Disassembling Real-Time Fault-Tolerant Programs[*]

Borzoo Bonakdarpour
3115 Engineering Building
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48823, USA
borzoo@cse.msu.edu

Sandeep S. Kulkarni
3115 Engineering Building
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48823, USA
sandeep@cse.msu.edu

Anish Arora
395 Dreese Hall
Department of Computer Science and Engineering
Ohio State University
Columbus, OH 43210, USA
anish@cse.ohio-state.edu

ABSTRACT

We focus on decomposition of *hard-masking* real-time fault-tolerant programs (where safety, timing constraints, and liveness are preserved in the presence of faults) that are designed from their fault-intolerant versions. Towards this end, motivated by the concepts of *state predicate detection* and *state predicate correction*, we identify three types of fault-tolerance components, namely, *detectors*, *weak δ-correctors*, and *strong δ-correctors*. We show that any hard-masking program can be decomposed into its fault-intolerant version plus a collection of detectors, and, weak and strong δ-correctors. We argue that such decomposition assists in providing assurance about dependability and time-predictability of embedded systems.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance, Verification*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time and embedded systems*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logic of programs*

General Terms

Theory, Verification, Reliability

Keywords

Fault-tolerance, Real-time, Component-based analysis, Decomposition, Bounded-time recovery, Formal methods

1. INTRODUCTION

Dependability and time-predictability are two vital properties of most embedded (especially, safety/mission-critical) systems. Consequently, providing *fault-tolerance* and meeting *timing constraints* are two inevitable aspects of dependable real-time embedded systems. Thus, it is highly desirable to have access to methodologies to formally reason about and, hence, gain assurance of these aspects during the design and analysis of embedded systems. In the context of analysis, verification of fault-tolerant real-time embedded systems may be accomplished by illustrating the existence of constituents that (1) guarantee fault-tolerance, (2) ensure timing constraints, and (3) perform basic functionalities. With this motivation, we focus on the following question:

> *Can a real-time fault-tolerant program be decomposed into components that can assist in its verification?*

In this paper, we answer this question affirmatively by broadening *the theory of fault-tolerance components* [5] to the context of real-time programs. The theory in [5] essentially separates fault-tolerance and functionality concerns of untimed systems. More specifically, the theory identifies two types of fault-tolerance components, namely *detectors* and *correctors*. These components are based on the principle of detecting a *state predicate* to ensure that program actions would be safe and correcting a *state predicate* to ensure that the program eventually reaches a legitimate state. We emphasize that since these components do not rely on *detecting faults* or *correcting faults*, they can be applied in cases where faults are not detectable (e.g., Byzantine faults).

In the context of real-time programs, we focus on decomposition of *hard-masking* [6] real-time fault-tolerant programs, where (1) (timing independent) safety, (2) timing constraints, and (3) liveness properties (including recovery to legitimate states) are met even in the presence of faults. We identify three types of components, namely, *detectors*, *weak δ-correctors*, and *strong δ-correctors*. We show that these three components are in turn responsible for meeting the three properties of hard-masking fault-tolerant programs. Our proofs are constructive in the sense that they assist in identifying and subsequently decomposing a given hard-masking program into its fault-intolerant version, detectors, and δ-correctors.

Intuitively, detectors and δ-correctors work as follows. Each of these components is specified using two predicates: a *detection* (respectively, *correction*) predicate and a *witness* predicate. The goal of the detector component is to detect whether the given detection predicate is true and

[*]This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

subsequently satisfy the witness predicate. It is required that whenever the witness predicate is true, the detection predicate must be true as well. Thus, the fault-tolerant program can use the witness predicate of the detector to provide the desired fault-tolerance requirements. In case of a δ-corrector, the component restores the program to a state where the correction predicate is true within a bounded amount of time. The use of the witness predicate by the program is optional, as the program may not need to know when the program state is restored.

Since we focus on demonstrating the *existence* of these components in a given hard-masking program, our notion of decomposition differs from that in [5]. In particular, we precisely define what it means for a fault-tolerant program to *reuse* a fault-intolerant program. Furthermore, we formally define what it means for a fault-tolerant program to *contain* detectors and/or δ-correctors. We note that for assistance in analysis, it is necessary to ensure that the specification of the added components can be derived from the fault-intolerant version. Thus, in case of detectors, it is necessary to specify how the results of these components (intuitively, the conclusion that the predicate being detected is true) are *syntactically* used in the fault-tolerant real-time program. This part is not important in the context of a design methodology and, hence, is not formalized in [5].

Although detectors and correctors have been found to be useful in the *design* of fault-tolerant programs [1], their significance in analysis has not been evaluated except in empirical case studies. In these studies [11, 18], decomposition of a fault-tolerant program into its components has been found valuable in formal verification of the program. Thus, we expect that an affirmative answer to existence of the components would significantly assist in analysis of real-time embedded fault-tolerant programs.

Organization. First, we present related work in Section 2. In Section 3, we formally define real-time programs and specifications. Section 4 is dedicated to present our fault model and the notion of hard-masking fault-tolerance. Then, in Section 5 (respectively, 6), we present the notion of detector (respectively, δ-corrector) components, the concept of their containment in real-time programs, and their theory of decomposition. Finally, we make concluding remarks and discuss future work in Section 7.

2. RELATED WORK

The theory of detectors and correctors [5] was extended in [14] for safety-critical systems. In [20], the authors have used a similar approach for proving convergence of systems to legitimate states. The theory has also been used in design of several multi-tolerant examples [12, 17] where tolerance to different types of faults is provided and the level of fault-tolerance varies depending upon the severity of faults. In the context of automation of addition of fault-tolerance, the theory has been exploited in [6, 7, 11, 15, 17]. In the context of verification, simplified versions of this theory are applied in verification of time-triggered architectures [19]. It has also been used in software verification through separation of concerns [18].

This work differs from the work on failure detectors (e.g., the line of research pioneered by Chandra and Toueg) in that the predicates being detected in [8, 9] are of the type "process j has failed". To the contrary, the predicates in our work are arbitrary state predicates. Moreover, in [8, 9], the authors have considered detectors that are not perfect; similar detectors can also be constructed from the components in this paper. However, this issue is important in the context of a *design* methodology and is discussed in [4, 6, 17]. Thus, issues such as *atomicity* or *perfectness* of the fault-tolerance components are outside the scope of this paper; in the context of analysis, the components *contained* in a fault-tolerant program, by definition, would satisfy any atomicity restrictions imposed on that fault-tolerant program.

[1]The components have been shown to suffice in the design of a large class of fault-tolerant programs [5] including programs designed using replication, state machine approach, and, checkpointing and recovery.

3. REAL-TIME PROGRAMS AND SPECIFICATIONS

In our framework, real-time programs are specified in terms of their state space and their transitions [2,3]. The definition of specification is adapted from Alpern and Schneider [1] and Henzinger [13].

3.1 Real-Time Programs

Let $V = \{v_1, v_2 \cdots v_n\}$, $n \geq 1$, be a finite set of *discrete variables* and $X = \{x_1, x_2 \cdots x_m\}$, $m \geq 1$, be a finite set of *clock variables*. Each discrete variable v_i, $1 \leq i \leq n$, is associated with a finite *domain* D_i of values. Each clock variable x_j, $1 \leq j \leq m$, ranges over nonnegative real numbers (denoted $\mathbb{R}_{\geq 0}$). A *location* is a function that maps discrete variables in V to a value from their respective domain. A *clock constraint* over X is a Boolean combination of formulae of the form $x \preceq c$ or $x - y \preceq c$, where $x, y \in X$, $c \in \mathbb{Z}_{\geq 0}$, and $\preceq$ is either $<$ or $\leq$. We denote the set of all clock constraints over X by $\Phi(X)$. A *clock valuation* is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable.

For $\tau \in \mathbb{R}_{\geq 0}$, we write $\nu + \tau$ to denote $\nu(x) + \tau$ for every clock variable x in X. Also, for $\lambda \subseteq X$, $\nu[\lambda := 0]$ denotes the clock valuation that assigns 0 to each $x \in \lambda$ and agrees with ν over the rest of the clock variables in X. A *state* (denoted σ) is a pair (s, ν), where s is a location and ν is a clock valuation for X. Let u be a (discrete or clock) variable and σ be a state. We denote the value of u in state σ by $u(\sigma)$. The set of all possible states is called the *state space* obtained from the associated variables.

