other clocks and processors can be resolved into a 0 or 1 (in a probabilistic fashion), the only unbounded values occur in the processor's own initial conditions. Namely, the initial values for the processor's quorum functions, as well as its own initial value for V_i. As the quorum functions:

$$Q^N_{N-F} = \bigvee_{i=N-F}^{N-F} (Q^{N-F}_i Q^F_{N-F-i}) \tag{20}$$

triggers a leading edge and

$$Q^N_{F+1}(x, y) = \bigvee_{i=1}^{F+1} (Q^{N-F}_i Q^F_{F+1-i}) \tag{21}$$

triggers a falling edge, when they each change value to one and zero, respectively, they however can be initialized to an arbitrary value at startup. At least one of the quorum functions will be forced to change into a valid value once $2F + 1$ distinct processors have sent at least three distinct messages each in time $t_{min} < \Delta t < t_{max}$. If this has not occurred by the time t_{max}, the processor will change its own arbitrary initial value of V_i to either one or zero (again, in a probabilistic fashion), and broadcast this new, valid value of V_i to all other processors. The processor must again wait for at most $2F + 1$ distinct processors to have sent at least three distinct messages each in time $t_{min} < \Delta t < t_{max}$ before a state value (quorum function) is changed, forcing a change in the value V_i, and so forth.

Thus, the rate of change of the value of V_i is strictly bounded from above by $(t_{min})^{-1}$ and from below by $(t_{max})^{-1}$ after the first valid value of V_i is generated. Since for a non-faulty processor, after a time $\max(C, P) + t_{max}$ has elapsed, the first valid value of V_i must be generated, the only difficulty occurs if the value of V_i is initially unbounded before this switch (i.e. possibly infinite). If we eliminate this case from the set of permissible initial conditions, it is always possible to choose a bound B such that the value of $\mid V_i(h) - V_i(0) \mid \leq B \mid h \mid$.

Note that if the value was undefined, then the derivative at this point would simply not exist, creating no difficulties (as the derivative is not required to exist everywhere). Thus, the dynamical system $\mathcal{F}$ governing the decision making logic for the Draper protocol is Lipschitz. If we define $val(B_i) = V_i$, then for all non-faulty processors after time $\max(C, P) + 3\Delta$, $\mid val(\mathcal{B}_i)(t) - val(\mathcal{B}_j)(t) \mid = \mid V_i(t) - V_j(t) \mid \leq 2 *$

$(t_{max})^{-1}, \forall i, j \in I$. *However, at this point, all values of the quorum function and* V_i *are now valid for all non-faulty processors.*

□

7.2 Precision

Once all valid clock elements of all non-faulty processors have each received at least $(2F + 1)$ valid messages (and $t > \max(C, P)$), they must become phased locked in the following manner. Without loss of generality, we assume that the clocks phase lock based on an initial leading edge (value 1). Since the **first** leading edge of the **first** valid clock output must be triggered by the falling edge of its own corresponding V_i, the valid clock output leading edge must follow the falling edge of its V_i by at least $\delta_{min} = 1/(2\pi\omega_{max})$, the smallest of the delays of the valid clock elements (clock with the highest frequency), and by at most $\delta_{max} = 1/(2\pi\omega_{min})$, the largest of the clock delays (clock with lowest frequency).

Theorem 5 (Precision) . *After the stabilization time of* $\max(C, P)$*, where there are* N *nodes, at most* F *of which possess Byzantine faults, the* $(N - F)$ *valid clocks will phase lock to within a precision* $\gamma = 3\delta_{max} + 3\delta_{latency} + (F + 3)t_{processing}$.

Proof 5 *By Construction. The leading edge of* V_i *in a valid clock element cannot precede the leading edge of any other quorum function* Q^N_{F+1} *in another valid clock element* k *by more than* $\delta_{max} + \delta_{latency} + (F+3)t_{processing}$*. That is, if the leading edge of a valid clock output* V_k *is triggered by the leading edge of its own* Q^N_{F+1}*, then at least one other valid clock output (say* V_i*) must have gone high previously. The maximum amount of time it would take for the* k^{th} *node to sense this would be the sum of the delay at the clock output of* V_i *(which is* $\delta_i \leq \delta_{max}$*), along with the message latency time and the time to process the message.*

The leading edge of Q^N_{N-F} *cannot follow the leading edge of a valid* V_i*, in any valid clock element, by more than* $(F+1)t_{processing} + \delta_{latency} + \delta_{max}$*. Supposing that* V_k *is the last valid clock to possess* $AC_k = 0$*, and the the second last valid clock* V_i *has the lowest frequency (and longest delay* δ_{max}*);* V_i *will have to change from low to high by* δ_{max} *in order to remain valid, and then send its message, bounding the maximum time that all valid clock messages can take to change from high to low. That is, previously, to set* $AC_k = 0$ *you had at least* $N - F$ *low outputs, and now you must have at least* $N - F$ *high outputs in order for it to be set high. Thus, at least* $F + 1$ *outputs must have changed from low to high. The valid node would have to process at least* $F + 1$ *messages, and evaluate the quorum function* $F + 1$ *times. Thus, the quorum function of the* k^th *node must change after it has received and processed* $F + 1$ *high messages, the last of which can be received* $\delta_{max} + \delta_{latency}$ *after the second last valid clock output* V_i *has changed from low to high.*

The leading edge of V_i *in a valid clock element cannot follow the leading edge of* Q^N_{N-F} *of any other valid clock element, by more than the fixed propagation delay of* $\delta_{max} + \delta_{latency} + t_{processing}$*. Consider the first valid processor* k *to have its quorum function* Q^N_{N-F} *to evaluate to high. This means that the* k^th *node must have received at least* $N - F$ *high messages, of which* $F + 1$ *came from valid elements. The* i^{th}

valid processor must also have received these valid messages, as have all other valid processors. All valid processors which are low must go high within the period bounded by δ_{max}, $F + 1$ of which have already done so. After a maximal possible delay of $\delta_{max} + \delta_{latency} + t_{processing}$, the last valid processor receives the message from the second last processor to go high (i.e. the $j^{th} processor$) $AC_j = 1$ and re-evaluates the quorum function.

Thus, the leading edge of V_k of one valid clock element must follow the leading edge V_j of any other valid clock element by at at most the fixed propagation delay $3\delta_{max} + 3\delta_{latency} + (F + 3)t_{processing}$. Note that if the V_i of valid processors are phase locked, so are the $V_i(\delta_i)$ and AC_i.

□

8 Discussion and Conclusions

In the context of classical clock synchronization, it is known that if all valid clocks are running at a rate bounded by some linear function of real time (from both above and below), then clock synchronization is impossible with one third or more clocks being faulty, without authentication. Dolev et al. [15] have shown that if there is a finite upper bound on the rate at which messages can be generated by a processor, then clock synchronization is achievable without authentication (and assuming the absence of network partitioning). The pulse-based self-stabilizing algorithm of Daliot, Dolev and Parnas [16] assumes an underlying notion of coarse synchrony at startup, by specifying an upper bound on the real-time between the invocation of the pulses in correct nodes. Recently, they have developed fast, self-stabilizing Byzantine clock synchronization protocols [17, 18], which obtain an optimal probabilistic solution, though time is modeled using integer abstraction. There has been a large quantity of work on self-stabilizing clock synchronization in the past decade [19],[20],[21],[22],[23], much of which uses an integer descretization of time and considers diverse network topologies [24], in contrast to the continuous approach adopted in this paper.

There are several hardware approaches taken to this problem that deal with clock tick generation [25] and arbitrary initialization under failures [26]. However, this hardware is more complex than the circuit diagrams outlined in [1], and a more detailed analysis for comparison will be performed in the future. Synchronization incorporating continuous trajectories under a real time computing model is studied using optimal analysis[27], in contrast to a robust setting implied by Byzantine fault tolerance.

In the Draper protocol outlined in this paper, as well as the attendant proof of convergence for arbitrary initial conditions, there are no inherent assumptions of even coarse synchrony upon startup, nor are there any restrictions on initial values. However, the requisite assumptions of finite gain and of Lipschitz ODEs governing the overall dynamical system representation of the composed clock, communication channel and logical decision making hybrid input output automata act to constrain the rate of change of the output. This acts to limit the acceptability of the ∞ value for any of the quorum functions at any non-faulty processor. Furthermore, the digital design of the system posits the existence of a mechanism by which a non-valid initial value of any clock in the system is assigned a value of zero or one at each non-faulty processor. This function

can be probabilistically achieved in the circuit design by the use of a Schmitt Trigger. Thus, for a realistic set of arbitrary initial conditions (excluding ∞ for quorum functions and V_i), all $(2F + 1)$ non-faulty processors converge to attain synchrony with the bound $\gamma = 3\delta_{max} + 3\delta_{latency} + (F + 3)t_{processing}$.

The fundamental issue of self stabilization of clock synchronization under Byzantine faults and arbitrary starting conditions hinges around the notion of boundedness. The theoretical distinction between possibility and impossibility depends upon not only the fault models for the system, but also on boundedness properties related to message delivery, inherent coarse synchrony imposed by a maximum bound on computational time of the protocol execution, as well as message ordering. The use of the Lipschitz condition to bound the behavior of the ODEs governing the composed system of HIOAs imposes a notion of local stability. The idea of metastability in the synchronization of asynchronous signals appears to be, though without formal proof at this time, principally unavoidable. However, for practical purposes, digital circuits can be designed to minimize metastable behavior, through the introduction of delay.

Of course, it should be pointed out that there are several probabilistic approaches to attaining clock synchronization and consensus [28],[29] in the presence of Byzantine faults in a distributed and asynchronous setting (i.e. under finite time with probability approaching 1). Additionally, there are algorithms for approximate agreement in both the clock synchronization and distributed consensus settings [30],[5].

Conclusions. In this paper, we have investigated the notion of self-stabilizing clock synchronization under arbitrary startup conditions, in the context of an implemented digital clocking system fielded at the C. S. Draper Laboratory in 1973 by Daly, Hopkins and McKenna [1]. A proof was developed by deriving a dynamical system which simulated the composed system of clocks, communication channels and logical decision making protocols, which were formally specified as Hybrid Input Output Automata. By ensuring that the ordinary differential equations governing the dynamical system were Lipschitz, convergence of the system is guaranteed.

However, this condition restricts the startup condition of the quorum functions in the logical decision making automata; they are all required to have a finite, bounded rate of change, thereby eliminating the startup value of ∞. However, under realistic startup conditions, and using proper digital circuit design, this condition can be accommodated. Furthermore, this notion of boundedness appears to be an inherent assumption in all of the current self stabilizing clock synchronization protocols currently available. An interesting direction of future work would encompass proving conclusively that this structure of assumptions are necessary for any self-stabilizing clock synchronization algorithm under arbitrary startup in the face of Byzantine faults.

9 Acknowledgments

The author would like to thank Paul Miner at NASA Langley Research Center, for his invaluable discussions and patient care in shepherding the work, especially for providing a lively devil's advocate view throughout the proof derivation process. The author would also like to thank Mahyar Malekpour and Anthony Narkawicz at NASA Langley Research Center for their insightful observations and feedback.

References Cited

1. W. Daly, J. A.L. Hopkins, and J. McKenna, "A fault-tolerant digital clocking system," in *The State of the Art: From Device Testing to Reconfigurable Systems, FTCS 3*, 1973, pp. 17–22.
2. L. Lamport and P. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal Of The ACM*, vol. 32, pp. 52–78, 1985.
3. J. Lundelius and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Information and Computation*, vol. 77, pp. 1–36, 1988.
4. J. Halpern, B. Simons, R. Strong, and D. Dolov, "Fault tolerant clock synchronization," in *Proc. 3rd Ann. ACM Symp. On Principles of Distributed Computing*, 1984, pp. 89–102.
5. S. Mahaney and F. B. Schneider, "Inexact agreement: Accuracy, precision and graceful degradation," in *Proc. 4th Ann. ACM Symp. On Principles of Distributed Computing*. ACM, 1985, pp. 237–249.
6. P. S. Miner, A. Geser, L. Pike, and J. Maddalon, "A unified fault-tolerance protocol," 2004.
7. T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, pp. 626–645, 1987.
8. D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl, "Reaching approximate agreement in the presence of faults," *Journal of the ACM*, vol. 33, pp. 449–516, 1986.
9. K. Marzullo, "Loosely-coupled distributed services: A distributed time service," Ph.D. dissertation, Stanford University, Palo Alto, C, 1983.
10. K. Marzullo and S. Owicki, "Maintaining the time in a distributed system," in *Proceedings of the Second Symposium on Principles of Distributed Computing*. ACM SIGPLAN/SIGOPS, 1983.
11. D. Dolev, J. Halpern, B. Simons, and H. R. Strong, "D. dolev, j. halpern, b. simons, and h. r. strong," *Information and Computation*, vol. 72, pp. 180–198, 1987.
12. N. Lynch, R. Segala, and F. Vaandrager, "Hybrid i/o automata," *Information and Computation*, vol. 185, no. 1, pp. 105 – 157, 2003.
13. M. Branicky, "Studies in hybrid systems: Modeling, analysis, and control," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1995.
14. S. Dolev, A. Israeli, and S. Moran, "Resource bounds for self stabilizing message driven protocols," in *Proceedings of the tenth annual ACM symposium on Principles of Distributed Computing*, 1991, pp. 281–293.
15. D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, pp. 393–402, 1983.
16. A. Daliot, D. Dolev, and H. Parnas, "Linear-time self-stabilizing byzantine clock synchronization," *CoRR*, vol. abs/cs/0608096, 2006.
17. E. N. Hoch, D. Dolev, and A. Daliot, "Self-stabilizing byzantine digital clock synchronization," in *SSS*, 2006, pp. 350–362.
18. M. Ben-Or, D. Dolev, and E. N. Hoch, "Fast self-stabilizing byzantine tolerant digital clock synchronization," in *PODC*, 2008, pp. 385–394.
19. E. N. Hoch, M. Ben-Or, and D. Dolev, "A fault-resistant asynchronous clock function," in *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*, ser. SSS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 19–34. [Online]. Available: http://portal.acm.org/citation.cfm?id=1926829.1926836
20. R. E. L. DeVille and S. Mitra, "Stability of distributed algorithms in the face of incessant faults," in *SSS*, 2009, pp. 224–237.
21. J.-H. Hoepman, A. Larsson, E. M. Schiller, and P. Tsigas, "Secure and self-stabilizing clock synchronization in sensor networks," in *SSS*, 2007, pp. 340–356.

