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
Then, in Section 5 (respectively, 6), we present the notion of
deﬁne what it means for a fault-tolerant program to
is not important in the context of a design methodology and,
hence, is not formalized in [5].

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 pre-
cisely 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 conclu-
sion that the predicate being detected is true) are syntacti-
cally 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, their signiﬁ-
cance 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 com-
ponents would signiﬁcantly assist in analysis of real-time
embedded fault-tolerant programs.

**Organization.** First, we present related work in Sec-
tion 2. In Section 3, we formally deﬁne real-time programs
and speciﬁcations. Section 4 is dedicated to present our
decomposition. In Sections 5 and 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 tolerances
to different types of faults is provided and the level of fault-
tolerance varies depending upon the severity of faults. In
the context of automatic generation 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; sim-
ilar detectors can also be constructed from the components
in this paper. However, this issue is important in the con-
text 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.

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 de-
finition of speciﬁcation is adapted from Alpern and Schnei-
der [1] and Henzinger [13].

3.1 Real-Time Programs

Let $V = \{v_1, v_2, \ldots, v_n\}$, $n \geq 1$, be a ﬁnite set of discrete
variables and $X = \{x_1, x_2, \ldots, x_m\}$, $m \geq 1$, 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 num-
bers (denoted $\mathbb{R}_{\geq 0}$). A location is a function that maps
discrete variables in $V$ to a value from their respective do-
main. A clock constraint over $X$ is a Boolean combination of
formulae of the form $x \leq c$ or $x = y \leq c$, where $x, y \in X$,
$c \in \mathbb{Z}_{\geq 0}$, and $\leq$ is either $\leq$ 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 $\nu$ over the rest of the clock variables in $X$. A state
(denoted $\sigma$) is a pair $(s, \nu)$, where $s$ is a location and $\nu$ is a
clock valuation for $X$. Let $u$ be a (discrete or clock) variable
and $\sigma$ be a state. We denote the value of $u$ in state $\sigma$ 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 computa-
tion is a ﬁnite or inﬁnite timed state sequence of the form
$\sigma_0, \tau_0 \rightarrow \sigma_1, \tau_1 \rightarrow \cdots$ if the following con-
sitions are satisﬁed:

1. $\nu_0 = (s_0, \nu_0)$ is a state in the state space of $V$
and $X$ for all $i \in \mathbb{Z}_{\geq 0}$, and (2) the sequence $\tau_0, \tau_1, \ldots$ (called
the global time), where $\tau_i \in \mathbb{R}_{\geq 0}$ for all $i \in \mathbb{Z}_{\geq 0}$, satisfies
the following constraints:

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

Notice that in Deﬁnition 3.1, we do not specify an initial
value for the global time. Now, let $\Sigma$ be any set of computa-
tions. We require that $\Sigma$ must be closed with respect to time
offsets. That is, $\forall \tau \in \Sigma : \forall t \in \mathbb{R} : (\tau + t) \in \Sigma$, where $\tau + t$
denotes the computation $(\sigma_0, \tau_0 + t) \rightarrow (\sigma_1, \tau_1 + t) \rightarrow \cdots$,
Notation. Let \( \pi \) denote the pair \((\pi_1, \pi_2)\) in computation \(\pi\). Also, let \(\Gamma\) be a finite computation of length \(n\) and \(\bar{\Gamma}\) be a finite or infinite computation. The concatenation of \(\pi\) and \(\bar{\Gamma}\) (denoted \(\pi\bar{\Gamma}\)) is a computation, iff states \(\pi_{n+1}\) and \(\bar{\Gamma}_0\) meet the constraints of Definition 3.1. Otherwise, the result of concatenation is null. If \(\Gamma\) and \(\Psi\) are two sets containing finite and/or infinite computations respectively, then \(\Gamma\Psi = \{\pi\bar{\Gamma} \mid (\pi \in \Gamma) \land (\bar{\Gamma} \in \Psi)\}\).

Definition 3.2 (suffix and fusion closure) Suffix closure of a set of computations means that if a computation \(\pi\) is in that set then so are all the suffixes of \(\pi\). Fusion closure of a set of computations means that if computations \(\pi(\sigma, \tau)\bar{\pi}\) and \(\bar{\pi}(\sigma, \tau)\pi\) are in that set then so are the computations \(\pi(\sigma, \tau)\pi\) and \(\bar{\pi}(\sigma, \tau)\bar{\pi}\), where \(\pi\) and \(\bar{\pi}\) are computation prefixes, \(\pi\) and \(\bar{\pi}\) are computation suffixes, and \(\sigma\) is a state at global time \(\tau\).