Definition 3.1 (computations) Let V and X be finite sets of discrete and clock variables respectively. A *computation* is a finite or infinite timed state sequence of the form $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ iff the following conditions are satisfied (1) $\sigma_i = (s_i, \nu_i)$ is a state in the state space of V and X for all $i \in \mathbb{Z}_{\geq 0}$, and (2) the sequence $\tau_0, \tau_1, \cdots$ (called the *global time*), where $\tau_i \in \mathbb{R}_{\geq 0}$ for all $i \in \mathbb{Z}_{\geq 0}$, satisfies the following constraints:

- *(monotonicity)* for all $i \in \mathbb{Z}_{\geq 0}$, $\tau_i \leq \tau_{i+1}$,
- *(time consistency)* for all $i \in \mathbb{Z}_{\geq 0}$, (1) if $\tau_i < \tau_{i+1}$ then $s_i = s_{i+1}$ and $\nu_{i+1}(x) = \nu_i(x) + (\tau_{i+1} - \tau_i)$ for all $x \in X$, and (2) if $\tau_i = \tau_{i+1}$ then $\nu_{i+1} = \nu_i[\lambda := 0]$ for some λ, where $\lambda \subseteq X$. ■

Notice that in Definition 3.1, we do not specify an initial value for the global time. Now, let Σ be any set of computations. We require that Σ must be closed with respect to *time offsets*. That is, $\forall \overline{\sigma} \in \Sigma : \forall t \in \mathbb{R} : (\overline{\sigma} + t) \in \Sigma$, where $\overline{\sigma} + t$ denotes the computation $(\sigma_0, \tau_0 + t) \rightarrow (\sigma_1, \tau_1 + t) \rightarrow \cdots$, st. $\tau_0 + t \geq 0$.

Notation. Let $\overline{\sigma}_i$ denote the pair (σ_i, τ_i) in computation $\overline{\sigma}$. Also, let $\overline{\alpha}$ be a finite computation of length n and $\overline{\beta}$ be a finite or infinite computation. The *concatenation* of $\overline{\alpha}$ and $\overline{\beta}$ (denoted $\overline{\alpha}\overline{\beta}$) is a computation, iff states $\overline{\alpha}_{n-1}$ and $\overline{\beta}_0$ meet the constraints of Definition 3.1. Otherwise, the result of concatenation is null. If Γ and Ψ are two sets containing finite and finite/infinte computations respectively, then $\Gamma\Psi = \{ \overline{\alpha}\overline{\beta} \mid (\overline{\alpha} \in \Gamma) \wedge (\overline{\beta} \in \Psi) \}$.

Definition 3.2 **(suffix and fusion closure)** *Suffix closure* of a set of computations means that if a computation $\overline{\sigma}$ is in that set then so are all the suffixes of $\overline{\sigma}$. *Fusion closure* of a set of computations means that if computations $\overline{\alpha}(\sigma, \tau)\overline{\gamma}$ and $\overline{\beta}(\sigma, \tau)\overline{\psi}$ are in that set then so are the computations $\overline{\alpha}(\sigma, \tau)\overline{\psi}$ and $\overline{\beta}(\sigma, \tau)\overline{\gamma}$, where $\overline{\alpha}$ and $\overline{\beta}$ are computation prefixes, $\overline{\gamma}$ and $\overline{\psi}$ are computation suffixes, and σ is a state at global time τ. ∎

Definition 3.3 **(real-time programs)** A *real-time program* $\mathcal{P}$ is specified by the tuple $\langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P} \rangle$ where $V_\mathcal{P}$ is a finite set of discrete variables, $X_\mathcal{P}$ is a finite set of clock variables, and $\Pi_\mathcal{P}$ is a suffix closed and fusion closed set of infinite *maximal* computations in the state space of $\mathcal{P}$. By maximal, we mean that if $\overline{\sigma} = \overline{\alpha}\overline{\beta}$ is in $\Pi_\mathcal{P}$, where (1) $\overline{\alpha} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$, (2) $\overline{\beta} = ((s_{n+1}, \nu_{n+1}), \tau_{n+1}) \rightarrow ((s_{n+2}, \nu_{n+2}), \tau_{n+2}) \rightarrow ((s_{n+3}, \nu_{n+3}), \tau_{n+3}) \cdots$, and (3) for all $j > n$, $s_j = s_{j+1}$ and $\nu_{j+1} = \nu_j + (\tau_{j+1} - \tau_j)$, then no other computation in $\Pi_\mathcal{P}$ has a prefix of $\overline{\alpha}$. In other words, given a computation prefix $\overline{\alpha}$ of $\mathcal{P}$, $\mathcal{P}$ does not contain the computation that stutters σ_{n+1} infinitely if there exists other computation of $\mathcal{P}$ that extends $\overline{\alpha}$. ∎

Observation. One can observe that Definitions 3.1 and 3.3 allow real-time programs (and later specifications) to exhibit *Zeno* behavior. The reason is due to the fact that when we develop the theory of fault-tolerance components, we allow components to exhibit Zeno behavior. However, as we will illustrate, it is important that the collection of components in a program does not exhibit Zeno behavior.

Definition 3.4 **(state predicates)** A *state predicate* S of a program $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P} \rangle$ is any subset of state space of $\mathcal{P}$ st. in the corresponding Boolean expression, clock constraints are in $\Phi(X_\mathcal{P})$, i.e., the magnitude of clock variables are nonnegative integers. ∎

Definition 3.5 **(closure)** We say that a state predicate S is *closed* in $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P} \rangle$ iff in every computation $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ in $\Pi_\mathcal{P}$, if $\sigma_j \models S$ (i.e., state predicate S holds in state σ_j), $j \in \mathbb{Z}_{\geq 0}$, then $\sigma_k \models S$, for all k, $k \geq j$. ∎

Definition 3.6 **(S-computations)** Let S be a state predicate and $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P} \rangle$ be a program. The S-computations of $\mathcal{P}$, denoted as $\mathcal{P} \mid S$, is the set of all computations in $\Pi_\mathcal{P}$ that start in a state where S is true. ∎

3.1.1 Example

We use the following example throughout the paper as a running demonstration. Consider a one-lane turn-based bridge where cars can travel in only one direction at any time. The bridge is controlled by two traffic signals and each signal changes phase from green to yellow and then to red, based on a set of timing constraints. Moreover, if one signal is red, it will turn green some time after the other signal turns red. A traffic controller program ($\mathcal{TC}$) for the bridge has two discrete variables to represent the status of the signals, i.e., $V_{\mathcal{TC}} = \{sig_0, sig_1\}$, where sig_0 and sig_1 range over $\{G, Y, R\}$. Thus, at any time, the values of sig_0 and sig_1 show in which direction cars are traveling. Moreover, for each signal, $\mathcal{TC}$ has three timers to change signal phase, i.e., $X_{\mathcal{TC}} = \{x_i, y_i, z_i \mid i = 0, 1\}$. When a signal turns green, it may turn yellow within 10 time units, but not sooner than 1 time unit. Subsequently, the signal may turn red between 1 and 2 time units after it turns yellow. Finally, when the signal is red, it may turn green within 1 time unit after *the other* signal becomes red. Both signals operate identically.

To concisely present computations of a program, we use timed guarded commands. Notice that since the set of computations of a program is suffix closed and fusion closed, the program can be written in terms of *transitions* that it can execute [5]. A *timed guarded command* (also called *timed action*) is of the form $L :: g \xrightarrow{\lambda} st$, where L is a label, g is a state predicate, st is a statement that describes how the program state is updated, and λ is a set of clock variables that are reset by execution of L. Thus, L denotes the set of transitions $\{(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0]) \mid g$ is true in state (s_0, ν), and s_1 is obtained by changing s_0 as prescribed by $st\}$. A *guarded wait command* (also called *delay action*) is of the form $L :: g \longrightarrow$ **wait**, where g identifies the set of states from where delay transitions with arbitrary durations are allowed to be taken as long as g continuously remains true.

Thus, the traffic controller program is as follows. For $i \in \{0, 1\}$:

$$
\begin{array}{lllll}
\mathcal{TC}1_i & :: & (sig_i = G) \wedge (1 \leq x_i \leq 10) & \xrightarrow{\{y_i\}} & (sig_i := Y); \\
\mathcal{TC}2_i & :: & (sig_i = Y) \wedge (1 \leq y_i \leq 2) & \xrightarrow{\{z_i\}} & (sig_i := R); \\
\mathcal{TC}3_i & :: & (sig_i = R) \wedge (z_j \leq 1) & \xrightarrow{\{x_i\}} & (sig_i := G); \\
\mathcal{TC}4_i & :: & ((sig_i = G) \wedge (x_i \leq 10)) \vee & & \\
 & & ((sig_i = Y) \wedge (y_i \leq 2)) \vee & & \\
 & & ((sig_i = R) \wedge (z_j \leq 1)) & \longrightarrow & \textbf{wait};
\end{array}
$$

where $j = (i + 1) \mod 2$. We note that the choice of execution of timed guarded commands is non-deterministic, i.e., a guarded command whose guard is true is non-deterministically chosen for execution at each time instance. Notice that the guard of $\mathcal{TC}3_i$ depends on z timer of signal j. For simplicity, we assume that once a traffic light turns green, all cars from the opposite direction have already left the bridge.