22. T. Herman and C. Zhang, "Best paper: Stabilizing clock synchronization for wireless sensor networks," in *SSS*, 2006, pp. 335–349.
23. M. R. Malekpour, "A byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems," in *SSS*, 2006, pp. 411–427.
24. C. Boulinier, F. Petit, and V. Villain, "When graph theory helps self-stabilization," in *PODC*, 2004, pp. 150–159.
25. M. Fuegger, U. Schmid, G. Fuchs, and G. Kempf, "Fault-tolerant distributed clock generation in vlsi systems-on-chip," *Sixth European Dependable Computing Conference (EDCC-6)*, Oct. 2006.
26. A. Emmanuelle, C. Delporte-Gallet, H. Fauconnier, M. Hurfin, and J. Widder, "Clock synchronization in the byzantine-recovery failure model," in *Proceedings of the 11th international conference on Principles of distributed systems*, ser. OPODIS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 90–104. [Online]. Available: http://portal.acm.org/citation.cfm?id=1782394.1782401
27. H. Moser and U. Schmid, "Reconciling distributed computing models and real-time systems," in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, Rio de Janeiro, Brazil, Dec 2006, (to appear, see [**?**] for an extended version).
28. M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, ser. PODC '83. New York, NY, USA: ACM, 1983, pp. 27–30. [Online]. Available: http://doi.acm.org/10.1145/800221.806707
29. G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM*, vol. 32, pp. 824–840, 1985.
30. M. H. Azadmanesh and R. M. Kieckhafer, "New hybrid fault models for asynchronous approximate agreement," *IEEE Trans. Comput.*, vol. 45, pp. 439–449, April 1996. [Online]. Available: http://portal.acm.org/citation.cfm?id=228410.228421

Appendix: Nomenclature and Definitions

Nomenclature

$\mathcal{A} = (V, \Theta, A, \mathcal{D}, \mathcal{T})$: Hybrid automaton

$\mathsf{Execs}_{\mathcal{A}}$: Execution traces of $\mathcal{A}$

$\mathsf{Reach}_{\mathcal{A}}$: Reachable states of $\mathcal{A}$

C: Stabilization time of a hybrid automaton; the time after startup after which all execution traces are suffixes of correct behaviors.

$\mathcal{F} = [X, \Gamma, \phi]$: Dynamical system (an alternative **equivalent** control theoretic description of $\mathcal{A}$)

$\|x\|_1$: The L^1 norm of a vector, defined as $\sum_{k=1}^{n} |x_k|$.

$\|x\|_\infty$: The L^∞ norm of a vector, defined as $\max_k |x_k|$.

$\|f(x)\|_2$: The L^2 norm of a function defined as $\int |f(x)|^2 dx$.

N: Number of processors or nodes in the system.

F: Number of faulty processors or nodes in the system.

m: Message sent between two processors.

$\mathcal{G}$: A graph $\mathcal{G} = (V, E)$ with vertices $v \in V$, which denote processors, and edges $e \in E$ which denote communication channels.

$e(i, j)$: An edge connecting processor i to processor j. The edge e, without explicit processor ordering, denotes the pair of anti-symmetric communication channels connecting the processors (i, j).

$[s]_e$: The projection of the global system state s of the graph $\mathcal{G}$ onto the subsystem formed by e (and its corresponding processors (i, j)).

P: The time after which the network topology of the graph $\mathcal{G}$ has stabilized after startup, and all non-faulty channels behave correctly.

$\delta_{latency}$: Maximum time for a message to be delivered between non-faulty processors on a valid channel.

γ: Maximum deviation between the leading (or falling) edges of any two synchronous processor clocks.

$t_{processing}$: Time by which a processor must finish processing all received messages.

t_{min}: Minimum dwell time in a given system state.

t_{max}: Maximum dwell time in a given system state.

HC_i: The value of the i^{th} node's hardware clock.

ω_i: The desired frequency of the i^{th} node's hardware clock.

Q_i^N: Quorum function which evaluates to 1 if i or more of N inputs are 1, otherwise evaluates to 0. It can be initialized to any value at start-up. It operates on the passed messages AC_i.

V_i: The output of the voting protocol. In the case of the Draper Protocol, it is conditioned on the relevant quorum functions.

Δ_i: The half period delay induced in the signal V_i before transferring it as the value of the adjusted clock AC_i

AC_i: The value of the i^{th} node's adjusted clock, which is passed between processors and used to arrive at convergence of all valid nodes by a processor's voting protocol.

δ_{max}: Largest half period of all valid adjusted clocks.

δ_{min}: Smallest half period of all valid adjusted clocks.

$\mathbb{R}^n$: The real space of dimension n. For instance, an ordinary differential equation which has n independent variables will have its solution defined on this space.

Definitions

Lipschitz Condition: A function $f(x)$ satisfies the Lipschitz condition of order β at $x = 0$ if $\|f(h) - f(0)\| \leq B\|h\|^\beta$, for some real valued constant B, and integer exponent β.

Continuous function extension $\tilde{f}$: Any continuous function $f : S \to \mathbb{R}^m$, S a closed subset of $\mathbb{R}^n$, may be extended to a continuous map $\tilde{f} : \mathbb{R}^n \to \mathbb{R}^m$.

Clock Automaton

types Index : Enumeration [Voter1, Voter2, Voter3, Voter4]
let legalTime(hour, minute) : Nat,Nat → Bool = minute < 60
∧ 0 < hour ∧ hour < 13;

automaton Clock(r: Real)
signature
output show(hour, minute: Nat) where legalTime(hour, minute)
output send(M: legalTime, j: Index, const i),
input set(hour, minute: Nat) where legalTime(hour, minute)

states
now: Real := 0;
nextHour: Nat := 12;
nextMinute: Nat := 0;
timeToShow: DiscreteReal := 0;
timeToSend: DiscreteReal := 0;
initially (0 ≤ r ∧ r < 1);

transitions
input set(hour, minute)
eff nextHour := hour;
nextMinute := minute;
timeToShow := now;
timeToSend := now;

output show(hour, minute)
pre hour = nextHour ∧ minute = nextMinute ∧ now = timeToShow;
eff nextMinute := mod(minute + 1, 60);
if nextMinute = 0 then nextHour := mod(hour + 1, 12); fi
timeToShow := now + 1;

output send(m: legalTime, j: Index, const i)
pre hour = nextHour ∧ minute = nextMinute ∧ now = timeToSend;
eff nextMinute := mod(minute + 1, 60);
if nextMinute = 0 then nextHour := mod(hour + 1, 12); fi
timeToSend := now + 1;

trajectories
trajdef timePassage
stop when now = timeToSend;
evolve (1 - r) ≤ d(now); d(now) < (1 + r);

Table 1. HIOA: Clock with Periodic Cyclical Counter

Indexed Channel Automaton

vocabulary V(T: Type)
types Packet : Tuple [message: T, deadline: Real]
end

automaton TimedChannel(ttl: Real, M: Type, i, j: Nat) where ttl ≥ 0
imports V(Type M)

signature
input send(m: M, const i, const j)
output receive(m: M, const i, const j)

states
queue: Seq[Packet] := {};
now: Real := 0;
initially ttl ≥ 0;

transitions
input send(m, i, j)
eff queue := queue $\vdash$ [m, now + ttl];

output receive(m, i, j)
pre queue $\neq \{\} \wedge$ head(queue).message = m;
eff queue := tail(queue);

trajectories
trajdef timePassage
stop when queue $\neq \{\} \wedge$ head(queue).deadline = now;
evolve d(now) = 1;
invariant of TimedChannel: $\mathcal{A}k$: Nat $\mathcal{A}l$: Nat ($0 < k \wedge k \leq l \wedge l <$ len(queue)
$\Rightarrow$ queue[k].deadline $\leq$ queue[l].deadline);

Table 2. HIOA: Timed Indexed Channel Automaton

Logical Decision Making Automaton (sketch)
vocabulary Continuous operators continuous: Real $\rightarrow$ Bool types
ClockIndex : Enumeration [c1, c2, c3, c4];
ProcessIndex : Enumeration [p1, p2, p3, p4];
automaton Voter(i:ProcessIndex, j:ClockIndex, tmin, r,e : Real)

imports Continuous
signature
input receive(Clk, i:ClockIndex, const j)
output send(V_i, const i , j:ProcessIndex)
internal sig_i sig_0
states
Q_i : Real :=0;
Clk,: Real := 0;
t, ti, tdelay: Real :=0;
transitions
input receive (Clk, i, j)
eff Use logic to determine which ith clock
send message and if message is valid
If valid send an internal signal, sig_i, fi

internal sig_i
output sig_0
pre Check if the clock message is in accordance
with all of the timing assumptions
in the utilized Quorum Functions Q_i^N,
to arrive at a new value for the clock time V_i.
eff Compute new value of V_i based on its previous value,
and the value of all valid messages received
since the last computation was made.

internal sig_0
output send(V_i, i, j)
pre the minimum allowable time since the last sent
message to that process has elapsed
eff reset the timing counter for that process

output send(V_i,i, j)
pre delay between last sent message
and max allowable time between messages times out
eff $t_{counter}$:= 0;

trajectories
trajdef Traj
invariant after a bounded time T_Q all Q_i^N are 1
$\wedge$ within the time T_Q + ε all values of Q_i^N become 0
stop when t = t_{final};
evolve
d(now) = r+e
d(tdelay) = 1;

Table 3. HIOA: Sketch of the Hysteresis Voter for 4 Clocks

To Crash or Not To Crash: Efficient Modeling of Fail-Stop Faults

Habib Saissi[†], Péter Bokor[†], Marco Serafini[‡] and Neeraj Suri[†]

[†]Technische Universität Darmstadt, Germany
{pbokor,saissi,suri}@cs.tu-darmstadt.de
[‡]Yahoo! Research, Barcelona, Spain
serafini@yahoo-inc.com

Abstract. A commonly used approach in practical verification is to verify a simplified *model* of the system rather than the system itself, which would entail infeasible verification complexity. This paper introduces a model for efficient model checking of message-passing systems with *crash* faults. The key to the achieved efficiency is the intuition that the event of process crash can be omitted in the model as crashed processes can be mimicked by "slow" ones. We formally prove this intuition for a general class of systems and their specifications.
We evaluate model checking efficiency using two models, one where crash events are modeled as separate state transitions (explicit model) and another where these events are omitted (implicit model). Our experiments with widely-used and representative protocol examples show significant reductions of model checking memory and time when using the implicit model instead of the explicit one.

1 Introduction

Fault-tolerance is a general concept for building dependable systems. It guarantees that the system delivers correct service despite the presence of faults. Usually, the behavior and number of faults is restricted by a fault-model, which is a set of assumptions about the system and its environment. For example, the Paxos protocol [12] delivers consensus (the service) as long as faulty processes fail by crashing. Of course, this concept is valid only if the system that implements fault-tolerance is not faulty itself. For example, faulty implementations of Paxos fail to deliver consensus even if the fault-model is respected [14].

Model checking [10] can be used to automatically verify that fault-tolerance is implemented correctly. As the efficiency of model checking decreases with increasing state space sizes, its applicability is limited to small (sub-)systems or to simplified *models* of the real system. These represent different use-cases of model checking, which can contribute to the correctness of the system in different ways. For example, model checking a faithful model of the system can help fast prototyping and verifying conceptual designs.

In this paper, we propose a model for efficient model checking of *message-passing* systems with *crash* faults. These systems see more and more applications

given that (a) message-passing is an intuitive communication model [3, 1] and (b) the crash fault-model is a widely-applied abstraction in the current practice of reliable systems [7]. We expect every verification method to be *sound*, i.e., it does not miss bugs in the system. In order to ensure that model checking using our proposed model is sound, we compare it with a reference model of crash events, which we adapt from [3]: A crashed process stops receiving and processing messages in the future; if a process crashes during sending messages, only a subset of these messages might be sent. We call this reference model *explicit* because a crash event is modeled via an explicit state transition that drives the system from a state where the process is correct into a state where this process is crashed.

The explicit model yields large state spaces because crash events are interleaved, i.e., executed concurrently, with other events. In order to mitigate this state space explosion, we leverage the intuition that slow processes cannot be distinguished from crashed ones. We carefully investigate models of computation and communication to verify this intuition. For example, the intuition does not hold if a protocol inherently relies on crash events (e.g., using failure detectors [8]) or if the property under verification explicitly mentions crashes.

We call a model without crash events *implicit* because it can mimic the effect of crashes through the above intuition. We formally prove the soundness (and completeness) of the implicit model by showing that the truth of a general class of properties is indistinguishable in the explicit and implicit models.

The explicit model is exponentially larger than the implicit model. This exponential blow can be further worsened in practical model checking, where storing and comparing states (stateful optimization) is hard or even impossible. On the other hand, reductions (such as symmetry or partial order reductions [10]) can be used to prune the state space that actually needs to be explored. As a practical implication of our equivalence result, we model check representative message-passing protocols and measure the realized benefit of using the implicit model instead of the explicit one.

In summary, we make the following specific contributions:

- We define a formal model eligible for model checking systems where processes communicate via messages and might fail by crashing. First, we adopt a model from [3] (Section 2) and use it as a reference model to show the soundness of our proposed, simplified model. We call this reference model explicit because it explicitly models the event of crashing. We define a model, called implicit model, by simply removing crash events from the explicit model (Section 3).
- We formally prove that the implicit model preserves arbitrary LTL (Linear Temporal Logic) properties [10] of the explicit model if these properties do not depend on (i) the crash status of some process and (ii) the set of undelivered messages (Section 4). This class of properties is general and expressive enough to specify standard properties of fault-tolerant message-passing protocols, e.g., consensus, or variants of linearizability.

– We use MP-Basset [5], a SW model checker for message-passing systems, to model check the explicit and implicit models of representative crash-tolerant protocols (Section 5). Using the stateful and partial-order reduced optimizations of MP-Basset, our experiments show that model checking the explicit model results in state space explosion, even with relaxed forms of the crash-fault semantics. At the same time, the implicit model enables feasible model checking of the same protocol instances.

