Compositional Verification of Real-Time Fault-Tolerant Programs[*]

Borzoo Bonakdarpour
VERIMAG
Centre Équation
2 ave de Vinage
38610, GIÈRES, France
borzoo@imag.fr

Sandeep S. Kulkarni
3115 Engineering Building
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
sandeep@cse.msu.edu

ABSTRACT

A *hard-masking* real-time program is one that satisfies safety (including timing constraints) and liveness properties in the absence and presence of faults. It has been shown that any hard-masking program can be decomposed into a fault-intolerant version and a set of fault-tolerance components known as *detectors* and *δ-correctors*. In this paper, we introduce a set of sufficient conditions for interference-freedom among fault-tolerance components and real-time programs. We demonstrate that such conditions elegantly enable us to compositionally verify the correctness of hard-masking programs. Preliminary model checking experiments show very encouraging results in both achieving speedups and reducing memory usage in verification 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, Compositional verification, Interference - freedom, Formal methods

[*]This is an extended version of the paper appeared in ACM/IEEE International Conference on Embedded Software (Emsoft'09). This work is partially sponsored by the COMBEST European project, NSF CAREER CCR-0092724, and ONR Grant N00014-01-1-0744.

1. INTRODUCTION

Component-based methods have increasingly become popular in all steps of system analysis and design. Conceptually, each component is assigned an (ideally) independent task in order to separate various concerns of a computing system, while their collaboration accomplishes a common goal. In the context of embedded systems, two crucial tasks that are desirable to separate are *meeting timing constraints* and *providing dependability*. However, these tasks have conflicting natures, making their separation and reasoning about their correctness fairly complex.

In [10], we proposed a *theory of real-time fault-tolerance components*. This theory separates fault-tolerance and functionality concerns of real-time systems. We identified three types of fault-tolerance components, namely *detectors*, *weak δ-correctors*, and *strong δ-correctors*. These components are conceptually 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 desirable state within a bounded amount of time. We also introduced *necessary conditions* for existence of the aforementioned components in a *hard-masking* fault-tolerant real-time program. A hard-masking program is one that meets timing-independent safety, timing constraints, and bounded-time recovery to the normal behavior even in the presence of faults. In particular, we showed that any hard-masking program can be decomposed into a fault-intolerant version and a set of fault-tolerance components. The fault-intolerant program provides the normal operation and functional behavior, while the fault-tolerance components ensure satisfaction of the program's specification in the presence of faults.

One can observe that applying our theory of decomposition in [10] provides us with structural insights for component-based analysis of fault-tolerant real-time programs. Nevertheless, the theory comes short of assisting in modular reasoning about the correctness of such programs. For instance, a natural question in this context is:

> *Does correctness of a fault-tolerance component imply the correctness of its properties in the hard-masking program that contains it?*

The answer to this question is affirmative, if execution of the component is not interfered by other components or by the fault-intolerant program.

With this motivation, in this paper, we focus on developing *sufficient conditions* tailored for exhibiting non-interference of fault-tolerance components. The conditions identified in this paper are *containment*, *superposition*, *atomicity*, *order of execution*, and *termination*. Obviously, correctness of fault-tolerance components along with their interference-freedom guarantees the correctness of their properties in the hard-masking program that contains them. A potential consequence of such modular reasoning, which is equally important, is to reduce the cost of verification of fault-tolerant real-time programs. Thus, we propose the following four steps in order to compositionally verify the correctness of hard-masking properties of a given real-time program:

1. decomposing the program into a fault-intolerant program, detector, and δ-corrector components,
2. demonstrating that these components and the (fault-intolerant) program do not interfere,
3. verifying the basic functionality of the program in the absence of faults (e.g., computing tasks, deadlock-freedom, etc), and
4. verifying the correctness of fault-tolerance components separately.

In [10], we showed the *necessity* of existence of fault-tolerance components in any hard-masking program and presented a constructive approach to obtain such components. In fact, in many cases, this task can be automated as well. This constitutes Step 1. Steps 3 and 4 can be performed using a model checker or a theorem prover. Thus, in this paper, our focus is on Step 2, i.e., sufficient conditions that enable us to achieve compositional verification. We emphasize that except termination, all other conditions can be verified through simple syntactic methods. In other words, the time for demonstrating interference-freedom of components using our results is expected to be negligible and all resources can be diverted to their verification. Our preliminary experiments on model checking of fault-tolerant real-time programs using the theory presented in this paper demonstrate very encouraging results in achieving better memory usage and speedups.

Organization. In Section 2, we define the preliminary concepts. Section 3 is dedicated to present our fault model and the notion of hard-masking fault-tolerance. Then, in Section 4, we introduce the notion of fault-tolerance components (i.e., detectors and δ-correctors). In Section 5, we propose the theory of interference-freedom for fault-tolerance components and present experimental results. Section 6 presents the related work. Finally, we make concluding remarks and discuss future work in Section 7. We use a running example (circular traffic controller) throughout the paper to better describe the concepts. A guide to notation is provided in Appendix A and all proofs appear in Appendix B.

2. REAL-TIME PROGRAMS AND SPECIFICATIONS

In our framework, real-time programs are specified in terms of their state space and their computations [4,5]. The definition of specification is adapted from Alpern and Schneider [3] and Henzinger [16].

2.1 Real-Time Programs

Let $V = \{v_0, v_1 \cdots\}$ be a set of *discrete variables* and $X = \{x_0, x_1 \cdots\}$ be a set of *clock variables*. Each discrete variable v_i, $0 \leq i$, is associated with a *domain* D_i of values. Each clock variable x_j, $0 \leq j$, 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. A *transition* is a pair (σ_0, σ_1) of states in the state space. We classify transitions as follows:

1. *immediate transitions* 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, and
2. *delay transitions* of the form $(s, \nu) \rightarrow (s, \nu + \tau)$, which preserves the location of state at s for time duration τ, $\tau \in \mathbb{R}_{\geq 0}$.

Definition 2.1 (state predicates) A *state predicate* S is any subset of the state space such that in the corresponding Boolean expression, clock constraints are in $\Phi(X)$, i.e., clock variables are only compared with nonnegative integers. By $\sigma \models S$, we mean that state σ is in state predicate S. ∎

Intuitively, a real-time program is defined in terms of its state space and set of transitions. However, to concisely present and reason about programs, we use timed guarded commands.

Definition 2.2 (real-time programs) A *real-time program* $\mathcal{P}$ is specified by the tuple $\langle V_{\mathcal{P}}, X_{\mathcal{P}}, GC_{\mathcal{P}} \rangle$, where $V_{\mathcal{P}}$ is a set of discrete variables, $X_{\mathcal{P}}$ is a set of clock variables, and $GC_{\mathcal{P}}$ is a finite set of *timed guarded commands* in the state space of $\mathcal{P}$ classified as follows:

- A *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 $V_{\mathcal{P}}$ is updated, and $\lambda \subseteq X_{\mathcal{P}}$ is a set of clock variables that are reset by execution of L. Thus, L denotes the set of transitions $\{(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0]) \mid (s_0, \nu) \models g$ and s_1 is obtained by changing s_0 as prescribed by λ and $st\}$.