Definition 3.3 (real-time programs) A real-time program \(P\) is specified by the tuple \((V_P, X_P, \Pi_P)\) where \(V_P\) is a finite set of discrete variables, \(X_P\) is a finite set of clock variables, and \(\Pi_P\) is a suffix closed and fusion closed set of infinite maximal computations in the state space of \(P\). By maximal, we mean that if \(\pi = \pi\bar{\pi} = \pi\bar{\pi} = \ldots\) is in \(\Pi_P\), where \((1)\) \(\pi = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots \rightarrow (\sigma_n, \tau_n)\), \((2)\) \(\bar{\pi} = (\sigma_{n+1}, \tau_{n+1}) \rightarrow (\sigma_{n+2}, \tau_{n+2}) \rightarrow \cdots \rightarrow (\sigma_{n+3}, \tau_{n+3}) \cdots\), and \((3)\) for all \(j > n\), \(\sigma_j = \sigma_{j+1}\) and \(\tau_{j+1} = \tau_j + (\tau_{j+1} - \tau_j)\), then no other computation in \(\Pi_P\) has a prefix of \(\pi\). In other words, given a computation prefix \(\pi\) of \(\Pi\), \(P\) does not contain the computation that stutters \(\sigma_{n+1}\) infinitely if there exists other computation of \(\Pi\) that extends \(\pi\).

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 \(P = (V_P, X_P, \Pi_P)\) is any subset of state space of \(P\) at \(t\). In the corresponding Boolean expression, clock constraints are in \(\Phi(X_P)\), i.e., the magnitude of clock variables are nonnegative integers.

Definition 3.5 (closure) We say that a state predicate \(S\) is closed in \(P = (V_P, X_P, \Pi_P)\) iff in every computation \((\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots \# \Pi_P\), if \(\sigma_r \models S\) (i.e., state predicate \(S\) holds in state \(\sigma_j\)), \(j \in \mathbb{Z}_{\geq 0}\), then \(\sigma_{k+1} \models S\) for all \(k, k \geq j\).

Definition 3.6 (S-computations) Let \(S\) be a state predicate and \(P = (V_P, X_P, \Pi_P)\) be a program. The \(S\)-computations of \(P\), denoted as \(P \models S\), is the set of all computations in \(\Pi_P\) 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 (TC) for the bridge has two discrete variables to represent the status of the signals, i.e., \(V_{TC} = \{\text{sig}_0, \text{sig}_1\}\), where \(\text{sig}_0\) and \(\text{sig}_1\) range over \(\{G, Y, R\}\). Thus, at any time, the values of \(\text{sig}_0\) and \(\text{sig}_1\) show in which direction cars are traveling. Moreover, for each signal, \(TC\) has three timers to change signal phase, i.e., \(X_{TC} = \{x_1, y_1, z_1\mid i = 0, 1\}\). 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 \(L:: \pi \models st\), where \(L\) is a label, \(\pi\) is a state predicate, \(st\) is a statement that describes how the program state is updated, and \(\lambda\) is a set of clock variables that are reset by execution of \(L\). Thus, \(\lambda\) denotes the set of transitions \(\{(\text{sig}_0, \nu) \rightarrow (\text{sig}_1, \nu(\lambda := 0))\mid \text{true}\}\) in state \((\text{sig}_0, \nu)\), 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 \(L:: \pi \models \text{wait}\), where \(y\) identifies the set of states from where delay transitions with arbitrary durations are allowed to be taken as long as \(y\) continuously remains true.

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

\[
\begin{align*}
T_{C1}:: & \quad (\text{sig}_1 := G) \land (1 \leq x_1 \leq 10) \rightarrow (\text{sig}_1 := Y); \\
T_{C2}:: & \quad (\text{sig}_1 := Y) \land (1 \leq y_1 \leq 2) \rightarrow (\text{sig}_1 := R); \\
T_{C3}:: & \quad (\text{sig}_1 := R) \land (z_1 \leq 1) \rightarrow (\text{sig}_1 := G); \\
T_{C4}:: & \quad ((\text{sig}_1 := G) \land (x_i \leq 10) \lor (\text{sig}_1 := Y) \land (y_i \leq 2)) \lor (\text{sig}_1 := R) \land (z_i \leq 1) \rightarrow \text{wait};
\end{align*}
\]