2 A Formal Model of Message-Passing Systems

We start by recalling a formal model of general message-passing systems (Section 2.1) and a suitable property language (Section 2.2). The precise semantics of the formalism is given via state graphs (Section 2.3).

2.1 Basic Message-Passing Model

Conceptually, we adopt the formal model of *message-passing system* (MP system) from [3]. Strictly speaking, the following model is taken from [4, 5], which was shown to be equivalent with the model in [3] but better suited for model checking.

An MP system consists of n *processes* that communicate via *messages*. Messages are sent between processes via *channels* according to a network topology. Every two processes i and j that are connected via a channel from i to j maintain a *buffer* $buf_{i,j}$, which is a set of messages for storing undelivered messages sent from process i to process j. The buffer $buf_{i,j}$ is called the *outgoing* (*incoming*) buffer of process i (j).

Every process i maintains a *local state* from a set Q_i. The *state* of the system is a tuple $s = (q_1, q_2, ..., q_n, b_1, ..., b_m)$, where $q_i \in Q_i$ for all $1 \leq i \leq n$ and $b_1, ..., b_m$ are the buffers of the system.

Transitions between the states of an MP system are modeled through *events*. The *execution* of an event denoted by $comp_i$ (short for computation) involves the following indivisible (atomic) change to the system: a (maybe empty) subset of the messages is removed from the union of all incoming buffers of i, the current local state q_i is changed to a (maybe the same) state from Q_i, and some (maybe zero) messages are added to the output buffers of i. Every event is associated with a *guard*, which is a predicate that depends only on a subset of the union of the incoming buffers and the local state of the process. The event can only be executed if the guard evaluates to true. In this case, we say the event is *enabled* in the current state. Otherwise, the event is *disabled*.

The set of all events is denoted as $COMP = \cup_{i=1}^{n} COMP_i$, where $COMP_i$ is the set of all events executed by i.

An *initial* state of the system is the state before the execution of any event. We assume that channels are empty in initial states.

2.2 Property Language: Temporal Logic

Properties that the system is expected to fulfill are interpreted over *runs*. A run of a message-passing system is a sequence of states $s_0, s_1, ...$ such that s_0 is an initial state and, for $i > 0$, the state s_i is the state resulting of the execution of an event $comp_i$ in s_{i-1} such that $comp_i$ is enabled in s_{i-1}. We call s_i a *reachable* state. By convention, initial states are also reachable.

The most simple properties specify single states. This requires the definition of a *labeling function*, which assigns *atomic propositions* from a set AP to each state. Formally, the labeling function is defined as $L : S \longrightarrow 2^{AP}$, where S denotes the set of all states. For example, atomic propositions combined with the usual Boolean connectives can be used to define *invariants*, a simple and expressive set of properties. A property is an invariant if it holds in every reachable state.

We adopt Linear Temporal Logic (LTL) [10] to specify *temporal* properties. In addition to atomic propositions and Boolean connectives, LTL defines *temporal operators*. For example, the operator **F** ("eventually" or "future") asserts that a property will hold in a state that is reachable (along a run) from the current state. As an example LTL formula, consider $\mathbf{F}p$. This formula expresses liveness, i.e., some atomic proposition p ("something good") must hold after the execution of an indefinite number of events.

2.3 Kripke Structure : Syntax & Semantics

We use the standard semantics of LTL [10]. As it is based on a *Kripke structure*, we associate MP systems with Kripke structures. A Kripke structure is a tuple (S, S_0, T, AP, L), where S is a set of states, $S_0 \subseteq S$ is a set of initial states, $T \subseteq S \times S$ is a set of transitions, AP is a set of atomic propositions, and L is a labeling function. Given an MP system M with (initial) state set S (S_0), and atomic propositions AP, and labeling function L, we associate with M the Kripke structure $M_{KS} = (S, S_0, T, AP, L)$, where $(s, s') \in T$ iff there is an event *comp* of the MP system such that *comp* is enabled in s and executing *comp* in s results in s'.

As a result, a run of the MP system M is a run (also called path [10]) of the Kripke structure M_{KS} and the standard semantics of LTL specifications can be applied. As this semantics assumes infinite runs, we define an additional event, called *dummy* event and denoted *dum*. The dummy event is enabled in every state and its execution does not alter the state of the system. Note that without the dummy event it is possible that no event is enabled in a state, resulting in finite runs.

3 MP Systems with Crash Faults

In this Section, we define MP systems where processes can *crash*. In the crash fault-model, a process can stop receiving, processing, and sending messages, and

it remains doing so forever. If the process crashes during the execution of an event, it executes the event as in the fault-free case except that it sends a subset of the messages that it was supposed to send [3].

Formally, given an MP system M, we define another MP system *crash* M by adding crash events. Note that we stay in the realm of MP systems (as defined in Section 2) without extending neither their syntax nor semantics.

The MP system *crash* M is identical with M except the following changes. In addition to a state from Q_i, the local state of process i (for every $1 \leq i \leq n$) contains a *crash flag*, which takes its values from $\{\bot, \top\}$. The value $\bot$ means that process i is crashed, otherwise the flag's value assumes $\top$. Formally, the local state of a process i is a tuple $q_i^c = (q_i, c_i)$, where $q_i \in Q_i$ and c_i is the crash flag of i. The set of events in *crash* M is $COMP^c = \cup_{i=1}^{n} COMP_i^c$, where, for every process i, $COMP_i^c = E_i \cup CE_i$ such that

- $E_i = \{comp' | comp \in COMP_i$ such that $comp'$ is identical with $comp$ and $comp'$ does not change $c_i\}$,
- $CE_i = \{comp^c | comp \in COMP_i$ such that $comp^c$ is identical with $comp$ and, when executed in a state, $comp^c$ sets $c_i = \bot$ and $MSG^c \subseteq MSG$ where MSG and MSG^c are the sets of messages sent by $comp$ and $comp^c\}$.

Intuitively, *crash* M inherits the events in E_i from the fault-free M, while CE_i contains the crash-faulty variants of these events. We call $comp^c$ in CE_i *crash-induced non-atomic send* if $MSG^c \subset MSG$ and $MSG^c \neq \emptyset$.

In addition, an event in *crash* M can only be executed by some process i if the crash flag c_i assumes $\top$. Formally, the guard of every event is extended with an additional condition (conjunct) defined as $c_i = \top$.[1]

4 The Equivalence of Explicit and Implicit Models

Given an MP system M, we call *crash* M an *explicit model* of crash faults. In contrast, M itself is an *implicit model* as no state transition directly models the crash of a process.

We first define a general equivalence between state graphs (Section 4.1), which we use to show as a special case that an explicit and the corresponding implicit models are equivalent (Section 4.2).

4.1 General Equivalence Basis

First, we define an equivalence relation between runs of Kripke structures. Intuitively, two runs are equivalent if they are of the same length and the i^{th} states in both runs are labeled the same.

Definition 1 *Given two Kripke structures* $M = (S, S_0, T, AP, L)$ *and* $M' = (S', S_0', T', AP, L')$*, a run* $\sigma = s_0, s_1, ...$ *in* M *is said to be* label-equivalent *with another run* $\sigma' = s_0', s_1', ...$ *in* M' *iff for every* $i = 0, 1, ...$, $L(s_i) = L'(s_i')$*. In this case, we write* $\sigma \approx_{AP} \sigma'$*.*

[1] Note that the guard of the dummy event (see Section 2.3) must not be changed.

The previous definition can be naturally generalized to the label-equivalence of two Kripke structures.

Definition 2 *Given two Kripke structures* $M = (S, S_0, T, AP, L)$ *and* $M' = (S', S'_0, T', AP, L')$*, they are said to be* label-equivalent *iff the following two conditions hold:*

- *For every run* σ *in* M*, there exists a run* σ' *in* M' *so that* $\sigma \approx_{AP} \sigma'$*.*
- *For every run* σ' *in* M'*, there exists a run* σ *in* M *so that* $\sigma \approx_{AP} \sigma'$*.*

The next corollary follows from the above definitions and the semantics of LTL [10]. It says that the truth of an arbitrary LTL property is indistinguishable in label-equivalent Kripke structures. The notation $M \models \phi$ means that the (LTL) formula ϕ holds for every run of the (Kripke structure) model M.

Corollary 1 *[10] Given two label-equivalent Kripke structures* M *and* M' *and a LTL formula* ϕ*, the following holds:*

$$M \models \phi \text{ iff } M' \models \phi \ .$$

Proof. The $\Rightarrow$ direction: Assume that $M' \not\models \phi$. Therefore, there must be a run σ' in M' such that $\sigma' \not\models \phi$. Since M and M' are label-equivalent, there is a run σ in M such that σ and σ' are label-equivalent. By definition, σ and σ' are of the same length and the corresponding states are labeled the same. This implies that $\sigma \not\models \phi$ [10], a contradiction.

The reverse direction can be similarly proven.

4.2 The Equivalence Theorem

In this section, we prove the label-equivalence between an MP system M and its crash-augmented version *crash* M. More precisely, we show label-equivalence between their Kripke structure counterparts.

To this end, we first define a special labeling function for MP systems, which is independent of the crashed status of processes and undelivered messages.

Definition 3 *Given an MP system* M*, a set of atomic propositions* AP*, the Kripke structure* (S, S_0, T, AP, L) *associated with* M*, and the Kripke structure* (S', S'_0, T', AP, L') *associated with* crash M*,* L *and* L' *are* crash/buffer -independent*, if for all* $s = (q_1, ..., q_n, b_1, ..., b_m) \in S$ *and* $s' = ((q_1, c_1), ..., (q_n, c_n), b'_1, ..., b'_m)) \in S'$*,* $L(s) = L'(s')$*.*

The following theorem states our main result, which together with Corollary 1 imply that an LTL formula holds for M iff it holds for *crash* M.

Theorem 1 *Given an MP system* M*, a set of atomic propositions* AP*, the Kripke structure* $M_{KS} = (S, S_0, T, AP, L)$ *associated with* M*, and the Kripke structure* $M^c_{KS} = (S', S'_0, T', AP, L')$ *associated with* crash M*, if* L *and* L' *are crash/buffer-independent, then* M_{KS} *and* M^c_{KS} *are label-equivalent.*

Proof. Let $\sigma = s_0, s_1, ...$ and $\sigma' = s'_0, s'_1, ...$ are runs of M_{KS} and M^c_{KS}, respectively. The proof is by induction on the length of the prefixes of σ and σ'. Given a prefix of σ (and σ'), we construct a prefix of a run in M^c_{KS} (in M_{KS}) such that label-equivalence holds for these prefixes. Then, label-equivalence between σ (and σ') and the constructed run follows by induction.

The $\Rightarrow$ direction. Intuitively, we construct a run σ' in *crash* M such that the events executed in M and *crash* M are the same. In other words, *crash* M simulates the non-faulty M.

Consider the prefix s_0, s_1 of σ as the base case. We know that $s_0 = (q_1, ..., q_m, b_1, ..., b_m) \in S_0$. In our construction, let $s'_0 = ((q'_1, c_1), ..., (q'_n, c_n), b'_1, ..., b'_m)$ be from S'_0 such that s_0 and s'_0 are *matching*, i.e., $q_1 = q'_1, ..., q_n = q'_n$ and $b_1 = b'_1, ..., b_m = b'_m$. Now, let *comp* be an event in M such that executing it in s_0 results in s_1. If *comp* is a dummy event, then we construct s'_1 such that $s'_1 = s'_0$. Otherwise, if *comp* is executed by process i, then let *comp*$'$ be a *matching event* with *comp*, i.e., *comp*$'$ is the event corresponding to *comp* as defined by E_i. Given that $s_1 = (qq_1, ..., qq_n, bb_1, ..., bb_m)$, let in our construction $s'_1 = ((qq_1, cc_1), ..., (qq_n, cc_n), bb_1, ..., bb_m)$ be the state resulting from the execution of *comp*$'$ in s'_0. Note that *comp*$'$ is enabled in s'_0 because $s'_0 \in S_0$ and so $c_i = \top$. Furthermore, since *comp* and *comp*$'$ are matching, there is an execution of *comp*$'$ satisfying that s_1 and s'_1 are matching over the local states of processes and the content of buffers. Since L and L' are crash/buffer-independent, we have that $L(s_0) = L'(s'_0)$ and $L(s_1) = L'(s'_1)$.

In the induction step, assume that there is a run in *crash* M with prefix $s'_0, s'_1, ..., s'_k$ that is label-equivalent with $s_0, s_1, ..., s_k$. Let s_k be the tuple $(q_1, ..., q_m, b_1, ..., b_m)$. By construction, we have that $s'_k = ((q_1, c_1), ..., (q_n, c_n), b_1, ..., b_m)$. The construction of s'_{k+1} is analogous to that of s'_1. Note that *comp*$'$ is enabled because $c_i = \top$ for all $1 \leq i \leq n$. This is because our construction selects *comp*$'$ from E_i, thus, the value of c_i remains unchanged.

The $\Leftarrow$ direction. Intuitively, we construct a run σ in M such that crashing and non-crashing events are replaced by there matching counterparts in M, i.e, non-faulty events that receive/send the same messages and perform the same local state transition.

Let s'_0, s'_1 be a prefix of σ' where $s'_0 = ((q_1, c_1), ..., (q_n, c_n), b_1, ..., b_m)$. Then, let $s_0 = (q_1, ..., q_n, b_1, ..., b_m)$ from S_0. We know that such s_0 exists by construction of *crash* M. Since L and L' are crash/buffer-independent, we have that $L'(s'_0) = L(s_0)$.

Now, let e be the event in *crash* M that results in s'_1 when executed s'_0. Similarly to the first part of the proof ($\Rightarrow$ direction), in case $e =$ *dum* and $e =$ *comp*$' \in E_i$, the corresponding event in M is *dum* and the matching *comp* that is used to construct s_1 when executed in s_0. If $e = comp^c \in CE_i$, then consider the matching event *comp* as defined by CE_i. Let s'_1 be the tuple $((qq_1, cc_1), ..., (qq_n, cc_n), bb_1, ..., bb_m)$. We construct $s_1 = (qq_1, ..., qq_n, bb'_1, ..., bb'_m)$ as the state resulting from the execution of *comp* in s_0. Note that the content of the buffers may be different, more precisely $bb_j \subseteq bb'_j$ for all $1 \leq j \leq m$, if $comp^c$

is a crash-induced non-atomic send. As L and L' are crash/buffer-independent, $L'(s'_1) = L(s_1)$ holds.