- A *delay action* is of the form

$$L :: g \longrightarrow \textbf{wait};$$

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, L denotes the set of transitions $\{(s, \nu) \rightarrow (s, \nu + \tau) \mid (s, \nu + \epsilon) \models g$, for all $\tau \in \mathbb{R}_{\geq 0}$ and $\epsilon \leq \tau$. ∎

2.1.1 *Example: Circular Traffic Controller*

Consider a chain of n traffic signals operating in a circular manner. A signal changes phase from green to yellow and then to red, based on a set of timing constraints. A signal turns green some time after the previous signal in chain turns red. A traffic controller program ($\mathcal{TC}$) has n discrete variables to represent the status of the signals, i.e., $V_{\mathcal{TC}} = \{sig_0, sig_1 \cdots sig_{n-1}\}$, where the domain of sig_i, $0 \leq i \leq n-1$ is $\{G, Y, R\}$. $\mathcal{TC}$ has three timers for each signal to change phase, i.e., $X_{\mathcal{TC}} = \{x_i, y_i, z_i \mid 0 \leq i \leq n-1\}$. 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 previous signal becomes red. All signals operate identically. Thus, $GC_{\mathcal{TC}}$ is as follows:

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

where $i \in \{0 \cdots n-1\}$ and $j = (i - 1 + n) \mod n$.

2.2 Specifications

Definition 2.3 (computations) Let V and X be 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 obtained from 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 2.3, we do not specify an initial value for the global time. Now, let Σ be a 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 $(\sigma_0, \tau_0 + t) \rightarrow (\sigma_1, \tau_1 + t) \rightarrow \cdots$, such that $\tau_0 + t \geq 0$.
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 2.3. Otherwise, the result of concatenation is null. If Γ and Ψ are two sets containing finite and finite/infinite computations respectively, then $\Gamma\Psi = \{\overline{\alpha}\overline{\beta} \mid (\overline{\alpha} \in \Gamma) \ \wedge \ (\overline{\beta} \in \Psi)\}$.

Let $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, GC_{\mathcal{P}} \rangle$ be a program. We define a computation of $\mathcal{P}$ by adding the following constraint to Definition 2.3: for all $i \in \mathbb{Z}_{\geq 0}$, (σ_i, σ_{i+1}) is a transition of $\mathcal{P}$ described by a timed guarded command in $GC_{\mathcal{P}}$. We denote the set of computations of $\mathcal{P}$ by $\Pi_{\mathcal{P}}$ and require its maximality, i.e., given a computation prefix $\overline{\alpha}$ of length n of $\mathcal{P}$, $\mathcal{P}$ does not contain the computation that stutters at $\overline{\alpha}_{n-1}$ indefinitely if there exists other computation of $\mathcal{P}$ that extends $\overline{\alpha}$.

Definition 2.4 (closure) We say that a state predicate S is *closed* in $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, GC_{\mathcal{P}} \rangle$ iff in every computation $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ in $\Pi_{\mathcal{P}}$, if $\sigma_j \models S$, $j \in \mathbb{Z}_{\geq 0}$, then $\sigma_k \models S$, for all k, $k \geq j$. ∎

Definition 2.5 (specifications) A *specification* (or *property*), denoted $SPEC$, is a tuple $\langle V_{SPEC}, X_{SPEC}, \Sigma_{SPEC} \rangle$ where V_{SPEC} is a set of discrete variables, X_{SPEC} is a set of clock variables, and Σ_{SPEC} is a set of infinite computations in the state space of $SPEC$. ∎

As we argued in [10], we allow real-time programs to exhibit *Zeno* behaviors. The reason is due to the fact that such modeling gives freedom in dealing with situations where a component reaches a state from where another component makes progress.
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* [16]. 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 $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ in which, for all $i \geq 0$, if $\sigma_i \models P$, then there exists j, $j \geq i$, such that:

1. $\sigma_j \models Q$,
2. $\tau_j - \tau_i \leq \delta$, and
3. for all k, $i \leq k < j$, $\sigma_k \models P$.

In other words, 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.*

The specifications considered in this paper are an intersection of a *safety* specification and a *liveness* specification [3,16]. 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 2.6 (safety specifications)

1. (*timing-independent safety*) Let $SPEC_{bt}$ be a finite set of instantaneous *bad transitions* of the form $(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0])$, 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 3.

Definition 2.7 **(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 $\overline{\beta}$ such that $\overline{\alpha}\overline{\beta} \in SPEC$. ∎

2.2.1 Example (cont'd)

The timing-independent safety specification of $\mathcal{TC}$ is characterized by the set of bad transitions where two signals are not red in their target states:

$$SPEC_{bt_{\mathcal{TC}}} = \{(\sigma_0, \sigma_1) \mid \exists i, j \in \{0 \cdots n-1\} : \\ (i \neq j) \wedge (sig_i(\sigma_1) \neq R) \ \wedge \ (sig_j(\sigma_1) \neq R)\}.$$

We present the timing constraints of $\mathcal{TC}$ (i.e., $SPEC_{\overline{br}}$) in Section 3, where we define the notion of recovery.

2.3 Refinement

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 2.8 **(projection)** Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ and $\mathcal{P}' = \langle V_{\mathcal{P}'}, X_{\mathcal{P}'}, GC_{\mathcal{P}'}\rangle$ be real-time programs such that $V_{\mathcal{P}'} = V_\mathcal{P} \cup \Delta_v$ 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 2.9 **(refines)** Let $\mathcal{P}$ and $\mathcal{P}'$ be real-time programs, S be a state predicate and $SPEC$ 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}). ∎

The reason we require closure of S in Definition 2.9 is that S typically expresses a set of legitimate states from where correct execution of a program is closed. In order to reason about the correctness of programs (in the absence of faults), we define the notion of program invariant.

Definition 2.10 **(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. ∎

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 $SPEC$". 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 property 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 3, 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.

2.3.1 Example (cont'd)

One invariant for the program $\mathcal{TC}$ is the following, where $l = (j-1+n) \mod n$:

$$\begin{array}{l} S_{\mathcal{TC}} = \forall i \in \{0 \cdots n-1\} \ : \\ \quad [(sig_i = G) \quad \Rightarrow \quad ((x_i \leq 10) \ \wedge \\ \qquad \forall j \in \{0 \cdots n-1\} : (j \neq i \Rightarrow (sig_j = R)) \ \wedge \ (z_l > 1))] \\ \quad \wedge \\ \quad [(sig_i = Y) \quad \Rightarrow \quad ((y_i \leq 2) \ \wedge \\ \qquad \forall j \in \{0 \cdots n-1\} : (j \neq i \Rightarrow (sig_j = R)) \ \wedge \ (z_l > 1))] \\ \quad \wedge \\ \quad [(\forall j \in \{0 \cdots n-1\} : (sig_j = R)) \\ \qquad \Rightarrow \qquad \exists j \in \{0 \cdots n-1\} : ((z_l \leq 1) \ \wedge \\ \qquad\qquad\qquad \forall k \in \{0 \cdots n-1\} : (j \neq k \Rightarrow (z_k > 1)))]. \end{array}$$

It is straightforward to see that $\mathcal{TC}$ refines $SPEC_{\overline{bt_{\mathcal{TC}}}}$ from $S_{\mathcal{TC}}$, i.e., starting from a state in $S_{\mathcal{TC}}$, the program never reaches a state where two signals are not red.