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 \(T_{C3}\) depends on \(x\) 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 \((V_{SPEC}, X_{SPEC}, \Sigma_{SPEC})\) where \(V_{SPEC}\) is a finite set of discrete variables, \(X_{SPEC}\) is a finite set of clock variables, and \(\Sigma_{SPEC}\) 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 \(P'\) (typically, a fault-tolerant program) to refine a program \(P\) (typically, a fault-intolerant program). Essentially, we would like to say that \(P'\) refines \(P\) iff computations of \(P'\) are a subset of that in \(P\). However, if \(P'\) is obtained by adding fault-tolerance to \(P\) then \(P'\) may contain additional variables that are not in \(P\). Hence, it will be necessary to project the computations of \(P'\) on (the variables of) \(P\) and then check if the projected computation is a computation of \(P\).
Definition 3.8 (state projection) Let \( P = (V_P, X_P, \Pi_P) \) and \( P' = (V_{P'}, X_{P'}, \Pi_{P'}) \) be real-time programs st. \( V_{P'} = V_P \cup \Delta_1 \) and \( X_{P'} = X_P \cup \Delta_2 \) for some \( \Delta_1 \) and \( \Delta_2 \). The projection of a state of \( P' \) on \( P \) is a state obtained by considering \( V_P \cup X_P \) only, i.e., by abstracting away the variables in \( \Delta_1 \) and \( \Delta_2 \).

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

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

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 \( P \) be a real-time program, \( S \) be a nonempty state predicate, and \( SPEC \) be a specification. We say that \( S \) is an invariant of \( P \) for \( SPEC \) iff \( P \) 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 \( P \)” abbreviates \( “S \) is an invariant of \( P \) 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 \( P \) be a real-time program, \( S \) be a state predicate, and \( SPEC \) be a specification. We say that program \( P \) maintains \( SPEC \) from \( S \) iff (1) \( S \) is closed in \( P \), and (2) for all computation prefixes \( \pi \) of \( P \) that start from \( S \), there exists a computation suffix \( \bar{\pi} \) st. the projection of \( \bar{\pi} \) on \( SPEC \) is in \( SPEC \). We say that \( P \) violates \( SPEC \) from \( S \) iff it is not the case that \( P \) maintains \( SPEC \) from \( S \).

We note that if \( P \) refines \( SPEC \) from \( S \) then \( P \) 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 \( \delta \)-corrector component to the intolerant program (see Section 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 \( P \rightarrow_{\leq \delta} Q \), where \( P \) and \( Q \) are two state predicates and \( \delta \in \mathbb{Z}_{\geq 0} \), is the set of all computations \( \langle \sigma_0, \tau \rangle \rightarrow \langle \sigma_1, \tau_1 \rangle \rightarrow \cdots \) 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) \( \tau_j \leq \tau_i \leq \delta \), 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 \( \delta \) time units and \( P \) remains true until \( Q \) becomes true. We call \( P \) the event predicate, \( Q \) the response (or recovery) predicate, and \( \delta \) the response (or recovery) time.

Assumption 3.12 We assume that the set of clock variables of any stable bounded response property \( P \rightarrow_{\leq \delta} Q \) 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 \( SPEC_{\text{bt}} \), (2) timing dependent safety characterized by a set of stable bounded response properties (denoted \( SPEC_{\text{br}} \)), and (3) liveness.

Definition 3.13 (safety specifications) Let \( SPEC_{\text{bt}} \) be a finite set of instantaneous bad transitions of the form \( \langle s_0, \nu, \lambda \models 0 \rangle \), where \( s_0 \) and \( s_1 \) are two locations and \( \lambda \subseteq X_{SPEC} \). We denote the specification whose computations have no transition in \( SPEC_{\text{bt}} \) by \( SPEC_{\text{br}} \).

2. (timing constraints) We denote \( SPEC_{\text{br}} \) by the conjunction \( \land_{i=0}^m (P_i \rightarrow_{\leq \delta_i} Q_i) \), for state predicates \( P_i \) and \( Q_i \), and, response times \( \delta_i \).

Thus, given a specification \( SPEC \), one can implicitly identify \( SPEC_{\text{br}} \) and \( SPEC_{\text{bt}} \) as defined above. Throughout the paper, \( SPEC_{\text{br}} \) 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 \( \pi \), there exists an infinite computation \( \exists \bar{\pi} \) st. \( \bar{\pi} \in SPEC \).

3.2.1 Example (cont’d)

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

\[ SPEC_{\text{br}TC} = \{(\sigma_0, \sigma_1) \mid (\text{sig}_0(\sigma_1) \neq R) \land (\text{sig}_1(\sigma_1) \neq R)\} \]

We define \( SPEC_{\text{br}TC} \) of \( TC \) in Section 4, where we formally define the notion of bounded-time recovery.