By the induction hypothesis, there is a run in M with prefix $s_0, ..., s_k$ that is label-equivalent with $s'_0, ..., s'_k$. By construction, given $s'_k = ((q_1, c_1), ..., (q_n, c_n), b_1, ..., b_m)$, we have that $s_k = (q_1, ..., q_n, b'_1, ..., b'_m)$ and $b_j \subseteq b'_j$ for all $1 \leq j \leq m$. The construction of s_{k+1} is similar to that of s_1. Note that the matching *comp* can always be executed in s_k because the buffers in s_k contains at least those messages in s'_k.

4.3 Implications of Different Buffer Models

Our model of MP systems from Section 2 assumes that every buffer is an *infinite* set of messages. As some applications might require modeling finite buffers, we now discuss how modeling finite buffers affects our equivalence result.

We consider two models of finite buffers. In the first model, a (non-dummy) event can only be enabled if *all* buffers that this event sends messages to have the capacity of delivering (storing) these messages. The proof of Theorem 1 can be easily modified using this model of finite buffers.

In the second model, a message m in a full buffer *buff* can be *overwritten* by a message m' that is sent via this buffer. This means that m will be lost and replaced by m' in *buff*. It turns out that the construction used in the proof of Theorem 1 does not work with this model of finite buffers. The problem is that these non-atomic send events can result in overwriting a *subset* of those messages that are overwritten in the non-faulty model. This might result in a process entering a local state that is unreachable for this process in the non-faulty model, thus, invalidating the equivalence result. Note that in our model with infinite buffers *all* messages that are available in *crash* M are also available in M, a property that does not hold using the second model of finite buffers.

5 Experiments: Model Checking Efficiency with Explicit and Implicit Models

Evaluation objective. Given an MP system with n processes, the explicit model is at least 2^n times larger than the implicit model. This is because for every every state in the implicit model there are 2^n corresponding states in the explicit model where every process can be crashed or alive. The exponential blow is further worsened by non-atomic sends. For simplicity, we consider a relaxed crash-model semantics where non-atomic sends are assumed not to happen.

Ideally, the size of a model is proportional with model checking memory and time. However, practical model checking can distort this trend. Firstly, a model checker might visit the successors of a state many times if this state is reachable through multiple runs. The reason for this is that storing and comparing states in stateful model checking [10] might be inefficient or even impossible given powerful specification languages [11]. Secondly, different reduction techniques [10]

enable sound verification by exploring only a fraction of the model. Depending on the system, one model can be better "reducable" than another.

Focusing on stateful and partial-order reduced [10] optimizations of model checking, our objective is to show that model checking the explicit model is exponentially more expensive (in terms of memory and time) than the implicit model. This would demonstrate the practicability of our equivalence result.

Example protocols. We consider two representative crash-tolerant protocols, i.e., they satisfy their specifications under the assumption that processes can only fail by crashing:

1. The Paxos protocol solves *consensus*, a fundamental primitive that can be used to implement state-machine replication [12]. Intuitively, consensus means that at most one value is "chosen", i.e., all processes agree on this value.
2. Our second example is *regular storage* protocol in the style of [2]. The objective of distributed storage is to reliably store data despite failures of the base (storing) objects. A regular storage guarantees that a read operation returns a value not older than the one written by the latest preceding write operation.

For debugging purposes, we inject faults into (a) correct processes and (b) the specification of the protocols and show that the model checker is able to find the bugs. In particular, we specify two faulty versions of Paxos, namely "Faulty Paxos" and "Faulty Paxos 2". For storage we require that a read operation that completes after a write has to return the value written by the write even if the two operations are concurrent ("Wrong Regularity"). More details and the source of these models can be found at [16].

Setup: tools, resources, and metrics. We use the MP-Basset model checker [5, 16] to conduct our experiments. MP-Basset is a model checker for message-passing systems implementing the following optimizations: stateful model checking via Java Pathfinder [15] and highly customizable static partial-order reduction [6]. In our experiments, partial-order reduction is customized for message-passing (read more details in Section 6). The experiments are run in the DETER testbed [17] on 2GHz Xeon machines with 4GB memory.

We measure model checking memory (the number of visited states) and time for each experiment. In the explicit model, we gradually add $1, 2, ...$ crash-prone processes and run a new experiment. In case of faulty protocols and specifications, the model checker stops at finding the first counterexample. Therefore, these searches are not exhaustive.

The model checker returns OK if the specification holds for the protocol. Otherwise, a counterexample (CE), i.e., a run violating the specification, is given. We add up to three (two) crashes for the OK (CE) cases. The reason of running more experiments without bugs is to measure how adding new crash-faulty processes affects the size of the explicit model.

Protocol (# processes)	Spec.	Result	Explicit model			Implicit model	
			# crashes	States	Time	States	Time
Paxos (6)	Safety	OK	1	1,541,622	9h50m	**548,961**	**3h18m**
			2	4,216,431	27h44m		
			3	11,843,034	83h		
Faulty Paxos (6)	Safety	CE	1	14,785	4m49s	**3,415**	**1m40s**
			2	33,598	10m53s		
Faulty Paxos 2 (7)	Safety	CE	1	1,442,262	12h20m	**173,414**	**1h28m**
			2	3,047,842	25h40m		
Register (5)	Regularity	OK	1	56,508	16m36s	**18,451**	**4m32s**
			2	128,697	40m50		
			3	301,562	1h40m		
Register (5)	Wrong regularity	CE	1	9,781	2m45	3,497	55s
			2	**1,213**	**29s**		
Register (6)	Wrong regularity	CE	1	18,272	7m	**6,987**	**2m32s**
			2	42,506	15m		

Table 1: Stateful and partial-order reduced state space exploration results with implicit and explicit models using the MP-Basset model checker.

Experimental results. Our results are shown in Table 1. We model check only meaningful instances of both protocols, i.e., at most one fault is tolerated. For each experiment, we emphasize the best result (least model checking memory and time) using bold text. We observe the following trends:

- The implicit model is more efficient than the explicit one in *all except one* experiments. In this one experiment the model checker finds the bug slightly faster using the explicit model. As the CE experiments are non-exhaustive, finding counterexamples quickly depends on how the model checker schedules events. In MP-Basset, the additional (crash) events in the explicit model affect this scheduling, as shown by our experiments. Heuristics can be used to "guide" the model checker towards the bug [13].
- Model checking memory and time of the explicit model is *exponential* in the number of crash-faulty processes compared to the implicit model. This trend is also depicted in Figure 1, where we show the number of states in the explicit model as a function of n, where n is the number of crashes. Note that the number of states grows even faster than 2^n. Again, this ideal formula is biased by the imperfect stateful optimization and partial-order reduction.

6 Related Work

Our reduction from the explicit to the implicit model allows sound and also complete verification of the specified class of properties (LTL with crash/buffer-independent labeling function). Although other reduction techniques such as symmetry or partial-order reductions [10] apply for a more general class of systems, they require manual intervention of the user. These techniques are orthogonal to the explicit/implicit model of crashes and can be applied for further reductions of both models.

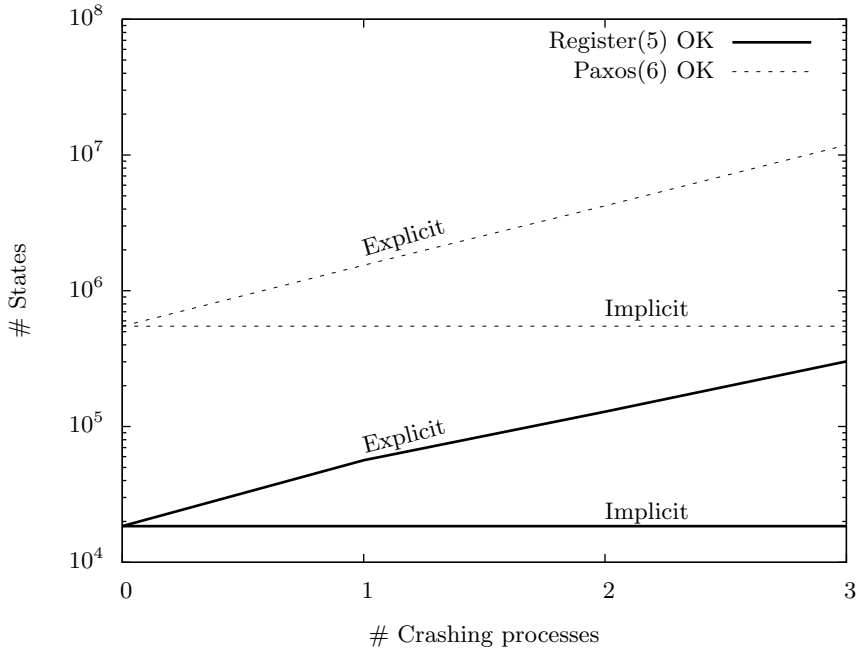

Fig. 1: The size of the explicit model as a function of the number of crashes.

In our experiments, we use partial-order reduction of both the explicit and implicit models. We apply this reduction for message-passing systems as proposed in [6]. For the explicit model, we extend the partial-order reduction with crash events and use the flexible and intuitive framework of [6] to prove the soundness of the reduction. Intuitively, we define events that are "non-interfering" with crash events, which is key to partial-order reduction. For example, a crash event e^c is non-interfering with every other event e in the sense that if e is disabled in a state, then it will remain so after the execution of e^c.

Another related reduction approach is [9], which reduces from a fine-grained model to a stuttering-equivalent coarse-grained model to allow efficient model checking. Although the underlying model is message-passing with crash faults, it assumes (synchronous) round-based communication and crashed-faults are expressed through so called Heard-Of sets. Our equivalence result does not directly apply under these assumptions but, instead, under the general model of [3].

7 Conclusion

We have defined a formal model that allows efficient model checking of message-passing systems with crash faults. The proposed model is a reduction from a detailed (and obviously sound) model and it accounts for sound verification

for a certain class of properties. Natural extensions of our approach include reductions for other fault-models (such as non-silent malicious faults) or proving the equivalence with respect to more general temporal logics (such as branching-time logics).

We see the strength of our contribution on the practical side. Our equivalence result formally verifies the natural intuition that crash events need not be modeled explicitly. Therefore, system designers can use this as a formal argument (rather than as "reasonable simplification") in the development and certification process. These are small but important steps towards scalable verification of real systems.

References

1. G. Agha, I.A. Mason, S. Smith, C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1): 1–72, 1997.
2. H. Attiya, A. Bar-Noy, D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42(1):124–142, 1995.
3. H. Attiya, J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley Series on Parallel and Distributed Computing, 2004.
4. P. Bokor, M. Serafini, N. Suri, On Efficient Models for Model Checking Message-Passing Distributed Protocols, IFIP Int. Conf. on Formal Techniques for Distributed Systems (FMOODS & FORTE), pages 216-223, 2010.
5. P. Bokor, J. Kinder, M. Serafini, N. Suri. Efficient Model Checking of Fault-Tolerant Distributed Protocols. *DSN-DCCS*, 2011, To appear.
6. P. Bokor, J. Kinder, M. Serafini, N. Suri. Supporting Domain-Specific State Space Reductions through Local Partial-Order Reduction. Technische Universität Darmstadt, Technical Report, 2011.
7. K. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*, Springer, 2005.
8. T.D. Chandra, S. Toueg Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, 1996.
9. M. Chaouch-Saad, V. Charron-Bost, S. Merz. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. Proc. *Reachability Problems*, pp. 93–106, 2009.
10. E. Clarke, O. Grumberg, D. Peled. *Model Checking*, MIT Press, 2000.
11. P. Godefroid. Model checking for programming languages using VeriSoft. *POPL*, pp. 174–186, 1997.
12. L. Lamport. The Part-time Parliament. *ACM Trans. Comp. Sys.*, 16(2):133–169, 1998.
13. M. Talupur, H. Han. Biased Model Checking Using Flows. *TACAS*, pp. 239–253, 2011.
14. J. Yang et al. MODIST: Transparent Model Checking of Unmodified Distributed Systems. *NSDI*, pp. 213–228, 2009.
15. http://babelfish.arc.nasa.gov/trac/jpf
16. http://www.deeds.informatik.tu-darmstadt.de/peter/mp-basset/
17. http://www.isi.deterlab.net/

A New Notion of Partial Correctness for Exception Handling

Emil Sekerinski and Tian Zhang

McMaster University, Hamilton, ON, Canada
emil@mcmaster.ca,zhangt26@mcmaster.ca

Abstract. We study the correctness of programs that use exception handling to deal with failures. A new notion of partial correctness is introduced for the design of programs that can continue safely after an unanticipated failure. Partial correctness is contrasted with total correctness though their verification rules. These rules are derived from a definition of statements with exceptions as higher order predicate transformers. The use of total and partial correctness is illustrated with three design patterns, rollback, degraded service, and recovery block.

1 Introduction

A program may fail to perform its intended task for several reasons: the specification may be in error, there may be errors in the design, or there may be failures of the underlying software or hardware. Some failures are always detected at run-time by the underlying (virtual) machine, some failures can be detected by programmer-added checks, while some failures would be too difficult to detect by any means.

Even with our best efforts to design error-free programs, in the design of any reasonably complex system, there always remains the possibility of a failure [13]. The question then arises how programs should respond to detected failures. Typically failures cannot be treated at the point in a program where they are detected, but control has to pass to some "outer" point. While the use of additional return values for indicating failure of a procedure call is common in operating sytems [8], the use of a dedicated mechanism for *exception handling* does not require additional variables and control structures to be interspersed in the original program for passing control to an outer point. Exception handling has the advantage that the original design remains visible. Besides, exception handling also has other uses, namely for an unobtrusive treatment of rare or undesired cases—cases that would affect the program structure in the same way as possible failures, and for coping with imperfections during the design process, like partial implementations of features.