3.2 Specifications

Definition 3.7 **(specifications)** A *specification* (or *property*), denoted $SPEC$, is a tuple $\langle V_{SPEC}, X_{SPEC}, \Sigma_{SPEC} \rangle$ where V_{SPEC} is a finite set of discrete variables, X_{SPEC} is a finite set of clock variables, and Σ_{SPEC} is a suffix closed and fusion closed set of infinite computations in the state space of $SPEC$. ∎

We now define what it means for a program to *refine* a specification and what it means for a program $\mathcal{P}'$ (typically, a fault-tolerant program) to refine a program $\mathcal{P}$ (typically, a fault-intolerant program). Essentially, we would like to say that '$\mathcal{P}'$ refines $\mathcal{P}$' iff computations of $\mathcal{P}'$ are a subset of that in $\mathcal{P}$. However, if $\mathcal{P}'$ is obtained by adding fault-tolerance to $\mathcal{P}$ then $\mathcal{P}'$ may contain additional variables that are not in $\mathcal{P}$. Hence, it will be necessary to project the computations of $\mathcal{P}'$ on (the variables of) $\mathcal{P}$ and then check if the projected computation is a computation of $\mathcal{P}$.

Definition 3.8 (state projection) Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P}\rangle$ and $\mathcal{P}' = \langle V_{\mathcal{P}'}, X_{\mathcal{P}'}, \Pi_{\mathcal{P}'}\rangle$ be real-time programs st. $V_{\mathcal{P}'} = V_\mathcal{P} \cup \Delta_v$ and $X_{\mathcal{P}'} = X_\mathcal{P} \cup \Delta_x$ for some Δ_v and Δ_x. The *projection* of a state of $\mathcal{P}'$ on $\mathcal{P}$ is a state obtained by considering $V_\mathcal{P} \cup X_\mathcal{P}$ only, i.e., by abstracting away the variables in $\Delta_v \cup \Delta_x$. ∎

The same concept applies to programs and specifications. Extending this definition for computations, we say that the projection of a computation of $\mathcal{P}'$ on $\mathcal{P}$ (respectively, *SPEC*) is a computation obtained by projecting each state in that computation on $\mathcal{P}$ (respectively, *SPEC*).

Definition 3.9 (refines) Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P}\rangle$ and $\mathcal{P}' = \langle V_{\mathcal{P}'}, X_{\mathcal{P}'}, \Pi_{\mathcal{P}'}\rangle$ be real-time programs, S be a state predicate and $SPEC = \langle V_{SPEC}, X_{SPEC}, \Sigma_{SPEC}\rangle$ be a specification. We say that $\mathcal{P}'$ *refines* $\mathcal{P}$ (respectively, *SPEC*) *from* S iff the following two conditions hold: (1) S is closed in $\mathcal{P}'$, and (2) for every computation in $\Pi_{\mathcal{P}'}$ that starts in a state where S is true, the projection of that computation on $\mathcal{P}$ (respectively, *SPEC*) is a computation of $\Pi_\mathcal{P}$ (respectively, Σ_{SPEC}). ∎

In order to reason about the correctness of programs (in the absence of faults), we define the notion of program invariant.

Definition 3.10 (invariants) Let $\mathcal{P}$ be a real-time program, S be a nonempty state predicate, and *SPEC* be a specification. We say that S is an *invariant of* $\mathcal{P}$ *for SPEC* iff $\mathcal{P}$ refines *SPEC* from S. ∎

We note that our rather unconventional definition of invariant is due to the fact that in our framework, an invariant has double role. First, it specifies the closure of a program in the absence of faults. Thus, starting from a set of initial states, one possible invariant can simply be the set of reachable states. Secondly, as we will describe in Section 4, the invariant predicate also specifies a set of legitimate states which in turn determines the reachability condition of a program for recovery when faults occur.

Whenever the specification is clear from the context, we will omit it; thus, "S is an invariant of $\mathcal{P}$" abbreviates "S is an invariant of $\mathcal{P}$ for *SPEC*". Note that Definition 3.9 introduces the notion of refinement with respect to infinite computations. In case of finite computations, we characterize them by determining whether they can be extended to an infinite computation in the specification.

Definition 3.11 (maintains) Let $\mathcal{P}$ be a real-time program, S be a state predicate, and *SPEC* be a specification. We say that program $\mathcal{P}$ *maintains SPEC* from S iff (1) S is closed in $\mathcal{P}$, and (2) for all computation prefixes $\overline{\alpha}$ of $\mathcal{P}$ that start from S, there exists a computation suffix $\overline{\beta}$ st. the projection of $\overline{\alpha}\overline{\beta}$ on *SPEC* is in *SPEC*. We say that $\mathcal{P}$ *violates SPEC* from S iff it is not the case that $\mathcal{P}$ maintains *SPEC* from S. ∎

We note that if $\mathcal{P}$ refines *SPEC* from S then $\mathcal{P}$ maintains *SPEC* from S as well, but the reverse direction does not always hold. We, in particular, introduce the notion of *maintains* for computations that a (fault-intolerant) program cannot produce, but the computation can be extended to one that is in *SPEC* via adding a δ-*corrector* component to the intolerant program (see Sections 4, 6 for details).

Specifying timing constraints. In order to express time-related behaviors of real-time programs (e.g., deadlines and recovery time), we focus on a standard property typically used in real-time computing known as the *stable bounded response property*. A stable bounded response property, denoted $P \mapsto_{\leq \delta} Q$, where P and Q are two state predicates and $\delta \in \mathbb{Z}_{\geq 0}$, is the set of all computations $(\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$ in which, for all $i \geq 0$, if $\sigma_i \models P$ then there exists j, $j \geq i$, st. (1) $\sigma_j \models Q$, (2) $\tau_j - \tau_i \leq \delta$, and (3) for all k, $i \leq k < j$, $\sigma_k \models P$, i.e., it is always the case that a state in P is followed by a state in Q within δ time units and P remains true until Q becomes true. We call P the *event predicate*, Q the *response* (or *recovery*) *predicate*, and δ the *response* (or *recovery*) *time*.

Assumption 3.12 We assume that the set of clock variables of any stable bounded response property $P \mapsto_{\leq \delta} Q$ contains a special clock variable, which gets reset whenever P becomes true. This assumption is necessary to ensure that stable bounded response properties are fusion closed. ∎

The specifications considered in this paper are an intersection of a *safety* specification and a *liveness* specification [1,13]. In particular, we concentrate on a special case where the specification is the intersection of (1) *timing independent safety* characterized by a set of bad instantaneous transitions (denoted $SPEC_{\overline{bt}}$), (2) *timing dependent safety* characterized by a set of stable bounded response properties (denoted $SPEC_{\overline{br}}$), and (3) liveness.

Definition 3.13 (safety specifications)

1. (*timing independent safety*) Let $SPEC_{bt}$ be a finite set of instantaneous *bad transitions* of the form $(s_0, \nu) \to (s_1, \nu[\lambda := 0])$, where s_0 and s_1 are two locations and $\lambda \subseteq X_{SPEC}$. We denote the specification whose computations have no transition in $SPEC_{bt}$ by $SPEC_{\overline{bt}}$.
2. (*timing constraints*) We denote $SPEC_{\overline{br}}$ by the conjunction $\bigwedge_{i=0}^{m}(P_i \mapsto_{\leq \delta_i} Q_i)$, for state predicates P_i and Q_i, and, response times δ_i. ∎

Thus, given a specification *SPEC*, one can implicitly identify $SPEC_{\overline{bt}}$ and $SPEC_{\overline{br}}$ as defined above. Throughout the paper, $SPEC_{\overline{br}}$ is meant to prescribe how a program should meet its timing constraints such as providing bounded-time recovery to its normal behavior after the occurrence of faults. We formally define the notion of recovery in Section 4.

Definition 3.14 (liveness specifications) A liveness specification of *SPEC* is a set of computations that meets the following condition: for each computation prefix $\overline{\alpha}$, there exists an infinite computation $\overline{\beta}$ st. $\overline{\alpha}\overline{\beta} \in SPEC$. ∎

3.2.1 Example (cont'd)

Following Definition 3.13, the safety specification of $\mathcal{TC}$ comprises of $SPEC_{\overline{bt}_{\mathcal{TC}}}$ and $SPEC_{\overline{br}_{\mathcal{TC}}}$. $SPEC_{bt_{\mathcal{TC}}}$ is the set of transitions where both signals are not red in their target states:

$$SPEC_{bt_{\mathcal{TC}}} = \{(\sigma_0, \sigma_1) \mid (sig_0(\sigma_1) \neq R) \ \wedge \ (sig_1(\sigma_1) \neq R)\}.$$

We define $SPEC_{\overline{br}}$ of $\mathcal{TC}$ in Section 4, where we formally define the notion of bounded-time recovery.