3. HARD-MASKING FAULT-TOLERANCE

3.1 Fault Model

Intuitively, the faults that a program is subject to may perturb the execution of the program by variable corruptions or unexpected time delays. Thus, faults can be formally represented by a set f of (immediate and delay) transitions in the state space of a program. Similar to program actions, faults can be concisely modeled by timed and delay actions. We denote the *program* $\mathcal{P}$ *in the presence of* f by $\mathcal{P}[]f = \langle V_\mathcal{P}, X_\mathcal{P}, GC_{\mathcal{P}[]f}\rangle$, where $GC_{\mathcal{P}[]f}$ is obtained by taking the union of fault and program timed guarded commands.

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 3.1 **(fault-spans)** Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ 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 $\Pi_{\mathcal{P}[]f}$. ∎

3.1.1 Example (cont'd)

The program $\mathcal{TC}$ is subject to clock reset faults due to circuit malfunctions. In particular, we consider faults that reset one z timer at a time at any state in the invariant $S_{\mathcal{TC}}$ without changing the location of $\mathcal{TC}$. Formally,

$$F_i :: \ S_{\mathcal{TC}} \quad \xrightarrow{\{z_i\}} \quad \textbf{skip};$$

where $i \in \{0 \cdots n-1\}$. It is straightforward to see that in the presence of the above faults, $\mathcal{TC}$ may violate $SPEC_{\overline{bt}_{\mathcal{TC}}}$. For instance, if F_1 occurs when $\mathcal{TC}$ is in a state 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 $SPEC_{\overline{bt}_{\mathcal{TC}}}$. Thus, we require that when a fault occurs, the program, first, ensures that nothing catastrophic happens by reaching a state where both signals are red, and then recovers to its normal behavior. Formally, the timing constraints of $\mathcal{TC}$ is as follows:

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

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

3.2 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'. This tolerance specification is 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 [8]. 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 3.2 (hard-masking tolerance specification) Let $SPEC$ 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 invariant predicate S' as follows.

Definition 3.3 (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 such that $T \supseteq S$ and $\mathcal{P}[]f$ refines the hard-masking tolerance specification of $SPEC$ for recovery time θ and recovery predicate S from T. ∎

3.2.1 Example (cont'd)

The following program is a hard-masking version of our traffic controller program, denoted by $\mathcal{TC}'$. In this program, $i, k \in \{0 \cdots n-1\}$, $j = (i-1+n) \mod n$, $k \neq i$, and t_1 and t_2 are two new clock variables to keep track of time elapsed since $\neg S$ and Q have become true, respectively:

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

Intuitively, timed guarded command $\mathcal{TC}'3_i$ is revised to ensure refinement of $SPEC_{\overline{bt}}$ in the presence of faults. Moreover, $\mathcal{TC}'5_i$ and $\mathcal{TC}'6_i$ are added to ensure refinement of the first stable bounded response property in $SPEC_{\overline{br}}$. Finally, $\mathcal{TC}'7_i$ and $\mathcal{TC}'8_i$ are added to ensure refinement of the second stable bounded response property in $SPEC_{\overline{br}}$. We will formally analyze this program in Section 4.

4. FAULT-TOLERANCE COMPONENTS

One way to ensure the correctness of a program is to deal with the program as one unit and verify its properties. Alternatively, if we can somehow decompose a program into a set of interference-free components and verify each component individually, we can conclude that their composition satisfies the same set of properties as well. Verification of components separately is advantageous, as we generally deal with simpler tasks in terms of complexity of properties, size of state space, and resources required to complete verification.

In this section, we present three types of components that are involved in a hard-masking program, namely, *detectors*, *weak δ-correctors*, and *strong δ-correctors*. Roughly speaking, these components ensure satisfaction of timing-independent safety (i.e., $SPEC_{\overline{bt}}$), timing constraints (i.e., $SPEC_{\overline{br}}$), and bounded-time recovery, respectively, in the presence of faults.

4.1 Detectors

Intuitively, a detector is a program component that ensures satisfaction of timing-independent safety (i.e., $SPEC_{\overline{bt}}$ in Definition 2.6).

Definition 4.1 **(detects)** Let W and D be state predicates. Let 'W *detects* D' be the specification, that is the set of all infinite computations $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$, 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 $k, k \geq i$, such that $\sigma_k \models W$ or $\sigma_k \not\models D$.
- (*Stability*) There exists $i \in \mathbb{Z}_{\geq 0}$, such that 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 4.2 **(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 $\mathcal{D} = \langle V_\mathcal{D}, X_\mathcal{D}, GC_\mathcal{D} \rangle$ 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 a predicate of the fault-intolerant program from where timing-independent safety should be always satisfied and W is a predicate of the fault-tolerant program that witnesses the detection of D.

4.1.1 Example (cont'd)

One detector of $\mathcal{TC}'$ is $\mathcal{D}^1_{\mathcal{TC}'_i}$ defined by the following timed guarded commands $GC_{\mathcal{D}^1_{\mathcal{TC}'_i}} = \{\mathcal{TC}'1_i, \mathcal{TC}'2_i, \mathcal{TC}'4_i\}$ with the following detection and witness predicates:

$$D_{\mathcal{D}^1_{\mathcal{TC}'_i}} = guard(\mathcal{TC}3_i) \wedge (sig_j = R)$$
$$W_{\mathcal{D}^1_{\mathcal{TC}'_i}} = guard(\mathcal{TC}'3_i).$$

where $i \in \{0 \cdots n-1\}$ and $j = (i - 1 + n) \mod n$. It is straightforward to show that $\mathcal{D}^1_{\mathcal{TC}'_i}$ satisfies *Safeness*, *Stability*, and *Progress* with respect to $D_{\mathcal{D}^1_{\mathcal{TC}'_i}}$ and $W_{\mathcal{D}^1_{\mathcal{TC}'_i}}$. Another detector of $\mathcal{TC}'$ is $\mathcal{D}^2_{\mathcal{TC}'_i}$, where $GC_{\mathcal{D}^2_{\mathcal{TC}'_i}} = \{\mathcal{TC}'5_i\}$ and where $D_{\mathcal{D}^2_{\mathcal{TC}'_i}} = W_{\mathcal{D}^2_{\mathcal{TC}'_i}} = (sig_i = sig_j = R)$.

4.2 δ-Correctors

Intuitively, a δ-corrector is a program component that ensures *bounded-time recovery* to a *correction predicate*. For instance, recovery to the invariant predicate is essential to guarantee that liveness properties (cf. Definition 2.7) and timing constraints (cf. $SPEC_{\overline{br}}$ in Definition 2.6) are met when the state of a program is perturbed by the occurrence of faults. Thus, we will use δ-correctors where we need refinement of stable bounded response properties in the presence of faults. Depending upon the closure of the correction predicate in δ-correctors, they are classified into *weak* and *strong*.

Definition 4.3 **(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 $i \in \mathbb{Z}_{\geq 0}$, such that $\sigma_i \models C$ 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 $k, k \geq i$, such that $\sigma_k \models W$ or $\sigma_k \not\models C$.