One invariant for the program \( TC \) is the following:

\[ S_{TC} = \forall i \in \{0, 1\} : \]

\[ ((\text{sig}_i = G) \Rightarrow ((\text{sig}_i = R) \land (z_i \leq 10) \land (z_i > 1)) \land \]

\[ ((\text{sig}_i = Y) \Rightarrow ((\text{sig}_i = R) \land (y_i \leq 2) \land (z_i > 1)) \land \]

\[ ((\text{sig}_i = R) \land (\text{sig}_j = R)) \Rightarrow ((z_i \leq 1) \lor (z_j \leq 1)) \]
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 \( P = (V_P, X_P, \Pi_P) \) be a real-time program. The set \( f \) of faults is specified by the union of the following two sets:

1. a finite set \( f^i \) 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 \( \lambda \) is a subset of \( X_P \), and
2. a finite set \( f^\ell \) of delay faults of the form \((s, \nu) \rightarrow (s, \nu + \tau)\), which keeps the program in location \( s \) for time duration \( \tau \), \( \tau \in \mathbb{R}_{\geq 0} \).

**Assumption 4.2** Given a real-time program \( P = (V_P, X_P, \Pi_P) \) and a set \( f \) of faults, we assume that no computation in \( \Pi_P \) 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 \( \mathcal{P} = (V_P, X_P, \Pi_P) \) be a program and \( f \) be a set of faults. We denote the program \( \mathcal{P} \) in the presence of \( f \) by \( \mathcal{P}[f] = (V_P, X_P, \Pi_P[f]) \). Intuitively, \( \Pi_P[f] \) is obtained by inserting transitions of \( f \) in computations of \( \Pi_P \). Formally, let \( Z \) be a set of computations and \( \bullet \) be the operator that fuses two (finite or infinite) computations of \( Z \),

\[
\bullet(Z) = \{ \pi(\sigma, \tau) \overline{\beta} \mid \exists \tau', \overline{\psi} : (\pi(\sigma, \tau) \overline{\beta} \in Z) \land (\overline{\psi}(\sigma, \tau) \overline{\beta} \in Z) \},
\]

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

\[
\Pi_{\mathcal{P}[f]} = \{ \sigma \mid \sigma \in \text{FCC}(\Pi_P \cup f) \land \sigma \notin \text{FCC}(f) \}.
\]

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 \( \mathcal{P} = (V_P, X_P, \Pi_P) \) 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)**

\( T_C \) 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 \( S_{T_C} \) (cf. Subsection 3.2), without changing the location of \( T_C \). Formally,

\[
F_0 : S_{T_C} \xrightarrow{\{z_0\}} \text{skip};
F_1 : S_{T_C} \xrightarrow{\{z_1\}} \text{skip}.
\]

It is straightforward to see that in the presence of \( F_0 \) and \( F_1 \), \( T_C \) may violate \( \text{SPEC}_{\overline{TC}} \). For instance, if \( F_1 \) occurs when \( T_C \) is in a state of \( S_{T_C} \) where \((s_{00} = \text{sig}_1 = R) \land (z_0 \leq 1) \land (z_1 > 1) \), in the resulting state, we have \((s_{00} = \text{sig}_2 = R) \land (z_0 \leq 1) \land (z_1 = 0) \). From this state, immediate execution of timed actions \( T_{C0} \) and then \( T_{C1} \) results in a state where \((\text{sig}_{0} = \text{sig}_1 = G) \), which is clearly a violation of the safety specification \( \text{SPEC}_{\overline{TC}} \). In our traffic controller example, for simplicity, we assume that faults \( F_0 \) and \( F_1 \) only occur in a state where \( S_{T_C} \) holds (i.e., faults do not re-occur outside the invariant of \( T_C \)).

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 \( \text{SPEC} \), the hard-masking tolerance specification of \( \text{SPEC} \) is identical to \( \text{SPEC} \). In other words, the occurrence of all faults are masked. Moreover, we require \( \text{SPEC} \) to prescribe a bounded-time recovery mechanism.

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

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 \( \text{SPEC} \) if it refines \( \text{SPEC} \) in the absence of faults and it refines the hard-masking tolerance specification of \( \text{SPEC} \) in the presence of \( f \), we define ‘hard-masking \( f \)-tolerant to \( \text{SPEC} \) 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, \( \text{SPEC} \) be a specification, and \( \theta \) be a nonnegative integer. We say that \( \mathcal{P} \) is hard-masking \( f \)-tolerant to \( \text{SPEC} \) with recovery time \( \theta \) from \( S \) iff the following two conditions hold:

1. \( \mathcal{P} \) refines \( \text{SPEC} \) from \( S \), and
2. there exists \( T \) st. \( T \supseteq S \) and \( \mathcal{P}[f] \) refines the hard-masking tolerance specification of \( \text{SPEC} \) for recovery time \( \theta \) and recovery predicate \( S \) from \( T \).