One invariant for the program $\mathcal{TC}$ is the following:

$$\begin{array}{l} S_{\mathcal{TC}} = \forall i \in \{0,1\} \ : \\ \quad [(sig_i = G) \ \Rightarrow \ ((sig_j = R) \wedge (x_i \leq 10) \wedge (z_i > 1))] \wedge \\ \quad [(sig_i = Y) \ \Rightarrow \ ((sig_j = R) \wedge (y_i \leq 2) \wedge (z_i > 1))] \wedge \\ \quad [((sig_i = R) \wedge (sig_j = R)) \\ \qquad\qquad \Rightarrow \ ((z_i \leq 1) \oplus (z_j \leq 1))], \end{array}$$

where $j = (i+1) \mod 2$ and $\oplus$ denotes the *exclusive or* operator. It is straightforward to see that $\mathcal{TC}$ refines $SPEC_{\overline{bt}_{\mathcal{TC}}}$ from $S_{\mathcal{TC}}$.

4. FAULT MODEL AND FAULT-TOLERANCE

4.1 Fault Model

Intuitively, the faults that a program is subject to are systematically represented by the union of transitions whose execution perturbs the program state and transitions that unexpectedly advance time.

Definition 4.1 **(faults)** Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P}\rangle$ be a real-time program. The set f of *faults* is specified by the union of the following two sets:

1. a finite set f^s of *immediate faults* of the form $(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0])$ where s_0 and s_1 are two locations and λ is a subset of $X_\mathcal{P}$, and

2. a finite set f^t of *delay faults* of the form $(s, \nu) \rightarrow (s, \nu + \tau)$, which keeps the program in location s for time duration τ, $\tau \in \mathbb{R}_{\geq 0}$. ∎

Assumption 4.2 Given a real-time program $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P}\rangle$ and a set f of faults, for simplicity, we assume that no computation in $\Pi_\mathcal{P}$ contains a transition in f. ∎

We emphasize that such representation is possible notwithstanding the type of faults (be they stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), the nature of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults (be they detectable or undetectable).

Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P}\rangle$ be a program and f be a set of faults. We denote the *program $\mathcal{P}$ in the presence of f* by $\mathcal{P}[]f = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_{\mathcal{P}[]f}\rangle$. Intuitively, $\Pi_{\mathcal{P}[]f}$ is obtained by inserting transitions of f in computations of $\Pi_\mathcal{P}$. Formally, let Z be a set of computations and $\bullet$ be the operator that fuses two (finite or infinite) computations of Z st.

$$\bullet(Z) = \{\overline{\alpha}(\sigma,\tau)\overline{\beta} \mid \exists \overline{\gamma}, \overline{\psi} : (\overline{\alpha}(\sigma,\tau)\overline{\gamma} \in Z) \ \wedge \ (\overline{\psi}(\sigma,\tau)\overline{\beta} \in Z)\},$$

i.e., $\bullet(Z)$ adds additional computations obtained by inserting timed state (σ, τ) to computations in the given set. Also, let $\bullet^i(Z) = \bullet(\bullet^{i-1}(Z))$, $i \geq 1$, and $FFC(Z) = \cup_{i=1}^{\infty} \bullet^i(Z)$. Now, we define the computations of $\mathcal{P}$ in the presence of f as follows:

$$\Pi_{\mathcal{P}[]f} = \{\overline{\sigma} \mid \overline{\sigma} \in FFC(\Pi_\mathcal{P} \cup f) \ \wedge \ \overline{\sigma} \notin FFC(f)\}.$$

Just as we use invariants to show program correctness in the absence of faults, we use *fault-spans* to show the correctness of programs in the presence of faults.

Definition 4.3 **(fault-spans)** Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P}\rangle$ be a real-time program with invariant S, T be a state predicate, and f be a set of fault transitions. We say that T is an *f-span of $\mathcal{P}$ from S* iff (1) $S \subseteq T$, and (2) T is closed in $\Pi_{\mathcal{P}[]f}$. ∎

4.1.1 Example (cont'd)

$\mathcal{TC}$ is subject to clock reset faults due to circuit malfunctions. In particular, we consider faults that reset either z_0 or z_1 at any state in the invariant $S_\mathcal{TC}$ (cf. Subsection 3.2), without changing the location of $\mathcal{TC}$. Formally,

$$F_0 :: \ S_\mathcal{TC} \quad \xrightarrow{\{z_0\}} \quad \textbf{skip};$$
$$F_1 :: \ S_\mathcal{TC} \quad \xrightarrow{\{z_1\}} \quad \textbf{skip};$$

It is straightforward to see that in the presence of F_0 and F_1, $\mathcal{TC}$ may violate $SPEC_{\overline{bt}_\mathcal{TC}}$. For instance, if F_1 occurs when $\mathcal{TC}$ is in a state of $S_\mathcal{TC}$ where $(sig_0 = sig_1 = R) \wedge (z_0 \leq 1) \wedge (z_1 > 1)$, in the resulting state, we have $(sig_0 = sig_1 = R) \wedge (z_0 \leq 1) \wedge (z_1 = 0)$. From this state, immediate execution of timed actions $\mathcal{TC}3_0$ and then $\mathcal{TC}3_1$ results in a state where $(sig_0 = sig_1 = G)$, which is clearly a violation of the safety specification $SPEC_{\overline{bt}_\mathcal{TC}}$. In our traffic controller example, for simplicity, we assume that faults F_0 and F_1 only occur in a state where $S_\mathcal{TC}$ holds (i.e., faults do not re-occur outside the invariant of $\mathcal{TC}$).

4.2 Fault-Tolerance

We now define what we mean by *fault-tolerance* in the context of real-time programs. Obviously, in the absence of faults, a program should refine its specification. In the presence of faults, however, it may not refine its specification and, hence, it may refine some 'tolerance specification'. These specifications are based on refinement of a combination of (timing independent) safety, liveness, timing constraints, and a desirable bounded-time recovery mechanism in the presence of faults. The resulting tolerance specification with respect to each combination, defines a *level of fault-tolerance.* In this paper, we focus on the strongest level, known as *hard-masking* fault-tolerance [6]. Intuitively, given a specification $SPEC$, the hard-masking tolerance specification of $SPEC$ is identical to $SPEC$. In other words, the occurrence of all faults are *masked.* Moreover, we require $SPEC$ to prescribe a bounded-time recovery mechanism.

Definition 4.4 **(hard-masking tolerance specification)** Let $SPEC = \langle V_{SPEC}, X_{SPEC}, \Sigma_{SPEC}\rangle$ be a specification where $SPEC \Rightarrow (\neg R \mapsto_{\leq \theta} R)$ for some recovery predicate R and some recovery time $\theta \in \mathbb{Z}_{\geq 0}$. The *hard-masking* tolerance specification of $SPEC$ is $SPEC$. ∎

We are now ready to define what it means for a program to be hard-masking f-tolerant. With the intuition that a program is hard-masking f-tolerant to $SPEC$ if it refines $SPEC$ in the absence of faults and it refines the hard-masking tolerance specification of $SPEC$ in the presence of f, we define 'hard-masking f-tolerant to $SPEC$ from S' as follows.

Definition 4.5 **(hard-masking programs)** Let $\mathcal{P}$ be a real-time program with invariant S, f be a set of fault transitions, $SPEC$ be a specification, and θ be a nonnegative integer. We say that $\mathcal{P}$ is *hard-masking f-tolerant to $SPEC$ with recovery time θ from S* iff the following two conditions hold:

- $\mathcal{P}$ refines $SPEC$ from S, and
- there exists T st. $T \supseteq S$ and $\mathcal{P}[]f$ refines the hard-masking tolerance specification of $SPEC$ for recovery time θ and recovery predicate S from T. ∎

4.2.1 Example (cont'd)

Thus, the fault-tolerant version of $\mathcal{TC}$ has to, first, reach a state where both signals are red and subsequently recover to $S_\mathcal{TC}$ where exactly one signal turns green. To this end, we specify the following stable bounded-response properties:

$$SPEC_{\overline{br}_\mathcal{TC}} \equiv (\neg S_\mathcal{TC} \mapsto_{\leq 7} S_\mathcal{TC}) \ \wedge \ (\neg S_\mathcal{TC} \mapsto_{\leq 3} Q_\mathcal{TC}),$$

where $Q_{\mathcal{TC}} = \forall i \in \{0,1\} \;:\; ((sig_i = R) \;\wedge\; (z_i > 1))$. The response times in $SPEC_{\overline{br}_{\mathcal{TC}}}$ are simply two arbitrary numbers for illustration.