Statements that can *raise* exceptions have a single entry, a normal exit, and additional exceptional exits. Hence such statements can be specified by a single precondition and one postcondition for each exit, originally suggested by Cristian [3] and more recently advocated in textbooks [11] and supported by

verification tools for the specification of class methods [2,9]. In this approach all possible failures have to be anticipated by the designer and any implementation of a method must not fail in any other way—a rather optimistic view. Hence a more typical use of exceptional exists is for undesired and rare cases, for example when looking up an entry or opening a file that may not be present. However, to deal with failures that can appear "everywhere", like running out of memory, these tools do not require that all exceptional exists are specified and do not verify the absence of so-called *unchecked* exceptions [6]. A different approach is advocated by Meyer [12]: each method has one entry, a normal exit, and an exceptional exit, but is specified by a single precondition and single postcondition only. The normal exit is taken if the desired postcondition is established and the exceptional exit is taken if the desired postcondition cannot be established. Thus the situations under which an exceptional exit is taken is implicit in the method specification and a "valid" outcome is always possible, even in the presence of unanticipated failures. Here we refine this view by restricting the exceptional postcondition in case the specified postcondition cannot be established.

We propose a notion of *partial correctness* that is meant to allow for unanticipated failure: statements may fail to establish the desired postcondition, but if they fail, they either must establish an alternative postcondition or not change the state (e.g. by restoring the original state). By extension, a *partial implementation* of a specification is one that is only partially correct. [1] For example, database transactions can be explained in terms of partial correctness: either a transaction succeeds, establishing the desired postcondition, or it fails and the original state is restored. Another example for partial correctness is the recovery block for software fault tolerance [4,14]: a list of alternative implementations is attempted in given order. If one alternative fails, the original state is restored and the next attempted, until either one succeeds or all fail. The design of class methods in robust object-oriented programs also follows the principles of partial correctness: if a method fails, it must at least establish the object invariant as the alternative postcondition, such that program execution can continue and methods of the object may still be called.

Partial correctness is contrasted to *total correctness* though their verification rules. The rules are justified with respect to a semantics of statements as higher order predicate transformers. The use of total and partial correctness is illustrated with three design patterns of increasing complexity, rollback, degraded service, and recovery block. Their treatment with total correctness in our earlier work [15] inspired the notion of partial correctness.

In the original approach by Cristian [3] statements have one entry and multiple exits (one of those being the normal one) and are defined by a set of predicate transformers, one for each exit. As pointed out by King and Morgan [7], this disallows nondeterminism, which precludes the use of the language for specification

[1] Partial correctness is also used in the literature when statements are not required to terminate. Here, this would be more appropriately referred to as *conditional correctness*, as our use of partial correctness does require termination.

and design; their solution is to use a single predicate transformer with one postcondition for each exit instead, which we follow here.

A mechanical formalization of try-catch-finally statements is given by Jacobs [5]. That formalization includes all the other "abrupt termination" modes of Java, which we do not need here, and uses state transformers, which precludes nondeterminism, and thus is less suited for our needs. Leino and Snepscheut [10] study basic algebraic properties of a language with exception handling and derive weakest exceptional preconditions of statements from a trace semantics. Here we start directly with a predicate transformer definition from which the proof rules for both total and partial correctness are derived. Algebraic properties of exceptions are studied by King and Morgan [7], but for a language in which a try-statement does not include a catch-statement. Our formalization in terms of higher order predicate transformers is inspired by that of Back and von Wright [1]. All theorems in this paper have been checked mechanically using the Isabelle/HOL prover; we therefore leave out the proofs or only sketch them.

The next section starts by defining a small but expressive set of core statements as higher order predicate transformers. Section 3 continues by defining common statements like assignment and conditional in terms of those; program expressions as they appear in assignments and conditionals are allowed to be partially defined. Section 4 defines the weakest precondition function and derives rules for common statements; separation is introduced as a desirable property, which restricts the class of statements that is considered from here on. Section 5 defines the termination, normal termination, exceptional termination, and enabledness domains of statements. Sections 6 and 7 define total and partial correctness and derive the corresponding verification rules. Sections 8, 9, and 10 present the rollback, degraded service, and recovery block patterns.

2 Statements with Two Exits

Dijkstra defines $wp(S, q)$ to be the weakest precondition such that statement S terminates and establishes postcondition q. As here only the input/output behaviour of statements is of interest, we identify a statement with its predicate transformer [1]. Statements have a single entry, a *normal exit*, and an *exceptional exit*. Hence we define $S(q, r)$ to be the weakest precondition such that statement S terminates, on normal termination postcondition q holds finally, and on exceptional termination postcondition r holds finally. A statement either *succeeds*, meaning it terminates normally, *fails*, meaning it terminates exceptionally, *aborts*, meaning it is out of control and may not terminate at all, or *blocks*, meaning that it refuses to execute.

A *state predicate* of type $\mathcal{P}\Sigma$ is a function from elements of type Σ, the state space, to $Bool$, i.e. $\mathcal{P}\Sigma \mathrel{\widehat{=}} \Sigma \to Bool$. A *relation* is a function from Δ, the initial state space, to a state predicate over Σ, the final state space, i.e. is of the form $\Delta \to \mathcal{P}\Sigma$; we allow the initial and final state spaces to be different. A *predicate transformer* is a function from a normal postcondition of type $\mathcal{P}\Psi$ and an exceptional postcondition of type $\mathcal{P}\Omega$, to a precondition of type $\mathcal{P}\Delta$, i.e. of

the form $\mathcal{P}\Psi \times \mathcal{P}\Omega \to \mathcal{P}\Delta$, for types Ψ, Ω, Δ. We leave the types out if they can be inferred from the context.

On state predicates, conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, consequence $\Leftarrow$, and negation $\neg$ are defined by the pointwise extension of the corresponding operations on *Bool*, e.g. $(p \wedge q)\sigma \mathrel{\widehat{=}} p\,\sigma \wedge q\,\sigma$ for state predicates p, q. The entailment ordering $\leq$ is defined by universal implication, meaning $p \leq q \mathrel{\widehat{=}} \forall \sigma \cdot p\,\sigma \Rightarrow q\,\sigma$. The predicates true and false represent the universally true respectively false predicates. Predicate transformer S is *monotonic* if $q \leq q'$ and $r \leq r'$ implies $S(q, r) \leq S(q', r')$. Hence weakening the normal or exceptional postcondition can lead only to a weaker precondition. A *statement* is a monotonic predicate transformer.

We define some basic predicate transformers: abort is completely unpredictable and may terminate normally or exceptionally in any state or may not terminate at all; stop miraculously guarantees any postcondition by blocking execution; skip does nothing and succeeds whereas raise does nothing and fails. The *sequential composition* $S\,;\,T$ continues with T only if S succeeds whereas the *exceptional composition* $S\,;;\,T$ continues with T only if S fails. The *demonic choice* $S \sqcap T$ establishes a postcondition only if both S and T do. The *angelic choice* $S \sqcup T$ establishes a postcondition if either S or T does:

$$
\begin{array}{lll}
\mathsf{abort}(q, r) & \mathrel{\widehat{=}} \mathsf{false} & (S\,;\,T)(q, r) \mathrel{\widehat{=}} S(T(q, r), r) \\
\mathsf{stop}(q, r) & \mathrel{\widehat{=}} \mathsf{true} & (S\,;;\,T)(q, r) \mathrel{\widehat{=}} S(q, T(q, r)) \\
\mathsf{skip}(q, r) & \mathrel{\widehat{=}} q & (S \sqcap T)(q, r) \mathrel{\widehat{=}} S(q, r) \wedge T(q, r) \\
\mathsf{raise}(q, r) & \mathrel{\widehat{=}} r & (S \sqcup T)(q, r) \mathrel{\widehat{=}} S(q, r) \vee T(q, r)
\end{array}
$$

Predicate transformers abort, stop, skip, and abort are monotonic and hence statements. Operators ;, ;;, $\sqcap$, $\sqcup$ preserve monotonicity. Sequential composition is associative and has skip as unit, giving rise to a monoid structure. Dually, exceptional composition is associative and has raise as unit, giving rise to another monoid structure.

We introduce statements for inspecting and modifying the state. For state predicates u and v, the *assumption* $[u, v]$ succeeds if u holds, fails if v holds, choosing demonically among these possibilities if both u and v hold, and stops if neither u nor v holds. The *assertion* $\{u, v\}$ succeeds if u holds, fails if v holds, choosing angelically among these possibilities if both u and v hold, and aborts if neither u nor v holds. Both assumption and assertion do not change the state.

$$
[u, v](q, r) \mathrel{\widehat{=}} (u \Rightarrow q) \wedge (v \Rightarrow r) \qquad\qquad \{u, v\}(q, r) \mathrel{\widehat{=}} (u \wedge q) \vee (v \wedge r)
$$

Both assumption and assertion are monotonic and hence statements. We have that $[\mathsf{true}, \mathsf{false}] = \mathsf{skip} = \{\mathsf{true}, \mathsf{false}\}$ and that $[\mathsf{false}, \mathsf{true}] = \mathsf{raise} = \{\mathsf{false}, \mathsf{true}\}$. We also have that $[\mathsf{false}, \mathsf{false}] = \mathsf{stop}$ and that $\{\mathsf{false}, \mathsf{false}\} = \mathsf{abort}$. Finally we have that $[\mathsf{true}, \mathsf{true}] = \mathsf{skip} \sqcap \mathsf{raise}$ and that $\{\mathsf{true}, \mathsf{true}\} = \mathsf{skip} \sqcup \mathsf{raise}$.

The *demonic update* $[Q, R]$ and the *angelic update* $\{Q, R\}$ both update the state according to relation Q and succeed or update the state according to relation R and fail, the difference being that the choice offered by the relations

and the choice between succeeding and failing is demonic with $[Q, R]$ and is angelic with $\{Q, R\}$. If Q is of type $\Delta \to \mathcal{P}\Psi$ and R is of type $\Delta \to \mathcal{P}\Omega$, then $[Q, R]$ and $\{Q, R\}$ are of type $\mathcal{P}\Psi \times \mathcal{P}\Omega \to \mathcal{P}\Delta$:

$$[Q, R](q, r)\delta \;\widehat{=}\; (\forall \psi \cdot Q\,\delta\,\psi \Rightarrow q\,\psi) \wedge (\forall \omega \cdot R\,\delta\,\omega \Rightarrow r\,\omega)$$
$$\{Q, R\}(q, r)\delta \;\widehat{=}\; (\exists \psi \cdot Q\,\delta\,\psi \wedge q\,\psi) \vee (\exists \omega \cdot R\,\delta\,\omega \wedge r\,\omega)$$

Both demonic update and angelic update are monotonic in both arguments and hence statements. Writing $\bot$ for the empty relation and id for the identity relation we have that $[\mathsf{id}, \bot] = \mathsf{skip} = \{\mathsf{id}, \bot\}$ and that $[\bot, \mathsf{id}] = \mathsf{raise} = \{\bot, \mathsf{id}\}$. We also have that $[\bot, \bot] = \mathsf{stop}$ and that $\{\bot, \bot\} = \mathsf{abort}$. Finally we have that $[\mathsf{id}, \mathsf{id}] = \mathsf{skip} \sqcap \mathsf{raise}$ and that $\{\mathsf{id}, \mathsf{id}\} = \mathsf{skip} \sqcup \mathsf{raise}$. Writing $\top$ for the universal relation, both updates $[\top, \top]$ and $\{\top, \top\}$ terminate, with $[\top, \top]$ making a demonic choice between succeeding and failing and a demonic choice among the final states, and $\{\top, \top\}$ making these choices angelic.

3 Derived Statements

To establish the connection to predicate transformers with a single postcondition we define:

$$[u](q, r) \;\widehat{=}\; u \Rightarrow q \qquad\qquad [Q](q, r)\delta \;\widehat{=}\; (\forall \psi \cdot Q\ \delta\ \psi \Rightarrow q\ \psi)$$
$$\{u\}(q, r) \;\widehat{=}\; u \wedge q \qquad\qquad \{Q\}(q, r)\delta \;\widehat{=}\; (\exists \psi \cdot Q\ \delta\ \psi \wedge q\ \psi)$$

These definitions are identical as for predicate transformers with a single postcondition, except for the additional parameter r [1]. We have that $[u] = [u, \mathsf{false}]$ and $\{u\} = \{u, \mathsf{false}\}$ as well as $[Q] = [Q, \bot]$ and $\{Q\} = \{Q, \bot\}$.