Definition 4.4 **(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.
- (*Stability*) There exists $i \in \mathbb{Z}_{\geq 0}$, such that for all j, $j \geq i$, if $\sigma_j \models W$, then $\sigma_{j+1} \models W$ or $\sigma_{j+1} \not\models C$.
- (*Strong Convergence*) In addition to *Weak Convergence*, C is closed in $\overline{\sigma}$. ∎

Definition 4.5 **(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 *within* δ *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. ∎

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 *Weak/Strong 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, in case of strong δ-correctors, since $\mathcal{C}$ satisfies *Stability* from U, it follows that when W becomes true, W is never falsified.

Unlike weak δ-correctors, we use strong δ-correctors where we need bounded-time recovery to a state predicate closed in the execution of the program. Hence, 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.

4.2.1 Example (cont'd)

We now identify δ-correctors for each stable bounded-response property in $SPEC_{\overline{br}_{\mathcal{TC}}}$. First, consider the property $\neg S_{\mathcal{TC}} \mapsto_{\leq 3} Q$. Let $\mathcal{C}^1_{\mathcal{TC}'_i}$ 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, $\mathcal{C}^1_{\mathcal{TC}'_i}$ ensures that when $\mathcal{TC}'$ is in a state outside the invariant, it reaches a state where signal i and its previous signal are red within 3 time units. Likewise, for the property $Q_{\mathcal{TC}} \mapsto_{\leq 7} S_{\mathcal{TC}}$, let $\mathcal{C}^2_{\mathcal{TC}_i}$ 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 $\mathcal{C}^2_{\mathcal{TC}'_i}$, 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.

5. INTERFERENCE-FREEDOM OF FAULT-TOLERANCE COMPONENTS

As mentioned in Section 1, using the concept of fault-tolerance components, we propose a method for compositional verification of hard-masking properties of a real-time program via four steps. In this section, we focus on Step 2. Our goal is to develop *sufficient conditions* under which existence of correct fault-tolerance components guarantees a correct hard-masking program. Compositional proofs of interference-freedom have received substantial attention in the formal methods community in the past two decades (e.g., [1]). Drawing from these efforts, we identify simple sufficient conditions to ensure that when a program $\mathcal{P}$ is composed with a detector or a δ-corrector $\mathcal{Q}$, corresponding properties of $\mathcal{Q}$, (i.e., *Safeness*, *Stability*, *Progress*, *Weak/Strong Convergence*) are preserved. These conditions are *superposition*, *atomicity*, *order of execution*, *termination*, and *containment*. Except termination, the correctness of all these conditions can be verified syntactically making them useful in compositional verification.

In order to study the effect of programs and fault-tolerance components on each other and their possible interference, we focus on to types of commonly considered program composition: *parallel* and *sequential composition*. Intuitively, the parallel composition of two programs $\mathcal{P}$ and $\mathcal{P}'$ interleaves non deterministically between (timed and delay) actions of $\mathcal{P}$ and $\mathcal{P}'$.

Definition 5.1 (parallel composition) Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ and $\mathcal{P}' = \langle V_{\mathcal{P}'}, X_{\mathcal{P}'}, GC_{\mathcal{P}'}\rangle$ be two programs[1]. The *parallel composition* of $\mathcal{P}$ and $\mathcal{P}'$ is the program $\mathcal{P}[]\mathcal{P}' = \langle V_\mathcal{P} \cup V'_\mathcal{P},\ X_\mathcal{P} \cup X'_\mathcal{P},\ GC_\mathcal{P} \cup GC_{\mathcal{P}'}\rangle$. ∎

Informally, the sequential composition of two programs $\mathcal{P}$ and $\mathcal{P}'$ is a new program that first runs $\mathcal{P}$ and then $\mathcal{P}'$.

Definition 5.2 (sequential composition) Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ and $\mathcal{P}' = \langle V_{\mathcal{P}'}, X_{\mathcal{P}'}, GC_{\mathcal{P}'}\rangle$ be two programs. The *sequential composition* of $\mathcal{P}$ and $\mathcal{P}'$ is the program $\mathcal{P};\mathcal{P}' = \langle V_\mathcal{P} \cup V'_\mathcal{P},\ X_\mathcal{P} \cup X'_\mathcal{P},\ GC_{\mathcal{P};\mathcal{P}'}\rangle$ such that $\Pi_{\mathcal{P};\mathcal{P}'} = \{\overline{\alpha}\overline{\beta} \mid \overline{\alpha} \in \mathit{prefix}(\Pi_\mathcal{P}) \wedge \overline{\beta} \in \Pi_{\mathcal{P}'})\}$. ∎

5.1 Superposition

The first sufficient condition is *superposition*. Roughly speaking, a program $\mathcal{P}$ is superposed on a component $\mathcal{Q}$, if execution of $\mathcal{P}$ has no impact on concurrent execution of $\mathcal{Q}$. Characterizing superposition for real-time systems is more challenging due to the existence of timing constraints. To address this issue, we define the notion of *delay-compatible components*.

Definition 5.3 (delay-compatibility) Let $\mathcal{P}_1 = \langle V_{\mathcal{P}_1}, X_{\mathcal{P}_1}, GC_{\mathcal{P}_1}\rangle$ and $\mathcal{P}_2 = \langle V_{\mathcal{P}_2}, X_{\mathcal{P}_2}, GC_{\mathcal{P}_2}\rangle$ be two components and U be a state predicate. We say that $\mathcal{P}_1$ is *delay-compatible* with $\mathcal{P}_2$ from U iff

$$\begin{aligned} U \Rightarrow (\ &\forall gc_1 | gc_1 \text{ is a delay action in } GC_{\mathcal{P}_1} :: \\ &\exists gc_2 | gc_2 \text{ is a delay action in } GC_{\mathcal{P}_2} :: \\ &(\mathit{guard}(gc_1) \Rightarrow \mathit{guard}(gc_2))). \ \blacksquare \end{aligned}$$

[1]Observe that our fault-tolerance components have all the characteristics of Definition 2.2. Thus, from this point of the paper, when we refer to a program in definitions or theorems, it obviously covers the notion of components as well.

In other words, $\mathcal{P}_1$ is delay-compatible with $\mathcal{P}_2$ if and only if taking delays by $\mathcal{P}_1$ is concurrently permitted by $\mathcal{P}_2$ as well. In addition to delay-compatibility, another key factor for meeting liveness properties (i.e., *Progress*, *Stability*, and *Weak/Strong Convergence*) of detectors and δ-correctors is eventual execution of actions whose guards are constantly true. Hence, we assume that programs need to satisfy the following *fairness* condition.

Assumption 5.4 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. ∎

Theorem 5.5 *(superposition) Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{C} = \langle V_\mathcal{C}, X_\mathcal{C}, GC_\mathcal{C}\rangle$ be a strong (respectively, weak) δ-corrector component in which W strongly (respectively, weakly) corrects C within δ from U.*
If

- *$\mathcal{C}$ does not read any variable in $V_\mathcal{P}$ and $X_\mathcal{P}$,*
- *$\mathcal{P}$ only reads the discrete variables in $V_\mathcal{C}$ written by $\mathcal{C}$ and it cannot reset the clock variables in $X_\mathcal{C}$, and*
- *$\mathcal{P}$ is delay-compatible with $\mathcal{C}$,*

then

- *W strongly (respectively, weakly) corrects C within δ in $\mathcal{P}[]\mathcal{C}$ from U.* ∎

The superposition theorem is valid for detector components as well. It suffices to substitute the properties of a δ-corrector with properties of a detector in Theorem 5.5.