Below, we present a hard-masking version of our traffic controller program, denoted by $\mathcal{TC}'$, where $i \in \{0,1\}$ and $j = (i+1) \mod 2$:

$$\begin{array}{lllll}
\mathcal{TC}'1_i:: & (sig_i = G) \;\wedge\; (1 \leq x_i \leq 10) & \xrightarrow{\{y_i\}} & (sig_i := Y); \\
\mathcal{TC}'2_i:: & (sig_i = Y) \;\wedge\; (1 \leq y_i \leq 2) & \xrightarrow{\{z_i\}} & (sig_i := R); \\
\mathcal{TC}'3_i:: & (sig_i = R) \;\wedge\; (z_j \leq 1) \;\wedge & & \\
 & \qquad (sig_j = R) & \xrightarrow{\{x_i\}} & (sig_i := G); \\
\mathcal{TC}'4_i:: & ((sig_i = G) \;\wedge\; (x_i \leq 10)) \;\vee & & \\
 & ((sig_i = Y) \;\wedge\; (y_i \leq 2)) \;\vee & & \\
 & ((sig_i = R) \;\wedge\; (z_j \leq 1)) & \longrightarrow & \textbf{wait}; \\
\mathcal{TC}'5_i:: & (sig_i \neq R \;\vee\; sig_j \neq R) \;\wedge\; (z_i, z_j \leq 1) & & \\
 & & \longrightarrow & sig_i, sig_j := R; \\
\mathcal{TC}'6_i:: & (sig_i = sig_j = R) \;\wedge\; (z_i, z_j \leq 4) \;\wedge & & \\
 & \qquad (t_1 \leq 3) & \longrightarrow & \textbf{wait}; \\
\mathcal{TC}'7_i:: & (sig_i = sig_j = R) \;\wedge\; (z_i, z_j > 1) & & \\
 & & \xrightarrow{\{z_i\}} & \textbf{skip}; \\
\mathcal{TC}'8_i:: & (sig_i = sig_j = R) \;\wedge\; (z_i, z_j > 1) \;\wedge & & \\
 & \qquad (t_2 \leq 7) & \longrightarrow & \textbf{wait};
\end{array}$$

where t_1 and t_2 are the special clock variables that accompany stable bounded response properties in $SPEC_{\overline{br}_{\mathcal{TC}}}$ (cf. Assumption 3.12). In Sections 5 and 6, we demonstrate how we decompose $\mathcal{TC}'$ into $\mathcal{TC}$ and its fault-tolerance components.

5. DETECTORS AND THEIR ROLE IN HARD-MASKING PROGRAMS

This section is organized as follows. We formally introduce the notion of detector components in Subsection 5.1. In Subsection 5.2, we precisely define what we mean by containment of a detector in a real-time program. Then, we present detector components of the hard-masking version of our traffic controller in Subsection 5.3. Finally, in Subsection 5.4, we develop the theory of detectors by proving the necessity of existence of hard-masking detectors in hard-masking fault-tolerant programs.

5.1 Detectors

Intuitively, a detector is a program component that ensures satisfaction of timing independent safety (i.e., $SPEC_{\overline{bt}}$ in Definition 3.13).

Definition 5.1 (detects) Let W and D be state predicates. Let 'W *detects* D' be the specification, that is the set of all infinite computations $\overline{\sigma} = (\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$, satisfying the following three conditions:

- (*Safeness*) For all $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models W$ then $\sigma_i \models D$. (In other words, $\sigma_i \models (W \Rightarrow D)$.)
- (*Progress*) For all $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models D$ then there exists $k, k \geq i$, st. $\sigma_k \models W$ or $\sigma_k \not\models D$.
- (*Stability*) There exists $i \in \mathbb{Z}_{\geq 0}$, st. for all j, $j \geq i$, if $\sigma_j \models W$ then $\sigma_{j+1} \models W$ or $\sigma_{j+1} \not\models D$. ∎

Definition 5.2 (detectors) Let $\mathcal{D}$ be a program and D, W, and U be state predicates of $\mathcal{D}$. We say that W *detects* D *in* $\mathcal{D}$ *from* U (i.e., $\mathcal{D}$ is a *detector*) iff $\mathcal{D}$ refines 'W detects D' from U. ∎

A detector $\mathcal{D} = \langle V_{\mathcal{D}}, X_{\mathcal{D}}, \Pi_{\mathcal{D}} \rangle$ is used to check whether its "detection predicate", D, is true. Since $\mathcal{D}$ satisfies *Progress* from U, in any computation in $\Pi_{\mathcal{D}}$, if $U \wedge D$ is true continuously, $\mathcal{D}$ eventually detects this fact and makes W true. Since $\mathcal{D}$ satisfies *Safeness* from U, it follows that $\mathcal{D}$ never lets W witness D incorrectly. Moreover, since $\mathcal{D}$ satisfies *Stability* from U, it follows that once W becomes true, it continues to be true unless D is falsified. In the context of fault-tolerance, D is typically a predicate of the fault-intolerant program from where safety should be always satisfied and W is a predicate of the fault-tolerant program that witnesses the detection of D.

In order to analyze the behavior of a detector in the presence of faults, we consider the notion of hard-masking tolerant detectors. More specifically, a detector $\mathcal{D}$ is a *hard-masking tolerant detector* if $SPEC$ is substituted by 'W detects D' in Definition 4.5.

5.2 Containment of Detectors in Real-Time Programs

In order to show the existence of detectors in hard-masking fault-tolerant programs, we would like to show that the program contains a detector for a detection predicate associated with the fault-intolerant program. However, we need to identify syntactic characteristics of a program before detection predicates can be identified. In particular, since a detector is used to ensure that the execution of an action is safe, its witness predicate must be *used* by the fault-tolerant program. Intuitively, the syntactic constraints identified in this section require the witness predicate to be a guard of the corresponding action in the fault-tolerant program.

In order to accomplish our goal, first, we show that violation of timing independent safety (i.e., $SPEC_{\overline{bt}}$ in Definition 3.13) can be merely determined by considering transitions in $SPEC_{bt}$.

Lemma 5.3 *Let SPEC be a specification, $\overline{\alpha}$ be a computation prefix, σ and σ' be two states, and $\tau, \tau' \in \mathbb{R}_{\geq 0}$, where $\tau = \tau'$.*
If

- *$\overline{\alpha}(\sigma, \tau)$ maintains $SPEC_{\overline{bt}}$*

then

- *$\overline{\alpha}(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$ iff $(\sigma, \tau)(\sigma', \tau')$ maintains $SPEC_{\overline{bt}}$* . ∎

In Lemma 5.4, we show that there exists a set of states from where execution of programs maintains $SPEC_{\overline{bt}}$. We call such a state predicate a *detection predicate* for $SPEC_{\overline{bt}}$.

Lemma 5.4 *Given a program $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, \Pi_{\mathcal{P}} \rangle$, there exists a state predicate D (called* detection predicate*) st. all computations of $\Pi_{\mathcal{P}}$ that start from D maintain $SPEC_{\overline{bt}}$.* ∎

We now prove the uniqueness of the weakest detection predicate for a given program $\mathcal{P}$.

Lemma 5.5 *Given a program $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, \Pi_{\mathcal{P}} \rangle$ and a specification SPEC, there exists a unique weakest detection predicate of $\mathcal{P}$ for $SPEC_{\overline{bt}}$.* ∎

We are now ready to define what it means for a program to *contain* detectors. As mentioned in Subsection 3.1.1, since the set of computations of a program is suffix closed and fusion closed, the program can be written in terms of timed guarded commands. Given a timed guarded command, say L :: $\mathit{Guard} \xrightarrow{\lambda} \mathit{st}$, Lemma 5.5 shows that there exists a unique weakest detection predicate, say wdp, from where execution of L does not violate $\mathit{SPEC}_{\overline{bt}}$. Hence, to show the existence of detectors, we require the detection predicate of such a timed action to be $\mathit{Guard} \wedge \mathit{wdp}$. Furthermore, to show that the fault-tolerant program contains the desired detector, we show that it must be using the witness predicate of that detector to ensure that execution of the corresponding timed action is safe. Towards this end, we define the notion of *encapsulation*. Intuitively, if (typically, a fault-tolerant) program $\mathcal{P}'$ encapsulates (typically, a fault-intolerant) program $\mathcal{P}$ then for each timed action of $\mathcal{P}$ of the form $\mathit{Guard} \xrightarrow{\lambda} \mathit{st}$, $\mathcal{P}'$ contains a timed action of the form $\mathit{Guard} \wedge \mathit{Guard}' \xrightarrow{\lambda \cup \lambda'} \mathit{st}||\mathit{st}'$. The semantic of $\mathit{st}||\mathit{st}'$ corresponds to the statement where st and st' are executed simultaneously, clock variables in $\lambda \cup \lambda'$ are reset, and the timed action is executed only when its guard, $\mathit{Guard} \wedge g'$, is true. In other words, $\mathcal{P}'$ has a timed action corresponding to each timed action of $\mathcal{P}$ (possibly) with a stronger guard, additional assignments in st', and additional clock variables in λ'. Notice that the assignments in st' and clock variables in λ' may be added in order to add fault-tolerance to $\mathcal{P}$ (cf. the notion of projection in Subsection 3.2). To show that $\mathcal{P}'$ is using a detector for a timed action of $\mathcal{P}$, we require the witness predicate of that detector to be $\mathit{Guard} \wedge \mathit{Guard}'$ which is the guard of the corresponding timed action in $\mathcal{P}'$.