**4.2.1 Example (cont’d)**

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

\[
\text{SPEC}_{\overline{T_C}} \equiv (\neg S_{T_C} \iff \delta T_{C}) \land (\neg S_{T_C} \iff \leq \delta Q_{T_C}),
\]
where \( Q_{TC} = \forall i \in \{0, 1\} : ((s_i \equiv G) \land (z_i > 1)) \). The response times in \( \text{SPEC}_{\pi_{TC}} \) are simply two arbitrary numbers for illustration.

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

\[
\begin{align*}
TC'1 : & \quad (s_1 = G) \land (1 \leq x_i \leq 10) \quad \frac{(y_i)}{} \quad (s_i := Y); \\
TC'2 : & \quad (s_1 = Y) \land (1 \leq y_i \leq 2) \quad \frac{(z_i)}{} \quad (s_i := R); \\
TC'3 : & \quad (s_1 = R) \land (z_i \leq 1) \land \\
& \quad \frac{(s_i)}{} \quad (s_i := G); \\
TC'4 : & \quad ((s_1 = G) \land (x_i \leq 10)) \lor \\
& \quad ((s_1 = Y) \land (y_i \leq 2)) \lor \\
& \quad ((s_1 = R) \land (z_i \leq 1)) \quad \frac{\text{wait}}{}; \\
TC'5 : & \quad (s_1 \neq R \lor s_1 \neq R) \land (z_i, z_j \leq 1) \\
& \quad \frac{s_i, s_j := R}{s_i}; \\
TC'6 : & \quad (s_1 = s_2 = R) \land (z_i, z_j \leq 4) \land \\
& \quad (t_1 \leq 3) \quad \frac{\text{wait}}{}; \\
TC'7 : & \quad (s_1 = s_2 = R) \land (z_i, z_j > 1) \land \\
& \quad (t_2 \leq 7) \quad \frac{\text{skip}}{}; \\
TC'8 : & \quad (s_1 = s_2 = R) \land (z_i, z_j > 1) \land \\
& \quad (t_2 \leq 7) \quad \frac{\text{wait}}{};
\end{align*}
\]

where \( t_1 \) and \( t_2 \) are the special clock variables that accompany stable bounded response properties in \( \text{SPEC}_{\pi_{TC}} \) (cf. Assumption 3.12). In Sections 5 and 6, we demonstrate how we decompose \( TC' \) into \( TC \) 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., \( \text{SPEC}_{\pi}\) in Definition 3.13).

Definition 5.1 (detectors) Let \( W \) and \( D \) be state predicates. Let ‘\( W \) detects \( D \)’ be the specification, that is the set of all infinite computations \( \pi = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \ldots \), satisfying the following three conditions:

- (Safety) For all \( i \in \mathbb{Z}_{\geq 0} \), if \( \sigma_i \models W \) then \( \sigma_i \models D \).
- (Progress) For all \( i \in \mathbb{Z}_{\geq 0} \), if \( \sigma_i \models D \) then there exists \( k, k \geq i \); \( \sigma_k \models W \) 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 \( D \) be a program and \( D, W \), and \( U \) be state predicates of \( D \). We say that \( W \) detects \( D \) in \( D \) from \( D \) (i.e., \( D \) is a detector) iff \( D \) refines ‘\( W \) detects \( D \)’ from \( U \).

A detector \( D = (V_P, X_P, \Pi_D) \) is used to check whether its “detection predicate”, \( D \), is true. Since \( D \) satisfies Progress from \( U \), in any computation in \( \Pi_D \), if \( U \land D \) is true continuously, \( D \) eventually detects this fact and makes \( W \) true. Since \( D \) satisfies Safeness from \( U \), it follows that \( D \) never lets \( W \) witness \( D \) incorrectly. Moreover, since \( D \) 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 \( D \) is a hard-masking tolerant detector if \( \text{SPEC} \) 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., \( \text{SPEC}_{\pi} \) in Definition 3.13) can be merely determined by considering transitions in \( \text{SPEC}_{\pi}\).

Lemma 5.3 Let \( \text{SPEC} \) be a specification, \( \bar{\pi} \) be a computation prefix, \( \sigma \) and \( \sigma' \) be two states, and \( \tau, \tau' \in \mathbb{R}_{\geq 0} \), where \( \tau = \tau' \).

- \( \bar{\pi}(\sigma, \tau) \text{ maintains } \text{SPEC}_{\bar{\pi}} \)
- \( \bar{\pi}(\sigma, \tau)(\sigma', \tau') \text{ maintains } \text{SPEC}_{\bar{\pi}} \)

In Lemma 5.4, we show that there exists a set of states from where execution of programs maintains \( \text{SPEC}_{\pi} \). We call such a state predicate a detection predicate for \( \text{SPEC}_{\pi} \).