Theorem 5.6 *(superposition) Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{D} = \langle V_\mathcal{D}, X_\mathcal{D}, GC_\mathcal{D}\rangle$ be a detector component in which W detects D from U.*
If

- *$\mathcal{D}$ does not read or write any variable in $V_\mathcal{P}$ and $X_\mathcal{P}$,*
- *$\mathcal{P}$ only reads the discrete variables in $V_\mathcal{D}$ written by $\mathcal{D}$ and it cannot reset the clock variables in $X_\mathcal{D}$, and*
- *$\mathcal{P}$ is delay-compatible with $\mathcal{D}$,*

then

- *W detects D in $\mathcal{P}[]\mathcal{D}$ from U.* ∎

Superposition can often be characterized by simpler constraints. For instance, if the set of variables of a program and a component are disjoint, then the read/write restrictions in Theorems 5.5 and 5.6 are automatically satisfied. Thus, delay-compatibility becomes the sole constraint to be checked in order to prove interference-freedom of the composed components.

5.2 Atomicity

Notice that our fairness assumption is necessary for meeting liveness properties and in particular, *Progress* of superposed components. In other words, if the components are not executed fairly, then premises of Theorems 5.5 and 5.6 do not suffice to demonstrate non-interference of components. However, this fairness assumption is not always necessary. For instance, in many fault-tolerant systems, the recovery mechanism involves only a single step that leads a program back to its normal behavior. Thus, yet another condition for satisfying non-interference of composed components is to require that the component of interest makes progress for detection or correction in one atomic step.

Definition 5.7 (atomic detector) Let $\mathcal{D}$ be a non-zeno detector with detection predicate D and witness predicate W, and $\overline{\sigma} = (\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$ be in $\Pi_{\mathcal{D}}$. We say that $\mathcal{D}$ is an *atomic detector* iff for all i, $i \geq 0$, if $\sigma_i \models D$, then $\tau_i < \tau_{i+1}$ or $\sigma_{i+1} \models (W \vee \neg D)$. ∎

In other words, in an atomic detector component, if the detection predicate holds in a state, then the component takes permitted delays or satisfies *Progress* immediately. Notice that if the component is allowed to exhibit zeno-behaviors, then it has to establish the witness predicate without taking delays. Otherwise, there is no guarantee that the component satisfies *Progress*.

Theorem 5.8 *(atomicity) Let $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, GC_{\mathcal{P}} \rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{D} = \langle V_{\mathcal{D}}, X_{\mathcal{D}}, GC_{\mathcal{D}} \rangle$ be a detector component in which W detects D from U.*
If

- *for all $\overline{\sigma} \in \Pi_{\mathcal{P}}$ and $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models (U \wedge W)$, then $\sigma_{i+1} \models (W \vee \neg D)$,*
- *$\mathcal{D}$ is atomic, and*
- *$\mathcal{P}$ is delay-compatible with $\mathcal{D}$,*

then

- *W detects D in $\mathcal{P}[]\mathcal{D}$ from U.* ∎

Definition 5.9 (atomic δ-corrector) Let $\mathcal{C}$ be a non-zeno δ-corrector with correction predicate C and witness predicate W, and $\overline{\sigma} = (\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$ be in $\Pi_{\mathcal{C}}$. We say that $\mathcal{C}$ is an *atomic δ-corrector* iff for all i, $i \geq 0$, if $\sigma_i \not\models W$, then $\tau_i < \tau_{i+1}$ or $\sigma_{i+1} \models (C \wedge W)$. ∎

In other words, in an atomic δ-corrector component, if the witness predicate does not hold in a state, then the component takes permitted delays or establishes both the correction and witness predicate by one change of location.

Theorem 5.10 *(atomicity) Let $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, GC_{\mathcal{P}} \rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{C} = \langle V_{\mathcal{C}}, X_{\mathcal{C}}, GC_{\mathcal{C}} \rangle$ be a strong δ-corrector component in which W strongly corrects C within δ from U.*
If

- *for all $\overline{\sigma} \in \Pi_{\mathcal{P}}$ and $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models (U \wedge W)$, then $\sigma_{i+1} \models W$,*
- *$\mathcal{C}$ is atomic, and*
- *$\mathcal{P}$ is delay-compatible with $\mathcal{C}$,*

then

- *W strongly corrects C within δ in $\mathcal{P}[]\mathcal{C}$ from U.* ∎

The theorem for weak δ-correctors is a bit different from Theorem 5.10. Since these components do not have to exhibit *Stability*, and closure of the witness predicate, the first condition of Theorem 5.10 is not required for weak δ-correctors.

5.3 Order of Execution

As mentioned earlier, a program may often undo a component's efforts to satisfy *Progress* and establish its witness predicate. Another way to ensure that a program does not interfere with *Progress* of a component is to run them in order. In other words, the program only observes the execution of the component and is allowed to execute only when the component satisfies *Progress* and establishes its witness predicate. Such an order in execution of components can be captured by sequential composition. Roughly speaking, if we sequentially compose a strong δ-corrector and a program, the program does not interfere with the strong δ-corrector, as it is allowed to run only when the strong δ-corrector completes its correction. Thus, the only consideration for the program is preserving the closure of witness predicate of the strong δ-corrector. To formulate this constraint, we denote the constrained execution of a program $\mathcal{P}$ by state predicate W by $W \triangleright \mathcal{P}$ (i.e., computations of $\mathcal{P}$ where W always holds).

Theorem 5.11 *(order of execution) Let $\mathcal{P}$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{C}$ be a strong δ-corrector component in which W strongly corrects C within δ from U. It is the case that W strongly corrects C within δ in $\mathcal{C}; (W \triangleright \mathcal{P})$ from U.* ∎

Observe that delay-compatibility is not an issue here, as the execution of the component and the program are not interleaved. In case of weak δ-correctors, since *Strong Convergence* and *Stability* are not required, constrained execution of $\mathcal{P}$ is unnecessary and simple sequential composition of a weak δ-corrector and the program preserves all the properties of the weak δ-corrector.

Theorem 5.12 *(order of execution) Let $\mathcal{P}$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{C}$ be a weak δ-corrector component in which W weakly corrects C within δ from U. It is the case that W weakly corrects C within δ in $\mathcal{C}; \mathcal{P}$ from U.* ∎

In the case of sequential composition of a detector and a program, the only consideration is that the program does not violate *Stability* of witness predicate of the detector, when the detector stops working.

Theorem 5.13 *(order of execution) Let $\mathcal{P}$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{D}$ be a detector component in which W detects D from U.*
If

- *for all $\overline{\sigma} \in \Pi_{\mathcal{P}}$ and $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models (U \wedge W)$, then $\sigma_{i+1} \models (W \vee \neg D)$,*

then

- *W detects D in $\mathcal{D}; \mathcal{P}$ from U.* ∎

5.4 Termination

Another alternative that guarantees non-interference of a program $\mathcal{P}$ with *Progress* of a component $\mathcal{Q}$ is to force $\mathcal{P}$ to terminate. It follows that after $\mathcal{P}$ has terminated, execution of $\mathcal{Q}$ in isolation satisfies it *Progress*. The termination condition is in some sense the opposite of the order of execution condition presented in the previous subsection. That is, in the order of execution, the component completes its job and the program runs subsequently, whereas in termination, the program stops working at some point in order to let the component make progress. Termination of the program can be enforced by, for instance, reaching a particular state predicate.