Definition 5.6 (encapsulates) Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, \Pi_\mathcal{P}\rangle$ and $\mathcal{P}' = \langle V_{\mathcal{P}'}, X_{\mathcal{P}'}, \Pi_{\mathcal{P}'}\rangle$ be two real-time programs and S be a state predicate. We say that $\mathcal{P}'$ *encapsulates* $\mathcal{P}$ *from* S iff each timed action in $\mathcal{P}'$ that is enabled in a state in S and that updates variables in $V_\mathcal{P}$ is of the form $\mathit{Guard} \wedge g' \xrightarrow{\lambda \cup \lambda'} \mathit{st}||\mathit{st}'$, where $\mathit{Guard} \xrightarrow{\lambda} \mathit{st}$ is a timed action of $\mathcal{P}$ and st' does not update variables in $V_\mathcal{P}$ and $\lambda' \cap X_\mathcal{P} = \{\}$. ∎

Based on the above discussion, given a timed guarded command of the form $\mathit{Guard} \xrightarrow{\lambda} \mathit{st}$ of $\mathcal{P}$, its (weakest) detection predicate wdp and the corresponding action $\mathit{Guard} \wedge \mathit{Guard}' \xrightarrow{\lambda \cup \lambda'} \mathit{st}||\mathit{st}'$ of $\mathcal{P}'$, we require the detection predicate of the desired detector to be $\mathit{Guard} \wedge \mathit{wdp}$ and the witness predicate of the desired detector to be $\mathit{Guard} \wedge \mathit{Guard}'$.

Finally, in order to formalize the notion of containment and existence of detectors, we need to define what it means to obtain a fault-tolerant program by *reusing* its fault-intolerant version.

Definition 5.7 (reuses) Let $\mathcal{P}$ and $\mathcal{P}'$ be two real-time programs. We say that $\mathcal{P}'$ *reuses* $\mathcal{P}$ *from* S iff the following two conditions are satisfied:

- $\mathcal{P}'$ refines $\mathcal{P}$ from S, and
- $\mathcal{P}'$ encapsulates $\mathcal{P}$ from S. ∎

5.3 Example (cont'd)

It is straightforward to see that the weakest detection predicate for $\mathcal{TC}_i$ is:

$$\mathit{wdp}_{\mathcal{TC}'_i} = \{\sigma \mid \mathit{sig}_j(\sigma) = R\},$$

where $i \in \{0, 1\}$ and $j = (i + 1) \mod 2$. Thus, in program $\mathcal{TC}'$ (cf. Subsection 4.2), the guard of $\mathcal{TC}3_i$ is strengthened in order for $\mathcal{TC}'$ to refine $\mathit{SPEC}_{\overline{bt}_{\mathcal{TC}}}$ in the presence of faults. Intuitively, $\mathcal{TC}'$ is allowed to change phase from red to green only when the other signal is red. More precisely, $\mathcal{TC}'$ uses the detector $\mathcal{D}_{\mathcal{TC}'_i}$ which consists of timed guarded commands $\mathcal{TC}'1_i$, $\mathcal{TC}'2_i$, and $\mathcal{TC}'4_i$ with the following detection and witness predicates:

$$D_{\mathcal{TC}'_i} = \mathit{guard}(\mathcal{TC}3_i) \wedge \mathit{wdp}_{\mathcal{TC}'_i}$$
$$W_{\mathcal{TC}'_i} = \mathit{guard}(\mathcal{TC}'3_i).$$

It is easy to see that that $W_{\mathcal{TC}'_i}$ detects $D_{\mathcal{TC}'_i}$ in $\mathcal{D}_{\mathcal{TC}'_i}$ from $S_{\mathcal{TC}}$ in both absence and presence of F_0 and F_1. Notice that $\mathcal{D}_{\mathcal{TC}'_i}$ exhibits Zeno behavior since when the witness predicate becomes true, there does not exist a timed guarded command whose guard is enabled except $\mathcal{TC}'4_i$. However, it is important that the entire program does not show Zeno behavior. For instance, one can observe that, $\mathcal{TC}'3_i$ ensures time progress for $\mathcal{TC}'$.

5.4 The Necessity of Existence of Detectors in Hard-Masking Programs

Based on the formalization of the notion of containment, we are now ready to prove that hard-masking programs contain hard-masking tolerant detectors. Our strategy to accomplish our goal is as follows. First, based on Definitions 5.6 and 5.7, we show that if a program refines $\mathit{SPEC}_{\overline{bt}}$ *in the absence of faults* then it contains detectors. The intuition is that if program $\mathcal{P}'$ is designed by transforming $\mathcal{P}$ so as to refine $\mathit{SPEC}_{\overline{bt}}$, then the transformation must have added detectors for $\mathcal{P}$, and $\mathcal{P}'$ reuses $\mathcal{P}$. We formulate this in Claim 5.10. Then (*in the presence of faults*), using Claim 5.10, we show that if a hard-masking program $\mathcal{P}'$ is designed by reusing $\mathcal{P}$ to tolerate a set f of faults, $\mathcal{P}'$ contains a hard-masking tolerant detector for each action of $\mathcal{P}$. This is shown in Theorem 5.11.

In order to show that a program contains a detector component, we are required to show that the corresponding timed guarded commands satisfy the *Progress* condition of Definition 5.1. Thus, we assume that programs need to satisfy the following *fairness* condition.

Assumption 5.8 We assume that program computations are *fair* in the sense that in every computation, if the guard of an action is continuously true then that action is eventually chosen for execution. ∎

Assumption 5.9 Without loss of generality, for simplicity, we assume that transitions that correspond to different actions of the program are mutually disjoint, i.e., they do not contain overlapping transitions. The results in this paper are valid without this assumption since we can easily modify a given program to one that satisfies this assumption. ∎

We are now ready to formulate our claim on existence of detectors in programs that refine $\mathit{SPEC}_{\overline{bt}}$ in the absence of faults.

Claim 5.10 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, S be a nonempty state predicate, and SPEC be a specification.*
If

- *$\mathcal{P}'$ reuses $\mathcal{P}$ from S, and*
- *$\mathcal{P}'$ refines $\mathit{SPEC}_{\overline{bt}}$ from S,*

then

- *$(\forall ac \mid ac$ is a timed action of $\mathcal{P} : \mathcal{P}'$ contains a detector of a detection predicate of ac for $\mathit{SPEC})$.* ∎

Now, we show that if a hard-masking f-tolerant program $\mathcal{P}'$ is designed by reusing $\mathcal{P}$ then $\mathcal{P}'$ contains a hard-masking tolerant detector for each action in $\mathcal{P}$.

Theorem 5.11 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, S be a nonempty state predicate, f be a set of faults, and $SPEC$ be a specification.*
If

- *$\mathcal{P}$ refines $SPEC_{\overline{bt}}$ from S,*
- *$\mathcal{P}'$ reuses $\mathcal{P}$ from R, where $R \Rightarrow S$ for some nonempty state predicate R, and*
- *$\mathcal{P}'$ is hard-masking f-tolerant to $SPEC$ from R*

then

- *$(\forall ac \mid ac$ is a timed action of $\mathcal{P}$: $\mathcal{P}'$ is a hard-masking f-tolerant detector of a detection predicate of ac for $SPEC)$.* ∎

6. δ-CORRECTORS AND THEIR ROLE IN HARD-MASKING PROGRAMS

This section is organized as follows. We formally introduce the notion of weak and strong δ-corrector components in Subsection 6.1. In Subsection 6.2, we define what we mean by containment of a δ-corrector in a real-time program. Then, we present δ-corrector components of the hard-masking version of our traffic controller in Subsection 6.3. Finally, in Subsections 6.4 and 6.5, we develop the theory of strong and weak δ-correctors, respectively, by proving the necessity of existence of hard-masking weak and strong δ-correctors in hard-masking fault-tolerant programs.