The common $\mathsf{try}\ S\ \mathsf{catch}\ T$ statement with *body* S and *handler* T is just a different notation for exceptional composition. The statement $\mathsf{try}\ S\ \mathsf{catch}\ T\ \mathsf{finally}\ U$ with *finalization* U can be defined in terms of sequential and exceptional composition. The finalization U is executed either after S succeeds, after S fails and T succeeds, or after S fails and T fails, in which case the whole statement fails whether U succeeds or fails; see Fig. 1:

$$\mathsf{try}\ S\ \mathsf{catch}\ T \;\widehat{=}\; S\ ;;\ T$$
$$\mathsf{try}\ S\ \mathsf{catch}\ T\ \mathsf{finally}\ U \;\widehat{=}\; (S\ ;;\ (T\ ;;\ (U\ ;\ \mathsf{raise})))\ ;\ U$$

The assignment statement $x := E$ is defined in terms of an update statement that affects only component x of the state space. For this we assume that the state is a tuple and variables select elements of the tuple. Here E may be partially defined, as for example in $x := x\ \mathsf{div}\ y$. A division by zero should lead to failure without a state change, otherwise to success with x updated. A *program expression* E is a term for which *definedness* $\mathsf{def}\ E$ and *value* $\mathsf{val}\ E$ are given; the result of $\mathsf{def}\ E$ and $\mathsf{val}\ E$ are expressions of the underlying logic. For example, assuming that c

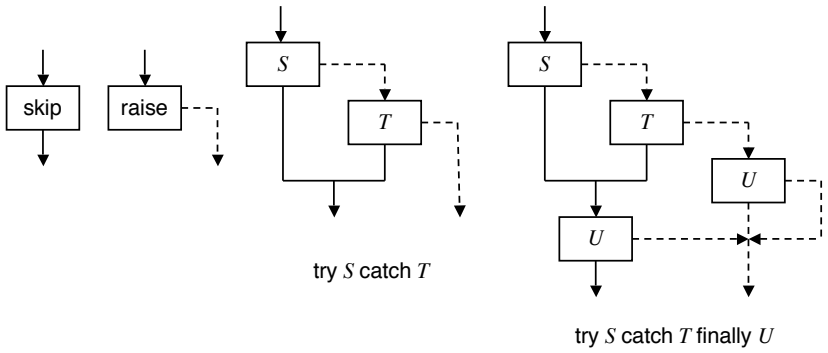

Fig. 1. Control flow of skip, raise, try S catch T and try S catch T finally U; outgoing solid lines represent the normal exit and outgoing dashed lines represent the exceptional exit

is a constant, x a variable, and $\sim$ is $+, -, =, <$ or another *strict* operator, we have:

$\mathsf{def}\,'c' = \mathsf{true}$	$\mathsf{val}\,'c' = c$
$\mathsf{def}\,'x' = \mathsf{true}$	$\mathsf{val}\,'x' = x$
$\mathsf{def}\,'E \sim F' = \mathsf{def}\,'E' \wedge \mathsf{def}\,'F'$	$\mathsf{val}\,'E \sim F' = \mathsf{val}\,'E' \sim \mathsf{val}\,'F'$
$\mathsf{def}\,'E\ \mathsf{div}\ F' = \mathsf{def}\,'E' \wedge \mathsf{def}\,'F' \wedge \mathsf{val}\,'F' \neq 0$	$\mathsf{val}\,'E\ \mathsf{div}\ F' = \mathsf{val}\,'E'\ \mathsf{div}\ \mathsf{val}\,'F'$
$\mathsf{def}\,'E\ \mathsf{mod}\ F' = \mathsf{def}\,'E' \wedge \mathsf{def}\,'F' \wedge \mathsf{val}\,'F' \neq 0$	$\mathsf{val}\,'E\ \mathsf{mod}\ F' = \mathsf{val}\,'E'\ \mathsf{mod}\ \mathsf{val}\,'F'$

For example, assuming that a, b are variables, we have for program expression $a\ \mathsf{mod}\ b$:

$$\begin{array}{ll} & \mathsf{def}\,'a\ \mathsf{mod}\ b' = \mathsf{true} \\ = & \mathsf{def}\,'a' \wedge \mathsf{def}\,'b' \wedge \mathsf{val}\,'b' \neq 0 \\ = & b \neq 0 \end{array} \qquad \begin{array}{ll} & \mathsf{val}\,'a\ \mathsf{mod}\ b' \\ = & \mathsf{val}\,'a'\ \mathsf{mod}\ \mathsf{val}\,'b' \\ = & a\ \mathsf{mod}\ b \end{array}$$

The *relational update* $x := e$ modifies component x of the state space to be e and leaves all other components of the state space unchanged; the initial and final state space are the same. The *nondeterministic relational update* $x :\in e$ modifies component x of the state space to be any element of the set e. Provided that the state space consists of variables x, y we define:

$$x := e \ \widehat{=}\ \lambda(x, y) \cdot \lambda(x', y') \cdot x' = e \wedge y' = y$$
$$x :\in e \ \widehat{=}\ \lambda(x, y) \cdot \lambda(x', y') \cdot x' \in e \wedge y' = y$$

For assumptions and assertions we introduce a syntactic abbreviation for leaving out the state space if that is evident from the context, thus following programming notation. Provided that the state space consists of variables x, y we define:

$$[b, c] \ \widehat{=}\ [\lambda(x, y) \cdot b, \lambda(x, y) \cdot c]$$
$$\{b, c\} \ \widehat{=}\ \{\lambda(x, y) \cdot b, \lambda(x, y) \cdot c\}$$

The *(deterministic) assignment* $x := E$ fails if program expression E is not defined, otherwise it succeeds and assigns the value of E to x. The *nondeterministic assignment* $x :\in E$ fails if E is not defined, otherwise it succeeds and assigns any element of the set E to x, the choice being demonic:

$$x := E \mathrel{\widehat{=}} \{\mathsf{def}\ E, \neg\, \mathsf{def}\ E\} \,;\, [x := \mathsf{val}\ E]$$
$$x :\in E \mathrel{\widehat{=}} \{\mathsf{def}\ E, \neg\, \mathsf{def}\ E\} \,;\, [x :\in \mathsf{val}\ E]$$

The *assertion statement* or *check statement* $\mathsf{check}\ B$ terminates normally if program expression B is defined and true and terminates exceptionally if B is undefined or false.

$$\mathsf{check}\ B \mathrel{\widehat{=}} \{\mathsf{def}\ B \wedge \mathsf{val}\ B, \neg\, \mathsf{def}\ B \vee \neg\, \mathsf{val}\ B\}$$

The *conditional* $\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T$ fails if B is not defined, otherwise continues with either S or T, depending on the value of B:

$$\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T \mathrel{\widehat{=}} \{\mathsf{def}\ B, \neg\, \mathsf{def}\ B\} \,;\, (([\mathsf{val}\ B] \,;\, S) \sqcap ([\neg\, \mathsf{val}\ B] \,;\, T))$$

As usually, $\mathsf{if}\ B\ \mathsf{then}\ S \mathrel{\widehat{=}} \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ \mathsf{skip}$; we leave out the treatment of loops. To illustrate the definitions, consider the statement $a := a \bmod b$:

$$\begin{aligned}
& a := a \bmod b \\
&= \{\mathsf{def}\ 'a \bmod b', \neg\, \mathsf{def}\ 'a \bmod b'\} \,;\, [a := \mathsf{val}\ 'a \bmod b'] \\
&= \{b \neq 0, b = 0\} \,;\, [a := a \bmod b] \\
&= \{\lambda(a,b) \cdot b \neq 0, \lambda(a,b) \cdot b = 0\} \,;\, [\lambda(a,b) \cdot \lambda(a',b') \cdot a' = a \bmod b \wedge b' = b]
\end{aligned}$$

4 Weakest Preconditions and Separation

We introduce the weakest precondition function $\mathsf{wp}(S, c, d)$, which allows the postconditions c and d to be written as (plain) predicates rather than state predicates. Provided that S is of type $\mathcal{P}\Sigma \times \mathcal{P}\Sigma \to \mathcal{P}\Sigma$ and assuming that $\overline{x}$ is the list program variables such that $\overline{x} : \Sigma$, we define:

$$\mathsf{wp}(S, c, d) \mathrel{\widehat{=}} S(\lambda \overline{x} \cdot c, \lambda \overline{x} \cdot d)(\overline{x})$$

Using wp allows us to reason about statements without needing to make the state space explicit, as in:

$$b > 0 \wedge x = a\ \mathsf{gcd}\ b \;\Rightarrow\; \mathsf{wp}(a := a \bmod b, x = a\ \mathsf{gcd}\ b, \mathsf{false})$$

We write $t[x \backslash u]$ for the substitution of variable x by u in t. The next theorem states the basic properties of wp.

Theorem 1. *Let S, T be predicate transformers, c, d be predicates, x be a variable, and E, B be program expressions:*

$$
\begin{array}{lcl}
\mathsf{wp}(\mathsf{abort}, c, d) & \equiv & \mathsf{false} \\
\mathsf{wp}(\mathsf{stop}, c, d) & \equiv & \mathsf{true} \\
\mathsf{wp}(\mathsf{skip}, c, d) & \equiv & c \\
\mathsf{wp}(\mathsf{raise}, c, d) & \equiv & d \\
\mathsf{wp}(\mathsf{check}\ B, c, d) & \equiv & (\mathsf{def}\ B \wedge \mathsf{val}\ B \Rightarrow c) \wedge \\
 & & (\mathsf{def}\ B \wedge \neg\, \mathsf{val}\ B \Rightarrow d) \wedge \\
 & & (\neg\, \mathsf{def}\ B \Rightarrow d) \\
\mathsf{wp}(x := E, c, d) & \equiv & (\mathsf{def}\ E \Rightarrow c[x \backslash \mathsf{val}\ E]) \wedge \\
 & & (\neg\, \mathsf{def}\ E \Rightarrow d) \\
\mathsf{wp}(x :\in E, c, d) & \equiv & (\mathsf{def}\ E \Rightarrow \forall x' \in \mathsf{val}\ E \cdot c[x \backslash x']) \wedge \\
 & & (\neg\, \mathsf{def}\ E \Rightarrow d) \\
\mathsf{wp}(S\ ;\ T, c, d) & \equiv & \mathsf{wp}(S, \mathsf{wp}(T, c, d), d) \\
\mathsf{wp}(\mathsf{try}\ S\ \mathsf{catch}\ T, c, d) & \equiv & \mathsf{wp}(S, c, \mathsf{wp}(T, c, d)) \\
\mathsf{wp}(S \sqcap T, c, d) & \equiv & \mathsf{wp}(S, c, d) \wedge \\
 & & \mathsf{wp}(T, c, d) \\
\mathsf{wp}(\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T, c, d) & \equiv & (\mathsf{def}\ B \wedge \mathsf{val}\ B \Rightarrow \mathsf{wp}(S, c, d)) \wedge \\
 & & (\mathsf{def}\ B \wedge \neg\, \mathsf{val}\ B \Rightarrow \mathsf{wp}(T, c, d)) \wedge \\
 & & (\neg\, \mathsf{def}\ B \Rightarrow d)
\end{array}
$$

As an example, consider determining the weakest precondition of $a := a \bmod b$ for normal postcondition $x = a \gcd b$ and exceptional postcondition d:

$$
\begin{array}{ll}
 & \mathsf{wp}(a := a \bmod b, x = a \gcd b, d) \\
\equiv & (\mathsf{def}\ 'a \bmod b' \Rightarrow (x = a \gcd b)[a \backslash \mathsf{val}\ 'a \bmod b']) \wedge (\neg\, \mathsf{def}\ 'a \bmod b' \Rightarrow d) \\
\equiv & (b \neq 0 \Rightarrow (x = a \gcd b)[a \backslash a \bmod b]) \wedge (b = 0 \Rightarrow d) \\
\equiv & (b \neq 0 \Rightarrow x = (a \bmod b) \gcd b) \wedge (b = 0 \Rightarrow d) \\
\equiv & (b \neq 0 \Rightarrow x = a \gcd b) \wedge (b = 0 \Rightarrow d)
\end{array}
$$

From this precondition we can deduce that if $b = 0$, the exceptional exit with postcondition d will be taken. To determine when then statement does not fail, we set d to false; the precondition then simplifies to $b \neq 0 \wedge x = a \gcd b$. In this rather simple example we have reasoned about normal and exceptional simultaneously. In larger programs it would be useful if such reasoning could be split into determining the weakest precondition for the normal and the exceptional postcondition separately.

A method for showing that a statement S establishes postconditions $c \wedge c'$, $d \wedge d$ (the normal and exceptional postcondition) is to show that S establishes c, d and it establishes c', d'. However, in general we have only following *subconjunctivity* property:

$$\mathsf{wp}(S, c, d) \wedge \mathsf{wp}(S, c', d') \Leftarrow \mathsf{wp}(S, c \wedge c', d \wedge d')$$

As a direct consequence of sub-conjuctivity we get following *sub-separation* property:

$$\mathsf{wp}(S, c, \mathsf{true}) \wedge \mathsf{wp}(S, \mathsf{true}, d) \Leftarrow \mathsf{wp}(S, c, d)$$

Unfortunately the direction of the implication does not allow the reasoning to be separated in general: we would like from $\mathsf{wp}(S, c, \mathsf{true})$ (succeeding with c or failing in any state) and $\mathsf{wp}(S, \mathsf{true}, d)$ (succeeding in any state or failing with d) to deduce $\mathsf{wp}(S, c, d)$. To rectify this, we define *(finite) conjunctivity*. Statement S is (finitely) conjunctive if:

$$\mathsf{wp}(S, c, d) \wedge \mathsf{wp}(S, c', d') \equiv \mathsf{wp}(S, c \wedge c', d \wedge d')$$

For conjunctive statement S *separation* holds:

$$\mathsf{wp}(S, c, \mathsf{true}) \wedge \mathsf{wp}(S, \mathsf{true}, d) \equiv \mathsf{wp}(S, c, d)$$

Statements abort, stop, skip, raise, assumption $[c, d]$, and demonic update $[Q, R]$ are conjunctive. Sequential composition, exceptional composition, and demonic choice preserve conjunctivity. The assertion $\{c, d\}$ is conjunctive only if c excludes d, i.e. $\neg(c \wedge d)$. Angelic choice and angelic update are in general not conjunctive. Since separation is a desirable property, we mainly consider conjunctive statements; all of the statements considered in Theorem 1 are conjunctive.

5 Domains

For statements with single exit, the *termination domain* $\mathsf{tr}\, S$ identifies when statement S does not abort and the *enabledness domain* $\mathsf{en}\, S$ identifies when the statement does not block. For statements with two exits, we have to distinguish further. The termination domain includes the *normal termination domain* $\mathsf{nr}\, S$ and the *exceptional termination domain* $\mathsf{ex}\, S$:

$$\begin{array}{ll} \mathsf{tr}\, S \;\widehat{=}\; \mathsf{wp}(S, \mathsf{true}, \mathsf{true}) & \mathsf{ex}\, S \;\widehat{=}\; \mathsf{wp}(S, \mathsf{false}, \mathsf{true}) \\ \mathsf{nr}\, S \;\widehat{=}\; \mathsf{wp}(S, \mathsf{true}, \mathsf{false}) & \mathsf{en}\, S \;\widehat{=}\; \neg\, \mathsf{wp}(S, \mathsf{false}, \mathsf{false}) \end{array}$$

As a corollary of sub-conjunctivity, we have $\mathsf{nr}\, S \wedge \mathsf{ex}\, S \Leftarrow \neg\, \mathsf{en}\, S$ for any statement S. This can be strengthen to $\mathsf{nr}\, S \wedge \mathsf{ex}\, S \equiv \neg\, \mathsf{en}\, S$ if S is conjunctive. The next theorem summarizes the basic properties of the domain operations.