Lemma 5.4 Given a program \( \mathcal{P} = (V_P, X_P, \Pi_P) \), there exists a state predicate \( D \) (called detection predicate) st. all computations of \( \Pi_P \) that start from \( D \) maintain \( \text{SPEC}_{\pi} \).

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

Lemma 5.5 Given a program \( \mathcal{P} = (V_P, X_P, \Pi_P) \) and a specification \( \text{SPEC} \), there exists a unique weakest detection predicate of \( \mathcal{P} \) for \( \text{SPEC}_{\pi} \).

We are now ready to deduce 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 \( L :: \text{Guard} \overset{\lambda}{\rightarrow} st \), Lemma 5.5 shows that there exists a unique weakest detection predicate, say \( wdp \), from where execution of \( L \) does not violate \( \text{SPEC}_{\Pi} \). Hence, to show the existence of detectors, we require the detection predicate of such a timed action to be \( \text{Guard} \land wdp \). 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 \( P' \) encapsulates (typically, a fault-intolerant) program \( P \) then for each timed action of \( P \) of the form \( \text{Guard} \overset{\lambda}{\rightarrow} st \), \( P' \) contains a timed action of the form \( \text{Guard} \land \text{Guard'} \overset{\lambda \cup \lambda'}{\rightarrow} st || st' \). The semantic of \( st || st' \) corresponds to the statement where \( st \) and \( st' \) are executed simultaneously, clock variables in \( \lambda \cup \lambda' \) are reset, and the timed action is enabled only when its guard, \( \text{Guard} \land g' \), is true. In other words, \( P' \) has a timed action corresponding to each timed action of \( P \) (possibly) with a stronger guard, additional assignments in \( st' \), and additional clock variables in \( \lambda' \). Notice that the assignments in \( st' \) and clock variables in \( \lambda' \) may be added in order to add fault-tolerance to \( P \) (cf. the notion of projection in Subsection 3.2). To show that \( P' \) is using a detector for a timed action of \( P \), we require the witness predicate of that detector to be \( \text{Guard} \land \text{Guard'} \) which is the guard of the corresponding timed action in \( P' \).

**Definition 5.6 (encapsulates)** Let \( P = (V_P, X_P, \Pi_P) \) and \( P' = (V_{P'}, X_{P'}, \Pi_{P'}) \) be two real-time programs and \( S \) be a state predicate. We say that \( P' \) encapsulates \( P \) from \( S \) if each timed action in \( P' \) is enabled in a state in \( S \) and that updates variables in \( V_P \) is of the form \( \text{Guard} \land g' \overset{\lambda \cup \lambda'}{\rightarrow} st || st' \), where \( \text{Guard} \overset{\lambda}{\rightarrow} st \) is a timed action of \( P \) and \( st' \) does not update variables in \( V_P \) and \( \lambda' \cap X_P = \{ \} \).

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

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 \( P \) and \( P' \) be two real-time programs. We say that \( P' \) reuses \( P \) from \( S \) iff the following two conditions are satisfied:
- \( P' \) refines \( P \) from \( S \), and
- \( P' \) encapsulates \( P \) from \( S \).

### 5.3 Example (cont’d)

It is straightforward to see that the weakest detection predicate for \( TC' \) is:
\[
wdp_{TC'} = \{ \sigma | sig_j(\sigma) = R \},
\]
where \( i \in \{0,1\} \) and \( j = (i + 1) \mod 2 \). Thus, in program \( TC' \) (cf. Subsection 4.2), the guard of \( TC' \), is strengthened in order for \( TC' \) to refine \( \text{SPEC}_{\Pi_{TC'}} \) in the presence of faults. Intuitively, \( TC' \) is allowed to change phase from red to green only when the other signal is red. More precisely, \( TC' \) uses the detector \( D_{TC'} \) which consists of timed guarded commands \( TC'_{1}, TC'_{2}, \) and \( TC'_{4} \), with the following detection and witness predicates:
\[
D_{TC_{i}} = \text{guard}(TC_{3}) \land wdp_{TC_{i}}
\]
\[
W_{TC_{i}} = \text{guard}(TC_{3}).
\]

It is easy to see that \( W_{TC_{i}} \) detects \( D_{TC_{i}} \) in \( D_{TC_{i}} \) from \( S_{TC} \) in both absence and presence of \( F_{0} \) and \( F_{1} \). Notice that \( D_{TC_{i}} \) exhibits Zeno behavior since when the witness predicate becomes true, there does not exist a timed guarded command whose guard is enabled except \( TC'_{4} \). However, it is important that the entire program does not show Zeno behavior. For instance, one can observe that \( TC'_{3} \) ensures time progress for \( TC' \).