6.1 Weak and Strong δ-Correctors

Intuitively, a δ-corrector is a program component that ensures *bounded-time recovery* to a *correction predicate*. In fault-tolerant computing, recovery is essential to guarantee that liveness properties (cf. Definition 3.14) and timing constraints (cf. $SPEC_{\overline{br}}$ in Definition 3.13) are met where the state of a program is perturbed by the occurrence of faults. Depending upon the closure of the correction predicate in δ-correctors, they are classified into *weak* and *strong*.

Definition 6.1 (weakly corrects) Let C and W be state predicates. Let 'W *weakly corrects* C *within* δ' be the specification, that is the set of all infinite computations $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$, satisfying the following conditions:

- (*[Weak] Convergence*) There exists $i \in \mathbb{Z}_{\geq 0}$, st. $\sigma_i \models C$ and $(\tau_i - \tau_0) \leq \delta$.
- (*Safeness*) For all $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models W$ then $\sigma_i \models C$.
- (*Progress*) For all $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models C$ then there exists $k, k \geq i$, st. $\sigma_k \models W$ or $\sigma_k \not\models C$.
- (*Stability*) There exists $i \in \mathbb{Z}_{\geq 0}$, st. for all j, $j \geq i$, if $\sigma_j \models W$ then $\sigma_{j+1} \models W$ or $\sigma_{j+1} \not\models C$. ∎

Definition 6.2 (strongly corrects) Let C and W be state predicates. Let 'W *strongly corrects* C *within* δ' be the specification, that is the set of all infinite computations $\overline{\sigma}$, satisfying the following two conditions:

- W weakly corrects C within δ, and
- (*[Strong] Convergence*) In addition to Weak Convergence, C is closed in $\overline{\sigma}$. ∎

Definition 6.3 (δ-correctors) Let $\mathcal{C}$ be a program and C, W, and U be state predicates of $\mathcal{C}$. We say that W *weakly/strongly corrects C in $\mathcal{C}$ from U* (i.e., $\mathcal{C}$ is a weak/strong *δ-corrector*) iff $\mathcal{C}$ refines 'W weakly/strongly corrects C within δ' from U. ∎

Similar to the concept of tolerant detectors, in order to analyze the behavior of a δ-corrector $\mathcal{C}$ in the presence of faults, we consider the notion of hard-masking tolerant δ-correctors. More specifically, a weak/strong δ-corrector $\mathcal{C}$ is a *hard-masking tolerant weak/strong δ-corrector* if $SPEC$ is substituted by 'W weakly/strongly corrects C within δ' in Definition 4.5.

Notice that since $\mathcal{C}$ satisfies Weak (respectively, Strong) Convergence from U, it follows that $\mathcal{C}$ reaches a state where C becomes true within δ time units (and, respectively, C continues to be true thereafter). In addition to convergence, a δ-corrector never lets the predicate W witness the correction predicate C incorrectly, as $\mathcal{C}$ satisfies Safeness from U. Moreover, since $\mathcal{C}$ satisfies Progress from U, it follows that W eventually becomes true. And, finally, since $\mathcal{C}$ satisfies Stability from U, it follows that when W becomes true, W is never falsified.

6.2 Containment of δ-Correctors in Real-Time Programs

As mentioned earlier, in the context of real-time programs, δ-correctors ensure bounded-time recovery to their correction predicate. Intuitively, we will use weak δ-correctors where we need refinement of stable bounded response properties in the presence of faults. In such properties, when the event predicate becomes true, the program needs to reach a state where the response predicate holds within the respective recovery time. Nonetheless, the program does not need to remain in the response predicate.

Unlike weak δ-correctors, we will use strong δ-correctors where we need bounded-time recovery to a state predicate in which the program is required to stay in. The correction predicate of a δ-corrector $\mathcal{C}$ is typically an invariant predicate of the fault-intolerant program while the witness predicate witnesses the correction of the correction predicate. This is obviously due to the fact that real-time programs are closed in their invariant predicate. Existence of strong δ-correctors are of special interest, since recovery to the invariant predicate automatically ensures refinement of the liveness specification. In particular, in Subsection 6.4, we show the necessity of existence of strong δ-correctors in hard-masking programs in order to refine the property $\neg S \mapsto_{\leq \theta} S$ (cf. Definition 4.4), where S is in invariant predicate.

In terms of the behavior of δ-correctors, observe that in Definition 6.1 (and, hence, Definition 6.2 as well), state σ_0 is the earliest state from where recovery must commence. Thus, τ_i is the time instance where correction is complete and $\tau_i - \tau_0$ is the duration of correction. In case of strong δ-correctors, σ_0 is also the earliest state reached outside the invariant due to occurrence of faults.

We note that although detectors and δ-correctors share three identical constraints, their semantics of containment are completely different. Intuitively, a detector uses the witness predicate in order to detect whether program execution is safe. Hence, as we developed the theory of detectors in Section 5, we imposed constraints that require the witness predicate to be *used*. To the contrary, a program may not use the witness predicate of a δ-corrector, as a fault-tolerant program may not need to know *when* correction is complete.

6.3 Example (cont'd)

Continuing with our traffic controller example, we identify δ-correctors for each stable bounded-response property in $SPEC_{\overline{br}_{\mathcal{TC}}}$ introduced in Subsection 4.2. To this end, first, consider the property $\neg S_{\mathcal{TC}} \mapsto_{\leq 3} Q$. When $\mathcal{TC}[]\{F_0, F_1\}$ reaches a state in $\neg S_{\mathcal{TC}} \wedge \neg Q$ where at least one signal is red and the value of both z timers is less than or equal to 1, it needs to recover to Q within 3 time units. Let $\mathcal{C}^1_{\mathcal{TC}'}$ be the weak 3-corrector in $\mathcal{TC}'$ consisting of timed actions $\mathcal{TC}'5_i$ and $\mathcal{TC}'6_i$ with correction and witness predicates both equal to Q. Intuitively, $\mathcal{C}^1_{\mathcal{TC}'}$ ensures that when $\mathcal{TC}'$ is in a state outside the invariant, it reaches a state where both signals are red within 3 time units.

Likewise, for the property $\neg S_{\mathcal{TC}} \mapsto_{\leq 7} S_{\mathcal{TC}}$, let $\mathcal{C}^2_{\mathcal{TC}}$ be the strong 7-corrector consisting of timed actions $\mathcal{TC}'7_i$ and $\mathcal{TC}'8_i$ with witness and corrections predicates equal to $S_{\mathcal{TC}}$. In $\mathcal{C}^2_{\mathcal{TC}'}$, a z timer gets reset when the state of $\mathcal{TC}'$ is in $\neg S_{\mathcal{TC}} \wedge Q$ within 7 time units since the occurrence of a fault. Such a reset takes the traffic controller back to its invariant predicate $S_{\mathcal{TC}}$ where timed action $\mathcal{TC}1_i$ is enabled. Notice that unlike the the entire program $\mathcal{TC}'$, both $\mathcal{C}^1_{\mathcal{TC}'}$ and $\mathcal{C}^2_{\mathcal{TC}'}$ exhibit Zeno behavior when running individually.

6.4 The Necessity of Existence of Strong δ-Correctors in Hard-Masking Programs

We are now ready to prove that hard-masking programs contain hard-masking tolerant strong δ-correctors. Our strategy to accomplish our goal is as follows. First, in Claim 6.5, we show that *in the absence of faults*, if a program refines a specification within θ time units then it contains strong δ-correctors for some $\delta \in \mathbb{Z}_{\geq 0}$. Then, (*in the presence of faults*), using Claim 6.5, we show that if a hard-masking program $\mathcal{P}'$ is designed by reusing $\mathcal{P}$ to tolerate a set f of faults, $\mathcal{P}'$ contains a hard-masking tolerant strong δ-corrector. This is shown in Theorem 6.8.

Notation. For simplicity, we use the pseudo-arithmetic expressions to denote timing constraints over finite computations. For instance, $\overline{\sigma}_{\leq \delta}$, denotes a finite computation $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ that satisfies the timing constraint $\tau_n - \tau_0 \leq \delta$, where $\delta \in \mathbb{Z}_{\geq 0}$. We use S^* to denote a finite computation $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ st. $\sigma_i \models S$ for all i, $0 \leq i \leq n$. Thus, $(true)^*_{\leq \infty}$ denotes an arbitrary finite computation with no specific time bound.