Theorem 2. *Let S, T be predicate transformers, c, d be predicates, Q, R be relations, x a variable, and E, B program expressions:*

$$\begin{array}{llll} \mathsf{tr}\,\mathsf{abort} \equiv \mathsf{false} & \mathsf{tr}\,\mathsf{stop} \equiv \mathsf{true} & \mathsf{tr}\,\mathsf{skip} \equiv \mathsf{true} & \mathsf{tr}\,\mathsf{raise} \equiv \mathsf{true} \\ \mathsf{nr}\,\mathsf{abort} \equiv \mathsf{false} & \mathsf{nr}\,\mathsf{stop} \equiv \mathsf{true} & \mathsf{nr}\,\mathsf{skip} \equiv \mathsf{true} & \mathsf{nr}\,\mathsf{raise} \equiv \mathsf{false} \\ \mathsf{ex}\,\mathsf{abort} \equiv \mathsf{false} & \mathsf{ex}\,\mathsf{stop} \equiv \mathsf{true} & \mathsf{ex}\,\mathsf{skip} \equiv \mathsf{false} & \mathsf{ex}\,\mathsf{raise} \equiv \mathsf{true} \\ \mathsf{en}\,\mathsf{abort} \equiv \mathsf{true} & \mathsf{en}\,\mathsf{stop} \equiv \mathsf{false} & \mathsf{en}\,\mathsf{skip} \equiv \mathsf{true} & \mathsf{en}\,\mathsf{raise} \equiv \mathsf{true} \end{array}$$

$$\mathsf{tr}[c,d] \equiv \mathsf{true} \qquad \mathsf{tr}\{c,d\} \equiv c \vee d$$
$$\mathsf{nr}[c,d] \equiv \neg d \qquad \mathsf{nr}\{c,d\} \equiv c$$
$$\mathsf{ex}[c,d] \equiv \neg c \qquad \mathsf{ex}\{c,d\} \equiv d$$
$$\mathsf{en}[c,d] \equiv c \vee d \qquad \mathsf{en}\{c,d\} \equiv \mathsf{true}$$

$$\mathsf{tr}(\mathsf{check}\ B) \equiv \mathsf{true}$$
$$\mathsf{nr}(\mathsf{check}\ B) \equiv \mathsf{def}\ B \wedge \mathsf{val}\ B$$
$$\mathsf{ex}(\mathsf{check}\ B) \equiv \neg\,\mathsf{def}\ B \vee \neg\,\mathsf{val}\ B$$
$$\mathsf{en}(\mathsf{check}\ B) \equiv \mathsf{true}$$

$$\mathsf{tr}(x := E) \equiv \mathsf{true} \qquad \mathsf{tr}(x :\in E) \equiv \mathsf{true}$$
$$\mathsf{nr}(x := E) \equiv \mathsf{def}\ E \qquad \mathsf{nr}(x :\in E) \equiv \mathsf{def}\ E$$
$$\mathsf{ex}(x := E) \equiv \neg\,\mathsf{def}\ E \qquad \mathsf{ex}(x :\in E) \equiv \mathsf{def}\ E \Rightarrow (\mathsf{val}\ E = \{\})$$
$$\mathsf{en}(x := E) \equiv \mathsf{true} \qquad \mathsf{en}(x :\in E) \equiv \mathsf{def}\ E \Rightarrow (\mathsf{val}\ E \neq \{\})$$

$$\mathsf{tr}(S\ ;\ T) \Rightarrow \mathsf{tr}\ S \qquad \mathsf{tr}(S\ ;;\ T) \Rightarrow \mathsf{tr}\ S$$
$$\mathsf{nr}(S\ ;\ T) \Rightarrow \mathsf{nr}\ S \qquad \mathsf{nr}(S\ ;;\ T) \Leftarrow \mathsf{nr}\ S$$
$$\mathsf{ex}(S\ ;\ T) \Leftarrow \mathsf{ex}\ S \qquad \mathsf{ex}(S\ ;;\ T) \Rightarrow \mathsf{ex}\ S$$
$$\mathsf{en}(S\ ;\ T) \Rightarrow \mathsf{en}\ S \qquad \mathsf{en}(S\ ;;\ T) \Rightarrow \mathsf{en}\ S$$

$$\mathsf{tr}(S \sqcap T) \equiv \mathsf{tr}\ S \wedge \mathsf{tr}\ T \qquad \mathsf{tr}(S \sqcup T) \equiv \mathsf{tr}\ S \vee \mathsf{tr}\ T$$
$$\mathsf{nr}(S \sqcap T) \equiv \mathsf{nr}\ S \wedge \mathsf{nr}\ T \qquad \mathsf{nr}(S \sqcup T) \equiv \mathsf{nr}\ S \vee \mathsf{nr}\ T$$
$$\mathsf{ex}(S \sqcap T) \equiv \mathsf{ex}\ S \wedge \mathsf{ex}\ T \qquad \mathsf{ex}(S \sqcup T) \equiv \mathsf{ex}\ S \vee \mathsf{ex}\ T$$
$$\mathsf{en}(S \sqcap T) \equiv \mathsf{en}\ S \vee \mathsf{en}\ T \qquad \mathsf{en}(S \sqcup T) \equiv \mathsf{en}\ S \wedge \mathsf{en}\ T$$

$$\mathsf{tr}(\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T) \equiv (\mathsf{def}\ B \wedge \mathsf{val}\ B \Rightarrow \mathsf{tr}\ S) \wedge (\mathsf{def}\ B \wedge \neg\,\mathsf{val}\ B \Rightarrow \mathsf{tr}\ T)$$
$$\mathsf{nr}(\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T) \equiv \mathsf{def}\ B \wedge (\mathsf{val}\ B \Rightarrow \mathsf{nr}\ S) \wedge (\neg\,\mathsf{val}\ B \Rightarrow \mathsf{nr}\ T)$$
$$\mathsf{ex}(\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T) \equiv (\mathsf{def}\ B \wedge \mathsf{val}\ B \Rightarrow \mathsf{ex}\ S) \wedge (\mathsf{def}\ B \wedge \neg\,\mathsf{val}\ B \Rightarrow \mathsf{ex}\ T)$$
$$\mathsf{en}(\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T) \equiv \mathsf{def}\ B \Rightarrow (\mathsf{val}\ B \wedge \mathsf{en}\ S) \vee (\neg\,\mathsf{val}\ B \wedge \mathsf{en}\ T)$$

Statement stop blocks execution, hence $\mathsf{en}\,\mathsf{stop} \equiv \mathsf{false}$, and stop "terminates orderly" by refusing execution, hence $\mathsf{tr}\,\mathsf{stop} \equiv \mathsf{true}$. We note that demonic choice, blocking statements, and the enabledness domain are useful to express concurrency, without further elaborating on this.

6 Total Correctness

Hoare's total correctness assertion $[\![b]\!]\ S\ [\![c]\!]$ states that under precondition b, statement S terminates with postcondition c. This is now generalized to two postconditions, the normal and exceptional postcondition.

$[\![b]\!]\ S\ [\![c,d]\!]$ $\equiv$ Under precondition b, statement S terminates and
– on normal termination c holds finally,
– on exceptional termination d holds finally.

We say that under b, statement S succeeds with c and fails with d. If $[\![b]\!]\ S\ [\![c, \mathsf{false}]\!]$ holds, then S never fails, and we write this more concisely as $[\![b]\!]\ S\ [\![c]\!]$. Let b, c, d be predicates:

$$[\![b]\!]\ S\ [\![c, d]\!] \mathrel{\widehat{=}} b \Rightarrow \mathsf{wp}(S, c, d)$$
$$[\![b]\!]\ S\ [\![c]\!] \mathrel{\widehat{=}} b \Rightarrow \mathsf{wp}(S, c, \mathsf{false})$$

The next theorem summarizes the basic properties of total correctness, see also [3,5,7,10]:

Theorem 3. *Let b, c, d be predicates, B, E be program expressions, x be a variable, and S, T be statements:*

$$\begin{array}{lcl}
[\![b]\!]\ \mathsf{abort}\ [\![c, d]\!] & \equiv & \neg b \\
[\![b]\!]\ \mathsf{stop}\ [\![c, d]\!] & \equiv & \mathsf{true} \\
[\![b]\!]\ \mathsf{skip}\ [\![c, d]\!] & \equiv & b \Rightarrow c \\
[\![b]\!]\ \mathsf{raise}\ [\![c, d]\!] & \equiv & b \Rightarrow d \\
[\![b]\!]\ \mathsf{check}\ B\ [\![c, d]\!] & \equiv & (\mathsf{def}\ B \wedge \mathsf{val}\ B \wedge b \Rightarrow c) \wedge \\
 & & (\mathsf{def}\ B \wedge \neg\, \mathsf{val}\ B \wedge b \Rightarrow d) \wedge \\
 & & (\neg\, \mathsf{def}\ B \wedge b \Rightarrow d) \\
[\![b]\!]\ x := E\ [\![c, d]\!] & \equiv & (\mathsf{def}\ E \wedge b \Rightarrow c[x \backslash \mathsf{val}\ E]) \wedge \\
 & & (\neg\, \mathsf{def}\ E \wedge b \Rightarrow d) \\
[\![b]\!]\ x :\in E\ [\![c, d]\!] & \equiv & (\mathsf{def}\ E \wedge b \Rightarrow \forall x' \in \mathsf{val}\ E \cdot c[x \backslash x']) \wedge \\
 & & (\neg\, \mathsf{def}\ E \wedge b \Rightarrow d) \\
[\![b]\!]\ S\ ;\ T\ [\![c, d]\!] & \equiv & \exists h \cdot [\![b]\!]\ S\ [\![h, d]\!] \wedge \\
 & & \quad [\![h]\!]\ T\ [\![c, d]\!] \\
[\![b]\!]\ \mathsf{try}\ S\ \mathsf{catch}\ T\ [\![c, d]\!] & \equiv & \exists h \cdot [\![b]\!]\ S\ [\![d, h]\!] \wedge \\
 & & \quad [\![h]\!]\ T\ [\![d, d]\!] \\
[\![b]\!]\ S \sqcap T\ [\![c, d]\!] & \equiv & [\![b]\!]\ S\ [\![c, d]\!] \wedge \\
 & & [\![b]\!]\ T\ [\![c, d]\!] \\
[\![b]\!]\ \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ [\![c, d]\!] & \equiv & [\![\mathsf{def}\ B \wedge \mathsf{val}\ B \wedge b]\!]\ S\ [\![c, d]\!] \wedge \\
 & & [\![\mathsf{def}\ B \wedge \neg\, \mathsf{val}\ B \wedge b]\!]\ T\ [\![c, d]\!] \wedge \\
 & & (\neg\, \mathsf{def}\ B \wedge b \Rightarrow d)
\end{array}$$

We immediately get following consequence rule for any statement S:

$$(b' \Rightarrow b) \wedge [\![b]\!]\ S\ [\![c, d]\!] \wedge (c \Rightarrow c') \wedge (d \Rightarrow d') \quad \Rightarrow \quad [\![b']\!]\ S\ [\![c', d']\!]$$

For conjunctive statement S we have also:

$$[\![b]\!]\ S\ [\![c, d]\!] \wedge [\![b']\!]\ S\ [\![c', d']\!] \quad \Rightarrow \quad [\![b \wedge b']\!]\ S\ [\![c \wedge c', d \wedge d']\!]$$

Separation arises as a special case:

$$[\![b]\!]\ S\ [\![c, \mathsf{true}]\!] \wedge [\![b]\!]\ S\ [\![\mathsf{true}, d]\!] \quad \Rightarrow \quad [\![b]\!]\ S\ [\![c, d]\!]$$

7 Partial Correctness

The notion of total correctness assumes that any possible failure has to be anticipated in the specification; the outcome in case of failure is specified by the exceptional postcondition. An implementation may not fail in any other way. *Partial correctness* weakens the notion of total correctness by allowing also "true" exceptions. For orderly continuation after an unanticipated exception, the restriction is that in that case, the state must not change. We introduce following notation:

$$\langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, c, d \,|\!\rangle \quad \equiv \quad \begin{array}{l}\text{Under precondition } b\text{, statement } S \text{ terminates and}\\ \quad \text{– on normal termination } c \text{ holds finally,}\\ \quad \text{– on exceptional termination } b \text{ or } d \text{ holds finally.}\end{array}$$

Both total and partial correctness guarantee termination when the precondition holds. If $\langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, c, \mathsf{false} \,|\!\rangle$ holds, then statement S does not modify the state when terminating exceptionally, and we write this more concisely as $\langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, c \,|\!\rangle$. Let p, q, r be state predicates:

$$\begin{array}{ll}\langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, c, d \,|\!\rangle & \mathrel{\widehat{=}}\ b \Rightarrow \mathsf{wp}(S, c, b \vee d)\\ \langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, c \,|\!\rangle & \mathrel{\widehat{=}}\ b \Rightarrow \mathsf{wp}(S, c, b)\end{array}$$

Total correctness implies partial correctness, but not vice versa. The very definition of partial correctness breaks the duality between normal and exceptional postconditions that total correctness enjoys. This leads to some curious consequences that we will explore.