Theorem 5.14 *(termination) Let $\mathcal{P}$ be a program, U and V be two state predicates closed in $\mathcal{P}$, and $\mathcal{D}$ be a detector component in which W detects D from U.*
If

- *$\mathcal{P}[]\mathcal{D}$ refines $U \mapsto_{\leq\infty} V$,*

then

- *W detects D in $(\neg V \triangleright \mathcal{P})[]\mathcal{D}$ from U.* ∎

In case of δ-correctors, the only condition that has to be taken into account is to ensure that there is an appropriate time bound on termination of $\mathcal{P}$. Otherwise, it is not possible to reason about the worst case correction time of the composed program.

Theorem 5.15 *(termination) Let $\mathcal{P}$ be a program, U and V be two state predicates closed in $\mathcal{P}$, and $\mathcal{C}$ be a weak/strong δ-corrector component in which W weakly/strongly corrects C within δ from U.*
If

- *$\mathcal{P}[]\mathcal{C}$ refines $U \mapsto_{\leq\theta} V$, for some $\theta \in \mathbb{Z}_{\geq 0}$,*

then

- *W weakly/strongly corrects C within $\delta + \theta$ in $(\neg V \triangleright \mathcal{P})[]\mathcal{C}$ from U.* ∎

5.5 Containment

A straightforward sufficient condition for non-interference is *containment*. Containment requires that the computations of a program $\mathcal{P}$ is a subset of the computations of the component that contains it. This condition occurs, for instance, when a program acts as a detector (or a δ-corrector). Thus the containment theorems are as follows.

Theorem 5.16 *(containment) Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{D} = \langle V_\mathcal{D}, X_\mathcal{D}, GC_\mathcal{D}\rangle$ be a detector component in which W detects D from U. If $\Pi_\mathcal{P} \subseteq \Pi_\mathcal{D}$, then W detects D in $\mathcal{P}[]\mathcal{D}$ from U.* ∎

Theorem 5.17 *(containment) Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$ and $\mathcal{C} = \langle V_\mathcal{C}, X_\mathcal{C}, GC_\mathcal{C}\rangle$ be a strong (respectively, weak) δ-corrector component in which W strongly (respectively, weakly) corrects C within δ from U. If $\Pi_\mathcal{P} \subseteq \Pi_\mathcal{C}$, then W strongly (respectively, weakly) corrects C within δ in $\mathcal{P}[]\mathcal{C}$ from U.* ∎

Since we often represent programs and components in terms of timed guarded commands, containment can be shown by a straightforward syntactical test for the components to be composed. Our experience with design and analysis of fault-tolerant programs shows that containment is often useful to reason about detector components. This is due to the fact that detectors ensure satisfaction of timing-independent part of a safety specification, which has to be maintained in every computation step of the program. Thus, the fault-tolerant program often has to act as a detector component as well.

5.6 Example (cont'd)

To prove interference-freedom between and $\mathcal{D}^1_{\mathcal{TC}'_i}$ and $\mathcal{P} = \{\mathcal{TC}'3_i\}$, we apply Theorem 5.13. It is straightforward to see that (1) $\mathcal{P}$ can be executed only when $\mathcal{D}^1_{\mathcal{TC}'_i}$ establishes the witness predicate, and (2) $\mathcal{P}$ satisfies the first condition of Theorem 5.13, as it immediately falsifies the detection predicate. In case of $\mathcal{D}^2_{\mathcal{TC}'_i}$, we also apply Theorem 5.13 where $\mathcal{P} = \{\mathcal{TC}'6_i, \mathcal{TC}'7_i, \mathcal{TC}'8_i\}$ as follows. Obviously, $\mathcal{P}$ can only execute when $\mathcal{D}^2_{\mathcal{TC}'_i}$ establishes $W_{\mathcal{D}^2_{\mathcal{TC}'_i}}$ (i.e., it turns both signals red). Moreover, $\mathcal{P}$ preservers the witness predicate of $\mathcal{D}^2_{\mathcal{TC}'_i}$, as it leaves the state of both signals unchanged. Hence, to verify that $\mathcal{TC}'$ never reaches a state where any two signals are not red even in the presence of faults, it suffices to verify the correctness of $\mathcal{D}^1_{\mathcal{TC}'_i}$ and $\mathcal{D}^2_{\mathcal{TC}'_i}$.

An alert reader can easily prove non-interference of the δ-correctors with each other and with the program by showing that they run in order (i.e., by applying Theorems 5.11 and 5.12). Alternatively, one can exploit the fact that both δ-correctors are atomic (i.e., by applying Theorem 5.10). Hence, to verify that $\mathcal{TC}'$ never violates its timing constraints even in the presence of faults, it suffices to verify the correctness of $\mathcal{C}^1_{\mathcal{TC}'_i}$ and $\mathcal{C}^2_{\mathcal{TC}'_i}$.

Table 1 summarizes verification of fault-tolerance properties of $\mathcal{TC}'$ with 100-500 traffic signals using the model checker UPPAAL [21]. All experiments are run on a PC with a 3GHz Pentium 4 processor and 1GB RAM. All times are in seconds and all memory usages are in megabytes.

As can be seen, verification of $\mathcal{TC}'$ as a whole requires considerable more resources than the case where we verify it compositionally. For instance, verification of deadlock-freedom in $\mathcal{TC}'$ with 200 traffic signals needs 1150MB of memory and 70s to complete, whereas verifying the same property in the fault-intolerant program and δ-correctors needs a total of 11.9s and a maximum of 500MB of main memory. Observe that verification of deadlock-freedom is accomplished in a compositional manner as follows: it suffices to verify this property for the fault-intolerant program and δ-correctors independently. This is due to the fact that when a strong δ-corrector establishes the witness predicate during recovery, the invariant predicate is reached. Reaching the invariant predicate in turn guarantees that only the fault-intolerant program operates (i.e., another application of order of execution).

As we increase the number of traffic signals, the advantage of our method becomes more clear. For instance, in case of $\mathcal{TC}'$ with 250 traffic signals, UPPAAL goes out of memory when verifying the recovery timing constraints in $SPEC_{\overline{br}}$. On the other hand, verifying the correctness of δ-correctors to show the same property is not even close to the limits.

	Compositional Verification						*Non-Compositional Verification*					
	Detectors		Correctors		Fault-Intolerant		$SPEC_{\overline{bt}}$		$SPEC_{\overline{br}}$		Deadlock-Freedom	
	memory	time	memory	time	memory	time	memory	time	memory	time	memory	time
$\mathcal{TC}^{100}$	70	2	38	1	112	2	146	9	250	9	350	19
$\mathcal{TC}^{200}$	96	5	40	1.9	500	10	220	50	1200	59	1150	70
$\mathcal{TC}^{250}$	155	19	42	2.6	900	30	340	118	-	-	2100	180
$\mathcal{TC}^{300}$	177	29	41	3.4	1650	90	500	178	-	-	-	-
$\mathcal{TC}^{350}$	290	60	45	4.2	2200	260	790	240	-	-	-	-
$\mathcal{TC}^{400}$	330	120	45	5	-	-	1100	357	-	-	-	-
$\mathcal{TC}^{500}$	524	262	53	6.6	-	-	-	-	-	-	-	-

Table 1: Experimental results for circular traffic signals.