### 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 \( \text{SPEC}_{\Pi} \) in the absence of faults then it contains detectors. The intuition is that if program \( P' \) is designed by transforming \( P \) so as to refine \( \text{SPEC}_{\Pi} \), then the transformation must have added detectors for \( P \), and \( P' \) reuses \( P \). 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 \( P' \) is designed by reusing \( P \) to tolerate a set \( f \) of faults, \( P' \) contains a hard-masking tolerant detector for each action of \( P \). 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 \( \text{SPEC}_{\Pi} \) in the absence of faults.

**Claim 5.10** Let \( P \) and \( P' \) be real-time programs, \( S \) be a nonempty state predicate, and \( \text{SPEC} \) be a specification. If
- \( P' \) reuses \( P \) from \( S \), and
- \( P' \) refines \( \text{SPEC}_{\Pi} \) from \( S \),
then
- \((\forall ac \mid ac \text{ is a timed action of } P : P' \text{ contains a detector of a detection predicate of } ac \text{ for } \text{SPEC})\).
Now, we show that if a hard-masking f-tolerant program $P'$ is designed by reusing $P$ then $P'$ contains a hard-masking tolerant detector for each action in $P$.

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

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

then

- $(\forall c | ac$ is a timed action of $P : P'$ 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. $SPEC_{\neg f}$ 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 $\delta\}$ be the specification, that is the set of all infinite computations $\tau = (\tau_0, \tau_1) \rightarrow (\tau_1, \tau_2) \rightarrow \cdots$, satisfying the following conditions:

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

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

- $W$ weakly corrects $C$ within $\delta$, and
- ([Strong] Convergence) In addition to Weak Convergence, $C$ is closed in $\tau$.

---

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

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

Notice that since $C$ satisfies Weak (respectively, Strong) Convergence from $U$, it follows that $C$ reaches a state where $C$ becomes true within $\delta$ 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 $C$ satisfies Safety from $U$. Moreover, since $C$ satisfies Progress from $U$, it follows that $W$ eventually becomes true. And, finally, since $C$ 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 $C$ 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 $\forall S \rightarrow S \subseteq S$ (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 $\sigma_0$ is the earliest state from where recovery must commence. Thus, $\tau_1$ is the time instance where correction is complete and $\tau_1 - \tau_0$ is the duration of correction. In case of strong δ-correctors, $\sigma_0$ 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-correction predicates for each stable bounded-response property in \( SPEC_{\tau} \), introduced in Subsection 4.2. To this end, first, consider the property \( \neg S_{Tc} \Rightarrow \neg \text{Q} \). When \( Tc \subset \{ F_0, F_1 \} \), reaches a state in \( \neg S_{Tc} \land \neg Q \) 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 \( C_{12}^{Tc} \) be the weak 3-corrector in \( Tc' \) consisting of timed actions \( Tc'5, \) and \( Tc'6, \) with correction and witness predicates both equal to \( Q \). Intuitively, \( C_{12}^{Tc} \) ensures that when \( Tc' \) is in a state outside the invariant, it reaches a state where both signals are red within 3 time units.

Likewise, for the property \( \neg S_{Tc} \Rightarrow \neg \text{Q} \), let \( C_{12}^{Tc} \) be the strong 3-corrector consisting of timed actions \( Tc'7, \) and \( Tc'8, \) with witness and corrections predicates equal to \( S_{Tc} \). In \( C_{12}^{Tc'}, \) a \( z \) timer gets reset when the state of \( Tc' \) is in \( \neg S_{Tc} \land Q \) within 7 time units since the occurrence of a fault. Such a reset takes the traffic controller back to its invariant predicate \( S_{Tc} \) where timed action \( TC1 \) is enabled. Notice that unlike the the entire program \( Tc'1, \) both \( C_{12}^{Tc} \) and \( C_{12}^{Tc'} \) 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 \( \theta \) 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 \( P' \) is designed by reusing \( P \) to tolerate a set \( \mathcal{F} \) of faults, \( P' \) 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, \( \pi_{\leq \delta} \), denotes a finite computation \( (\sigma_0, \tau_0) \Rightarrow (\sigma_1, \tau_1) \Rightarrow \cdots (\sigma_n, \tau_n) \) that satisfies the timing constraint \( \tau_0 - \tau_0 \leq \delta \), 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. \( \sigma_i = S \) for all \( 0 \leq i \leq n \). Thus, \( (true)_{\leq \delta} \) denotes an arbitrary finite computation with no specific time bound.