Theorem 4. *Let b, c, d be predicates, B, E be program expressions, x be a variable, and S, T be statements:*

$$\begin{array}{ll}\langle\!|\, b \,|\!\rangle\ \mathsf{abort}\ \langle\!|\, c, d \,|\!\rangle & \equiv \neg b\\ \langle\!|\, b \,|\!\rangle\ \mathsf{stop}\ \langle\!|\, c, d \,|\!\rangle & \equiv \mathsf{true}\\ \langle\!|\, b \,|\!\rangle\ \mathsf{skip}\ \langle\!|\, c, d \,|\!\rangle & \equiv b \Rightarrow c\\ \langle\!|\, b \,|\!\rangle\ \mathsf{raise}\ \langle\!|\, c, d \,|\!\rangle & \equiv \mathsf{true}\\ \langle\!|\, b \,|\!\rangle\ x := E\ \langle\!|\, c, d \,|\!\rangle & \equiv \mathsf{def}\ E \wedge b \Rightarrow c[x \backslash \mathsf{val}\ E]\\ \langle\!|\, b \,|\!\rangle\ x :\in E\ \langle\!|\, c, d \,|\!\rangle & \equiv \mathsf{def}\ E \wedge b \Rightarrow \forall x' \in \mathsf{val}\ E \cdot c[x \backslash x']\\ \langle\!|\, b \,|\!\rangle\ S\ ;\ T\ \langle\!|\, c, d \,|\!\rangle & \equiv \exists h \cdot \langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, h, d \,|\!\rangle \wedge\\ & \qquad [\![h]\!]\ T\ [\![c, b \vee d]\!]\\ \langle\!|\, b \,|\!\rangle\ \mathsf{try}\ S\ \mathsf{catch}\ T\ \langle\!|\, c, d \,|\!\rangle & \equiv \exists h \cdot [\![b]\!]\ S\ [\![c, h]\!] \wedge\\ & \qquad [\![h]\!]\ T\ [\![c, b \vee d]\!]\\ \langle\!|\, b \,|\!\rangle\ S \sqcap T\ \langle\!|\, c, d \,|\!\rangle & \equiv \langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, c, d \,|\!\rangle \wedge\\ & \quad \langle\!|\, b \,|\!\rangle\ T\ \langle\!|\, c, d \,|\!\rangle\\ \langle\!|\, b \,|\!\rangle\ \mathsf{check}\ B\ \langle\!|\, c, d \,|\!\rangle & \equiv \mathsf{def}\ B \wedge \mathsf{val}\ B \wedge b \Rightarrow c\\ \langle\!|\, b \,|\!\rangle\ \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ \langle\!|\, c, d \,|\!\rangle & \equiv [\![\mathsf{def}\ B \wedge \mathsf{val}\ B \wedge b]\!]\ S\ [\![c, b \vee d]\!] \wedge\\ & \qquad [\![\mathsf{def}\ B \wedge \neg\, \mathsf{val}\ B \wedge b]\!]\ T\ [\![c, b \vee d]\!]\end{array}$$

The raise statement miraculously satisfies any partial correctness specification by failing and leaving the state unchanged. The rules for assignment and nondeterministic assignment have only conditions in case E is defined; in case E is undefined, the assignment fails without changing the state, thus satisfies the partial correctness specification automatically. Likewise, the check statement and the conditional have only conditions in case B is defined, as if B is undefined, the statement fails without changing the state. We immediately get following consequence rule for any statement S:

$$\langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, c, d \,|\!\rangle \wedge (c \Rightarrow c') \wedge (d \Rightarrow d') \quad \Rightarrow \quad \langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, c', d' \,|\!\rangle$$

Like for total correctness, this rule allows the postconditions to be weakened, but does not allow the precondition to be weakened. For conjunctive statement S we have also:

$$[\![b]\!]\ S\ [\![c, d]\!] \wedge \langle\!|\, b' \,|\!\rangle\ S\ \langle\!|\, c', d' \,|\!\rangle \quad \Rightarrow \quad \langle\!|\, b \wedge b' \,|\!\rangle\ S\ \langle\!|\, c \wedge c', d \wedge d' \,|\!\rangle$$

Separation arises as a special case:

$$[\![b]\!]\ S\ [\![c, \mathsf{true}]\!] \wedge \langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, \mathsf{true}, d \,|\!\rangle \quad \Rightarrow \quad \langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, c, d \,|\!\rangle$$

For $S\,;\,T$ and $\mathsf{try}\ S\ \mathsf{catch}\ T$ let us consider the special case when $d \equiv \mathsf{false}$:

$$\begin{aligned} \langle\!|\, b \,|\!\rangle\ S\,;\,T\ \langle\!|\, c \,|\!\rangle &\equiv \exists h \cdot \langle\!|\, b \,|\!\rangle\ S\ \langle\!|\, h \,|\!\rangle \wedge [\![h]\!]\ T\ [\![c, b]\!] \\ \langle\!|\, b \,|\!\rangle\ \mathsf{try}\ S\ \mathsf{catch}\ T\ \langle\!|\, c \,|\!\rangle &\equiv \exists h \cdot [\![b]\!]\ S\ [\![c, h]\!] \wedge [\![h]\!]\ T\ [\![c, b]\!] \end{aligned}$$

The partial correctness assertion for $S\,;\,T$ is satisfied if S fails without changing the state, but if S succeeds with h, then T must either succeed with the specified postcondition c, or fail with the original precondition b. For the partial correctness assertion of $\mathsf{try}\ S\ \mathsf{catch}\ T$ to hold, either S must succeed with the specified postcondition c, or fail with h, from which T either succeeds with c or fails with the original precondition b.

8 Rollback

We continue with illustrating the use of total and partial correctness with three design patterns, starting with the *rollback* pattern. When a statement fails, it may leave the program in an inconsistent state, for example in one in which an invariant does not hold and from which another failure is likely, or in an undesirable state, for example one in which the only course of action is termination of the program. We give a pattern for *rolling back* to the original state such that the failure is *masked*, meaning that is it not visible to the outside. Rolling back relies on a procedure *backup*, which makes a copy of the state, and a procedure *restore*, which restores the saved state. We formalize this by requiring that *backup* establishes a predicate b, which *restore* requires to roll back, and which the attempted statement, called S, has to preserve in case of failure. The backup

may consist of a copy of all variables in main memory or secondary storage, or a partial or compressed copy, as long as a state satisfying c can be established. The attempted statement S does not need to preserve b in case of success, e.g. can overwrite the backup of the variables. We let statement T do some "cleanup" after restoring to achieve the desired postcondition.

Theorem 5. *Let b, c, d be predicates and let backup, restore, S, T be statements. If*

$$\langle\!| c |\!\rangle \ \mathit{backup} \ \langle\!| c \wedge b |\!\rangle \qquad [\![c \wedge b]\!] \ S \ [\![d, b]\!]$$
$$[\![b]\!] \ \mathit{restore} \ [\![c]\!] \qquad \langle\!| c |\!\rangle \ T \ \langle\!| d |\!\rangle$$

then:

$$\langle\!| c |\!\rangle \ \mathit{backup} \ ; \mathsf{try} \ S \ \mathsf{catch}(\mathit{restore} \ ; \ T) \ \langle\!| d |\!\rangle$$

Procedure *backup* may either fail with c or succeed with $c \wedge b$, but *restore* must always succeed. The cleanup T must either succeed with d or fail with its precondition c. Thus it can be implemented by raise, which would be an example of *re-raising* an exception.

The theorem follows directly from the rules of total correctness by first rephrasing the partial correctness assertions as total correctness assertions.

9 Degraded Service

Suppose that two or more statements achieve the same goal, but some statements are preferred over others—the preferred one may be more efficient, may achieve a higher precision of numeric results, may transmit faster over the network, may achieve a higher sound quality. If the most preferred one fails, we may *fall back* to one that is less desirable, but more likely to succeed, and if that fails, fall back to a third one, and so forth. The least preferred one may simply inform the user of the failure. We call this the pattern of *degraded service*. In the formulation below degraded service is combined with rollback such that each attempt starts in the original state, rather than in the state that was left from the previous attempt. Hence, all alternatives have to adhere to the same specification, but try to satisfy that by different means. In case all attempts fail, the failure is *propagated* to the user.

Theorem 6. *Let b, c, d be predicates and let backup, restore, S_1, S_2 be statements. If*

$$\langle\!| c |\!\rangle \ \mathit{backup} \ \langle\!| c \wedge b |\!\rangle \qquad [\![c \wedge b]\!] \ S_1 \ [\![d, b]\!]$$
$$[\![b]\!] \ \mathit{restore} \ [\![c \wedge b]\!] \qquad [\![c \wedge b]\!] \ S_2 \ [\![d, b]\!]$$

then:

$$\langle\!| c |\!\rangle \ \mathit{backup} \ ; \mathsf{try} \ S_1 \ \mathsf{catch}(\mathit{restore} \ ; \mathsf{try} \ S_2 \ \mathsf{catch}(\mathit{restore} \ ; \mathsf{raise})) \ \langle\!| d |\!\rangle$$

The theorem readily generalizes to more than two attempts. Again, the theorem follows directly from the rules of total correctness by first rephrasing the partial correctness assertions as total correctness assertions.

10 Recovery Block

The *recovery block* specifies N alternatives together with an *acceptance test* [4]. The alternatives are executed in the specified order. If the acceptance test at the end of an alternative fails or an exception is raised within an alternative, the original state is restored and the next alternative attempted. If an acceptance test passes, the recovery block terminates. If the acceptance test fails for all alternatives, the recovery block fails, possibly leading to alternatives taken at an outer level. Here is the originally suggested syntax of [14] and a formulation with try-catch statements [15]; A is the acceptance test:

ensure A	*backup* ;
by S_1	try(S_1 ; check A)
else by S_2	catch
else by S_3	*restore* ;
else error	try(S_2 ; check A)
	catch
	restore ;
	try(S_3 ; check A)
	catch(*restore* ; raise)

The acceptance test does not have to be the complete postcondition—that would be rather impractical in general. However, suppose that we know that alternative S_i succeeds with d_i, if it succeeds. If we can devise a predicate A_i such that $d_i \wedge A_i$ implies the desired postcondition d, then A_i is an *adequate acceptance test* for S_i; for this each alternative has to have its own acceptance test, a possibility already mentioned in [14]:

Theorem 7. *Let b, c, d, d_1, d_2, d_3 be predicates, let A_1, A_2, A_3 be program expressions, and let backup, restore, S_1, S_2, S_3 be statements. If*

$$\begin{array}{llll} \langle\!| c |\!\rangle \; \mathit{backup} \; \langle\!| c \wedge b |\!\rangle & [\![c \wedge b]\!] \; S_1 \; [\![d_1 \wedge b, b]\!] & d_1 \Rightarrow \mathsf{def}\, A_1 & d_1 \wedge \mathsf{val}\, A_1 \Rightarrow d \\ [\![b]\!] \; \mathit{restore} \; [\![c \wedge b]\!] & [\![c \wedge b]\!] \; S_2 \; [\![d_2 \wedge b, b]\!] & d_2 \Rightarrow \mathsf{def}\, A_2 & d_2 \wedge \mathsf{val}\, A_2 \Rightarrow d \\ & [\![c \wedge b]\!] \; S_3 \; [\![d_3 \wedge b, b]\!] & d_3 \Rightarrow \mathsf{def}\, A_3 & d_3 \wedge \mathsf{val}\, A_3 \Rightarrow d \end{array}$$

then:

$\langle\!| c |\!\rangle$
  backup ;
  try(S_1 ; check A_1)
  catch
    restore ;
    try(S_2 ; check A_2)
    catch
      restore ;
      try(S_3 ; check A_3)
      catch(*restore* ; raise)
$\langle\!| d |\!\rangle$

More generally, partial acceptance tests in form of additional check statements can be carried out anywhere within an alternative, rather than only at the end; failure should be detected early such that resources are not wasted.

The theorem is a consequence of degraded service with rollback, generalized to three attempts, by replacing S_1 by S_1 ; $\mathsf{check}\, A_1, \ldots$. The conclusion follows immediately provided that $[\![c \wedge b]\!]\, S_1; \mathsf{check}\, A_1\, [\![d, b]\!], \ldots$ hold. Given $[\![c \wedge b]\!]\, S_1\, [\![d_1 \wedge b, b]\!]$, $d_1 \Rightarrow \mathsf{def}\, A_1$, and $d_1 \wedge \mathsf{val}\, A_1 \Rightarrow d, \ldots$ this follows by the rules for total correctness assertions.

11 Conclusions

The formalization of the three design patterns gives some evidence that the notions of total and partial correctness (in our sense) are useful in providing a notation and a method for addressing "unanticipated" exceptions that signal some failure; the practicality depends of course on the degree to which failures can be detected, which we do not address here. While proofs are most easily carried out using total correctness assertions, partial correctness seems to be more useful for specification.

The focus of this paper is the theoretical foundation. Practical verification tools augment specification with a *frame* that restricts which variables can be modified and place other restrictions for making the verification conditions local [2,9]. We have not addressed these issues.

The rules given for partial and total correctness are not complete; notably, loops are modules have not been treated. They are needed for the formalization of further patterns and remain the topic of ongoing work [16,12]. While the statement $\mathsf{try}\, S\, \mathsf{catch}\, T\, \mathsf{finally}\, U$ was defined, no corresponding proof rules were given. A directly derived total correctness rule is:

$$\begin{aligned}[\![p]\!]\ \mathsf{try}\, S\, \mathsf{catch}\, T\, \mathsf{finally}\, U\ [\![q, r]\!] \equiv \exists k, l, m, n \cdot\ & [\![p]\!]\, S\, [\![k, l]\!] \wedge [\![l]\!]\, T\, [\![m, n]\!] \wedge \\ & [\![k]\!]\, U\, [\![q, r]\!] \wedge [\![m]\!]\, U\, [\![q, r]\!] \wedge \\ & [\![n]\!]\, U\, [\![r, r]\!]\end{aligned}$$

However, this rules has three—very different—conditions on the finalization U. While the rule is suitable for verifying programs, these three conditions are of no help to the programmer in systematically developing a finalization. More work on the verification of finalization is needed.

Acknowledgement. We are grateful to the reviewers; their comments lead to numerous improvements.

References

1. R. J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.

2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2005.
3. F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10(2):163–174, 1984.
4. J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium*, pages 171–187, London, UK, 1974. Springer-Verlag.
5. B. Jacobs. A formalisation of Java's exception mechanism. In D. Sands, editor, *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 284–301, London, UK, 2001. Springer-Verlag.
6. B. Jacobs, P. Müller, and F. Piessens. Sound reasoning about unchecked exceptions. In *SEFM 2007: Fifth IEEE International Conference on Software Engineering and Formal Methods.*, pages 113–122, September 2007.
7. S. King and C. Morgan. Exits in the refinement calculus. *Formal Aspects of Computing*, 7(1):54–76, 1995.
8. P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, 2000.
9. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31:1–38, May 2006.
10. K. R. M. Leino and J. L. A. van de Snepscheut. Semantics of exceptions. In E.-R. Olderog, editor, *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi*, IFIP Transactions A-56, pages 447–466. North-Holland Publishing Co., 1994.
11. B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., 2000.
12. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 2nd edition, 1997.
13. D. L. Parnas and H. Würges. Response to undesired events in software systems. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 437–446. IEEE Computer Society Press, 1976.
14. B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449. ACM, 1975.
15. E. Sekerinski. Exceptions for dependability. In L. Petre, K. Sere, and E. Troubitsyna, editors, *Dependability and Computer Engineering: Concepts for Software-Intensive Systems—a Handbook on Dependability Research*. IGI Global, 2011.
16. Jie Xu, B. Randell, A. Romanovsky, C.M.F. Rubira, R.J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 499–508. IEEE Computer Society, 1995.