Definition 6.4 (becomes) Let $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, \Pi_{\mathcal{P}} \rangle$ and $\mathcal{P}' = \langle V_{\mathcal{P}'}, X_{\mathcal{P}'}, \Pi_{\mathcal{P}'} \rangle$ be two real-time programs. We say that $\mathcal{P}'$ *becomes* $\mathcal{P}$ *within* θ *from* T iff $\mathcal{P}'$ refines $(true)^*_{\leq \theta} \Pi_{\mathcal{P}}$ from T. ∎

In Claim 6.5, we show that given a program $\mathcal{P}$, state predicate S, and specification $SPEC$, where $\mathcal{P}$ refines $SPEC$ from S, if a program $\mathcal{P}'$ is designed st. it behaves like $\mathcal{P}$ within θ and, thus, has a suffix in $SPEC$, then $\mathcal{P}'$ is a strong δ-corrector of an invariant predicate of $\mathcal{P}$ for some $\delta \in \mathbb{Z}_{\geq 0}$. We prove this Claim by showing that $\mathcal{P}'$ itself refines the required strong δ-corrector specification.

Claim 6.5 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, S and T be nonempty state predicates, SPEC be a specification, and θ be a nonnegative integer.*
If

- *$\mathcal{P}$ refines SPEC from S,*
- *$\mathcal{P}'$ refines $\mathcal{P}$ from S, and*
- *$\mathcal{P}'$ becomes $\mathcal{P}|S$ within θ from T,*

then

- *there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is a strong δ-corrector of S.* ∎

The next lemma generalizes Claim 6.5. In general, given a program $\mathcal{P}$ that refines $SPEC$ from S, $\mathcal{P}'$ may not behave like $\mathcal{P}$ from each state in S but only from a subset of S, say R. This may happen, for example, if $\mathcal{P}'$ contains additional variables and $\mathcal{P}'$ behaves like $\mathcal{P}$ only after the values of these additional variables are restored. Lemma 6.6 shows that in such a case, $\mathcal{P}'$ contains a hard-masking tolerant strong δ-corrector of an invariant predicate of $\mathcal{P}$. The strong δ-corrector is hard-masking in the sense that the correction predicate is preserved only after $\mathcal{P}'$ reaches a state where R is true.

Lemma 6.6 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, R, S, and T be nonempty state predicates where $R \Rightarrow S$, SPEC be a specification, and θ be a nonnegative integer.*
If

- *$\mathcal{P}$ refines SPEC from S,*
- *$\mathcal{P}'$ refines $\mathcal{P}$ from R, and*
- *$\mathcal{P}'$ becomes $(\mathcal{P}|R)$ within θ from T,*

then

- *there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is a hard-masking strong δ-corrector with recovery time θ of S.* ∎

We now illustrate the role of strong δ-correctors in hard-masking programs in the presence of faults. In particular, we use Claim 6.5 and Lemma 6.6 to show that if a hard-masking f-tolerant program $\mathcal{P}'$ with recovery time θ is designed by reusing $\mathcal{P}$ then there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ contains a hard-masking f-tolerant strong δ-corrector with recovery time θ for an invariant predicate of $\mathcal{P}$. Notice that, since our goal is to identify components that provide *bounded-time* recovery in the presence of faults, there needs to be some bound on the number of occurrence of faults. In fact, it is straightforward to show that providing bounded-time recovery in the presence of unbounded occurrence of faults is generally impossible.

Assumption 6.7 Let $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, \Pi_{\mathcal{P}} \rangle$ be a program with invariant predicate S and f be a set of faults. Also, let $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ be a computation prefix in $\Pi_{\mathcal{P}[]f}$ where $\sigma_0 \models S$ and $\sigma_i \not\models S$, $1 \leq i \leq n$. We assume that the number of occurrence of faults between states σ_1 and σ_n is at most k for some $k \in \mathbb{Z}_{\geq 0}$. ∎

Theorem 6.8 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, R and S be nonempty state predicates, SPEC be a specification, and θ be a nonnegative integer.*
If

- *$\mathcal{P}$ refines SPEC from S,*
- *$\mathcal{P}'$ is hard-masking f-tolerant for SPEC from R, where $R \Rightarrow S$,*
- *$\mathcal{P}'$ refines $\mathcal{P}$ from R, and*
- *$\mathcal{P}'$ becomes $\mathcal{P}|R$ within θ from T, where T is an f-span of $\mathcal{P}'$,*

then

- *there exist δ and θ' in $\mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is a hard-masking f-tolerant strong δ-corrector with recovery time θ' of S.* ∎

6.5 The Necessity of Existence of Weak δ-Correctors in Hard-Masking Programs

Just like the relation between recovery constraint and strong δ-correctors in hard masking programs, we show that if a hard-masking f-tolerant program $\mathcal{P}'$ is designed by reusing program $\mathcal{P}$ then $\mathcal{P}'$ contains a hard-masking tolerant weak δ-corrector for each stable bounded response property in $SPEC_{\overline{br}}$.

Theorem 6.9 *Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, R and S be nonempty state predicates where $R \Rightarrow S$, $SPEC$ be a specification, and θ be a nonnegative integer.*
If

- *$\mathcal{P}$ refines $SPEC$ from S,*
- *$\mathcal{P}'$ reuses $\mathcal{P}$ from R,*
- *$\mathcal{P}'$ is hard-masking f-tolerant to $SPEC$ from R*

then

- *$(\forall i \mid 0 \leq i \leq m$: there exists $\delta \in \mathbb{Z}_{\geq 0}$ st. $\mathcal{P}'$ is a hard-masking tolerant weak δ-corrector for the response predicate of stable bounded response property $P_i \mapsto_{\leq \delta_i} Q_i$ of $SPEC_{\overline{br}})$.* ∎

7. CONCLUSION AND FUTURE WORK

In this paper, we focused on a theory of fault-tolerance components in the context of hard-masking real-time programs. Our main contributions are we (1) identified three types of fault-tolerance components, namely, *detectors*, *weak δ-correctors*, and *strong δ-correctors*, (2) formally defined the notion of containment of components in real-time programs, and (3) showed that every hard-masking program can be decomposed into its fault-intolerant version and a collection of fault-tolerance components.

The significance of this work follows from the fact that detectors and correctors have been found to be useful for analysis of several fault-tolerant programs. Examples include Byzantine agreement, mutual exclusion, tree maintenance, leader election, termination detection, and bounded network management [10,16,18]. This work shows that such analysis is possible for all fault-tolerant programs that are designed from their fault-intolerant version. This work complements the results in [5] in that while [5] is focused on a design methodology, the current work focuses on analysis.

We are currently focusing on using these components in the context of automated addition of fault-tolerance to embedded systems. In particular, by designing and verifying these components in advance, it would be possible to speed up automated addition of fault-tolerance to real-time programs. Towards this end, we are working on extending our current work on automated addition of fault-tolerance [6, 7] to use pre-synthesized detectors and δ-correctors.

8. REFERENCES

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

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. *International Journal on Software Tools for Technology Transfer*, 1(1-2):86–109, 1997.

[4] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, 1998.

[5] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 436–443, 1998.

[6] B. Bonakdarpour and S. S. Kulkarni. Incremental synthesis of fault-tolerant real-time programs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 4280, pages 122–136, 2006.

[7] B. Bonakdarpour and S. S. Kulkarni. Masking faults while providing bounded-time phased recovery. In *International Symposium on Formal Methods (FM)*, pages 374–389, 2008.

[8] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[9] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[10] A. Ebnenasir and B. H. C. Cheng. *Architecting Dependable Systems IV*, chapter A Pattern-Based Approach for Modeling and Analyzing Error Recovery, pages 115–141. Springer Berlin / Heidelberg, 2007.

[11] F. C. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In *FORMATS/FTRTFT*, pages 183–198, 2004.

[12] S. Ghosh and X. He. Fault-containing self-stabilization using priority scheduling. *Information Processing Letters*, 73(3–4):145–151, 2000.

[13] T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.

[14] A. Jhumka, F. Gartner, C. Fetzer, and N. Suri. On systematic design of fast and perfect detectors. Technical Report 200263, School of Computer and Communication Sciences, EPFL, 2002.

[15] A. Jhumka, M. Hiller, and N. Suri. Component-based synthesis of dependable embedded software. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 111–128, 2002.

[16] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[17] S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 209–219, 2004.

[18] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In *Internationa Workshop on Self-Stabilization (WSS)*, pages 33–40, June 1999.

[19] J. Rushby. An overview of formal verification for the time-triggered architecture. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 83–105, 2002.

[20] O. Theel and F. Gartner. An exercise in proving convergence through transfer functions. In *Workshop on Self-Stabilizing Systems (SSS)*, pages 41–47, 1999.