Definition 6.4 (becomes) Let \( P = \langle V_P, X_P, \Pi_P \rangle \) and \( P' = \langle V_{P'}, X_{P'}, \Pi_{P'} \rangle \) be two real-time programs. We say that \( P' \) becomes \( P \) within \( \theta \) from \( T \) if \( P' \) refines \( (true)_{\geq \delta} \Pi_P \) from \( T \).

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

Claim 6.5 Let \( P \) and \( P' \) be real-time programs, \( S \) and \( T \) be nonempty state predicates, \( SPEC \) be a specification, and \( \theta \) be a nonnegative integer.
If
- \( P \) refines \( SPEC \) from \( S \),
- \( P' \) refines \( P \) from \( S \), and
- \( P' \) becomes \( P|S \) within \( \theta \) from \( T \),
then
- there exists \( \delta \in \mathbb{Z}_{\geq 0} \) st. \( P' \) is a strong \( \delta \)-corrector of \( S \).

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

Lemma 6.6 Let \( P \) and \( P' \) be real-time programs, \( R, S, \) and \( T \) be nonempty state predicates where \( R \Rightarrow S \), \( SPEC \) be a specification, and \( \theta \) be a nonnegative integer.
If
- \( P \) refines \( SPEC \) from \( S \),
- \( P' \) refines \( P \) from \( R \), and
- \( P' \) becomes \( P|R \) within \( \theta \) from \( T \),
then
- there exists \( \delta \in \mathbb{Z}_{\geq 0} \) st. \( P' \) is a hard-masking strong \( \delta \)-corrector with recovery time \( \theta \) of \( S \).

We now illustrate the role of strong \( \delta \)-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 \( P' \) with recovery time \( \theta \) is designed by reusing \( P \) then there exists \( \delta \in \mathbb{Z}_{\geq 0} \) st. \( P' \) contains a hard-masking \( f \)-tolerant strong \( \delta \)-corrector with recovery time \( \theta \) for an invariant predicate of \( P \). 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 \( P = \langle V_P, X_P, \Pi_P \rangle \) 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 \( P_{\Pi P} \) where \( \sigma_0 = S \) and \( \sigma_i \neq S, 1 \leq i \leq n \). We assume that the number of occurrence of faults between states \( \sigma_i \) and \( \sigma_n \) is at most \( k \) for some \( k \in \mathbb{Z}_{\geq 0} \).

Theorem 6.8 Let \( P \) and \( P' \) be real-time programs, \( R \) and \( S \) be nonempty state predicates, \( SPEC \) be a specification, and \( \theta \) be a nonnegative integer.
If
- \( P \) refines \( SPEC \) from \( S \),
- \( P' \) is hard-masking \( f \)-tolerant for \( SPEC \) from \( R \), where \( R \Rightarrow S \),
- \( P' \) refines \( P \) from \( R \), and
- \( P' \) becomes \( P|R \) within \( \theta \) from \( T \), where \( T \) is an \( f \)-span of \( P' \),
then
- there exist \( \delta \) and \( \theta' \) in \( \mathbb{Z}_{\geq 0} \) st. \( P' \) is a hard-masking \( f \)-tolerant strong \( \delta \)-corrector with recovery time \( \theta' \) of \( S \).
6.5 The Necessity of Existence of Weak \( \delta \)-Correctors in Hard-Masking Programs

Just like the relation between recovery constraint and strong \( \delta \)-correctors in hard masking programs, we show that if a hard-masking \( f \)-tolerant program \( P' \) is designed by reusing program \( P \) then \( P' \) contains a hard-masking tolerant weak \( \delta \)-corrector for each stable bounded response property in \( \text{SPEC}_{P'} \).

**Theorem 6.9** Let \( P \) and \( P' \) be real-time programs, \( R \) and \( S \) be nonempty state predicates where \( R \Rightarrow S \), \( \text{SPEC} \) be a specification, and \( \theta \) be a nonnegative integer. If
- \( P \) refines \( \text{SPEC} \) from \( S \),
- \( P' \) reuses \( P \) from \( R \),
- \( P' \) is hard-masking \( f \)-tolerant to \( \text{SPEC} \) from \( R \)
then
- \( (\forall i \mid 0 \leq i \leq n; \text{there exists } \delta \in \mathbb{Z}_{>0} \text{ st. } P' \) is a hard-masking tolerant weak \( \delta \)-corrector for the response predicate of stable bounded response property \( P_i \mapsto \leq \delta_i Q_i \) of \( \text{SPEC}_{P'} \).

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 \( \delta \)-correctors, and strong \( \delta \)-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 \( \delta \)-correctors.

8. REFERENCES