Since no two δ-correctors write each other's variables, and they are delay compatible, we can verify them separately. This is the main reason that verification of δ-correctors in Table 1 require negligible resources.

Finally, we note that similar to other compositional verification approaches, our method may not perform well if the fault-tolerance components of a program are tightly intertwined. For instance, some programs act as both a detector and a corrector and it is not possible to decompose them into completely disjoint components. One example of such programs is Dijkstra's token ring mutual exclusion algorithm [13], where recovery actions are embedded in the normal actions of the fault-tolerant program.

6. RELATED WORK

Compositional verification has mostly been studied in the context of assume-guarantee methods where properties are decomposed into two parts. One is an assumption about the global behavior of the environment of the component; the other is a property guaranteed by the component when the assumption about its environment holds [2, 6, 11, 15, 23]. As discussed in [12], issues such as finding decompositions into sub-systems and choosing adequate assumptions for a particular decomposition make the application of assume-guarantee rules difficult.

The theory of detectors and correctors was first introduced in [7] in the context of *untimed* systems. The theory was extended in [18] for safety-critical systems and in [24] for proving convergence of systems to legitimate states. The theory has also been used in design of several multi-tolerant examples [14, 19], 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 to real-time programs, the theory has been exploited in [8, 9]. In the context of verification, simplified versions of this theory are applied in verification of time-triggered architectures [22] and in [17, 20] for software verification through separation of concerns using the theorem prover PVS.

In [10], we extended the theory to the context of real-time programs and we showed the necessity of existence of fault-tolerance real-time components in hard-masking programs. This work differs from the previous work in that we introduce sufficient conditions for non-interference of fault-tolerance real-time components. These conditions enable us to compositionally verify fault-tolerant real-time program. Moreover, this work demonstrates the first application of the theory of fault-tolerance real-time components in model checking.

7. CONCLUSION AND FUTURE WORK

The theory of fault-tolerance real-time components proposed in [10] separates fault-tolerance and functionality concerns of real-time systems. It identifies three types of components, namely *detectors*, *weak δ-correctors*, and *strong δ-correctors* and shows that the *necessary condition* for a real-time program to be fault-tolerant is to contain a set of these components based upon the safety and liveness specifications that it has to satisfy in the presence of faults. In this paper, we enriched the theory by identifying various *sufficient conditions* to show interference-freedom among fault-tolerance components and programs. A majority of these conditions can be verified via simple syntactic methods.

A direct application of the sufficient conditions is in compositional verification of fault-tolerance properties as follows. Given a real-time program one can (1) decompose the program into a fault-intolerant version and a set of fault-tolerance components using the method presented in [10], (2) demonstrate non-interference among the intolerant program and the components, (3) verify the correctness of the intolerant program against functional properties of the program, and (4) verify the correctness of each component. Our preliminary experiments show a considerable improvement in memory usage and achieving speedups. To our knowledge, this work is the first in applying the theory of fault-tolerance components in model checking.

One future research direction is to automatically synthesize fault-tolerance real-time components. This enables us to design fault-tolerant programs using pre-synthesized components that are correct-by-construction. Another future work is to extend the theory in the context of multi-tolerant real-time programs, where programs must exhibit a different level of tolerance based on severity of faults. Such programs have more complex structure and, hence, more difficult to design and verify.

8. REFERENCES

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):507–534, May 1995.

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

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

[5] 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.

[6] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[7] 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.

[8] 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.

[9] 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.

[10] B. Bonakdarpour, S. S. Kulkarni, and A. Arora. Disassembling real-time fault-tolerant programs. In *ACM International Conference on Embedded Software (EMSOFT)*, pages 169–178, 2008.

[11] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Logic in Computer Science (LICS)*, pages 353–362, 1989.

[12] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2), 2008.

[13] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

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

[15] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

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

[17] R. D. Jeffords, C. L. Heitmeyer, and M. Archer. Adding fault-tolerance to requirements specifications. Under review - Personal communication, 2007.

[18] 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.

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

[20] 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.

[21] K. G. Larsen, P. Pattersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[22] 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.

[23] E. W. Stark. A proof technique for rely/guarantee properties. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 369–391, 1985.

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

APPENDIX

A. SUMMARY OF NOTATION

$\mathbb{Z}_{\geq 0}$	nonnegative integers
$\mathbb{R}_{\geq 0}$	nonnegative reals
V	set of discrete variables
X	set of clock variables
$\Phi(X)$	set of all clock constraints over X
σ	state
$\overline{\sigma}$	computation
$\mapsto_{\leq}$	stable bounded response
S	state predicate or program invariant
T	fault-span predicate
D	detection predicate
W	witness predicate
C	correction predicate
$\mathcal{P}$	real-time program
$GC_{\mathcal{P}}$	set of guarded commands of program $\mathcal{P}$
$\mathcal{D}$	detector component
$\mathcal{C}$	δ-corrector component
$X \triangleright \mathcal{P}$	constrained program $\mathcal{P}$ by state predicate W
$\mathcal{TC}$	traffic controller program
Π	set of computations
f	set of fault transitions
$\bullet$	fusion operator
$[]$	transition insertion operator
$SPEC$	specification
$SPEC_{bt}$	set of bad immediate transitions
$SPEC_{\overline{bt}}$	safety specification characterized by $SPEC_{bt}$
$SPEC_{\overline{br}}$	timing dependent safety specification

B. PROOFS

Theorem 5.5 *Let $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, GC_{\mathcal{P}}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{C} = \langle V_{\mathcal{C}}, X_{\mathcal{C}}, GC_{\mathcal{C}}\rangle$ be a strong (respectively, weak) δ-corrector component in which W strongly (respectively, weakly) corrects C within δ from U.*
If

- *$\mathcal{C}$ does not read any variable in $V_{\mathcal{P}}$ and $X_{\mathcal{P}}$,*
- *$\mathcal{P}$ only reads the discrete variables in $V_{\mathcal{C}}$ written by $\mathcal{C}$ and it cannot reset the clock variables in $X_{\mathcal{C}}$, and*
- *$\mathcal{P}$ is delay-compatible with $\mathcal{C}$,*

then

- *W strongly (respectively, weak) corrects C within δ in $\mathcal{P}[]\mathcal{C}$ from U.*

Proof. We need to show that $\mathcal{P}[]\mathcal{C}$ satisfies *Safeness*, *Stability*, *Progress*, and *Weak/Strong Convergence*. Consider a computation $\overline{\sigma} = (\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots$ of $\Pi_{\mathcal{P}[]\mathcal{C}}$:

- (*Safeness*) Consider a state σ_i, $i \geq 0$, of $\overline{\sigma}$ where $\sigma \models U \wedge W$. Now, if $\sigma \not\models U \wedge C$, then it must be the case that $\mathcal{P}[]\mathcal{C}$ has reached a state that neither $\mathcal{P}$ nor $\mathcal{C}$ could reach. However, this is in contradiction with the fact that U is closed in $\mathcal{P}$ and $\mathcal{C}$ satisfies *Safeness* from U. Hence, $\sigma \models (U \Rightarrow (W \Rightarrow C))$.
- (*Progress*) Let σ_i, $i \geq 0$, be a state of $\overline{\sigma}$ where $\sigma \models C$. Now, if C constantly remains true, but there does not exist a state σ_k, $k \geq i$, such that $\sigma_k \models W$, then it must be the case that the value of variables of the corrector $\mathcal{C}$ never establish the state predicate W. Obviously, W is not eventually established only if one or a combination of the following occurs: (1) $\mathcal{P}$ manipulates a variable in $V_{\mathcal{C}}$, (2) the guard of actions of $\mathcal{C}$ depend on the value of variables of $\mathcal{P}$, but $\mathcal{P}$ never lets the guard of actions of $\mathcal{C}$ establish so that $\mathcal{C}$ can progress, or (3) $\mathcal{P}$ takes a delay that leads $\mathcal{P}[]\mathcal{C}$ to a state where $\mathcal{C}$ cannot progress. However, the above scenarios are in contradiction with the three assumptions of the theorem, respectively. An intuitive argument of the above proof by contradiction is as follow. Consider an abstraction of computation $\overline{\sigma}$ on variables of δ-corrector $\mathcal{C}$. In this abstraction, (1) location changes of $\mathcal{P}$ are stutter (due to the read/write restrictions), and (2) delay actions of $\mathcal{P}$ are delay actions of $\mathcal{C}$ (due to delay-compatibility of $\mathcal{P}$ with $\mathcal{C}$). Hence, *Progress* of $\mathcal{P}[]\mathcal{C}$ is met.
- (*Strong/Weak Convergence*) Three scenarios can cause violation of *Strong/Weak Convergence* in $\mathcal{P}[]\mathcal{C}$: (1) state predicate C is not reached, (2) it is reached, but not within δ time units, and (3) C is not closed (for strong δ-correctors only). The first scenario cannot happen, since it requires that $\mathcal{P}$ either (1) changes the value of some variables in $V_{\mathcal{C}}$, (2) takes delays longer than $\mathcal{C}$ is allowed to take and, hence, causes $\mathcal{C}$ not to progress, or (3) guard of actions of $\mathcal{C}$ are evaluated using variables of $\mathcal{P}$. However, these reasons all contradict with the premises of the theorem. The second scenario contradicts with delay-compatibility of $\mathcal{P}$ and $\mathcal{C}$. Finally, the last scenario (violation of closure of C), contradicts with the fact that $\mathcal{P}$ cannot write the discrete variables or reset the clock variables of of $\mathcal{C}$. Thus, *Strong Convergence* of $\mathcal{P}[]\mathcal{C}$ holds.
- (*Stability*) First, note that the proof of *Stability* is not needed for weak δ-correctors. This property is violated when the truthfulness of the witness predicate W never gets stabled, i.e., it alternates between W and $\neg W$. Since $\mathcal{C}$ guarantees stability of W, the alternation situation may appear only if (1) $\mathcal{C}$ falsifies W due to the change of value of a variable of $\mathcal{P}$, or (2) $\mathcal{P}$ either modifies a variable of $\mathcal{C}$, or takes unexpected delays. However, all these scenarios are made impossible by the assumptions of the theorem. An intuitive argument of the above proof by contradiction is as follow. Consider an abstraction the computation $\overline{\sigma}$ on variables of corrector $\mathcal{C}$. In this abstraction, (1) location changes of $\mathcal{P}$ are stutter (due to the read/write restrictions), and (2) delay actions of $\mathcal{P}$ are delay actions of $\mathcal{C}$ (due to delay-compatibility of $\mathcal{P}$ with $\mathcal{C}$). Hence, *Stability* of $\mathcal{P}[]\mathcal{C}$ holds.

Note that since U is closed in both $\mathcal{P}$ and $\mathcal{C}$, closure of U in $\mathcal{P}[]\mathcal{C}$ immediately follows. Finally, since our fairness assumption holds for both $\mathcal{P}$ and $\mathcal{C}$ as well as $\mathcal{P}[]\mathcal{C}$, *Progress*, *Stability*, and *Weak/Strong Convergence* constraints are guaranteed to hold in $\mathcal{P}[]\mathcal{C}$. ∎

Theorem 5.6 *Let $\mathcal{P} = \langle V_{\mathcal{P}}, X_{\mathcal{P}}, GC_{\mathcal{P}}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{D} = \langle V_{\mathcal{D}}, X_{\mathcal{D}}, GC_{\mathcal{D}}\rangle$ be a detector component in which W detects D from U.*
If

- *$\mathcal{D}$ does not read or write any variable in $V_{\mathcal{P}}$ and $X_{\mathcal{P}}$,*

- *$\mathcal{P}$ only reads the discrete variables in $V_\mathcal{D}$ written by $\mathcal{D}$ and it cannot reset the clock variables in $X_\mathcal{D}$, and*
- *$\mathcal{P}$ is delay-compatible with $\mathcal{D}$,*

then

- *W detects D in $\mathcal{P}[]\mathcal{D}$ from U.*

Proof. The proof is identical to proof of Theorem 5.5, except we omit the proof of *Convergence*. ∎

Theorem 5.8 *Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{D} = \langle V_\mathcal{D}, X_\mathcal{D}, GC_\mathcal{D}\rangle$ be a detector component in which W detects D from U.*
If

- *for all $\overline{\sigma} \in \Pi_\mathcal{P}$ and $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models (U \wedge W)$, then $\sigma_{i+1} \models (W \vee \neg D)$,*
- *$\mathcal{D}$ is atomic, and*
- *$\mathcal{P}$ is delay-compatible with $\mathcal{D}$,*

then

- *W detects D in $\mathcal{P}[]\mathcal{D}$ from U.*

Proof. We need to show that $\mathcal{P}[]\mathcal{D}$ satisfies *Safeness*, *Stability*, and *Progress*. Consider a computation $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ of $\Pi_{\mathcal{P}[]\mathcal{D}}$:

- (*Safeness*) Consider a state σ_i, $i \geq 0$, of $\overline{\sigma}$ where $\sigma \models U \wedge W$. Now, if $\sigma \not\models U \wedge C$, then it must be the case that $\mathcal{P}[]\mathcal{D}$ has reached a state that neither $\mathcal{P}$ nor $\mathcal{D}$ could reach. However, this is in contradiction with the fact that U is closed in $\mathcal{P}$ and $\mathcal{D}$ satisfies *Safeness* from U. Hence, $\sigma \models (U \Rightarrow (W \Rightarrow D))$.
- (*Progress*) Let σ_i, $i \geq 0$, be a state of $\overline{\sigma}$ where $\sigma \models D$. Now, since $\mathcal{D}$ is atomic, any non-delay action of $\mathcal{D}$ either establishes W or falsifies D. Hence, *Progress* is met.
- (*Stability*) This condition is trivially satisfied due to the first condition of the theorem.

∎

Theorem 5.10 *Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{C} = \langle V_\mathcal{C}, X_\mathcal{C}, GC_\mathcal{C}\rangle$ be a strong δ-corrector component in which W strongly corrects C within δ from U.*
If

- *for all $\overline{\sigma} \in \Pi_\mathcal{P}$ and $i \in \mathbb{Z}_{\geq 0}$, if $\sigma_i \models (U \wedge W)$, then $\sigma_{i+1} \models W$,*
- *$\mathcal{C}$ is atomic, and*
- *$\mathcal{P}$ is delay-compatible with $\mathcal{C}$,*

then

- *W strongly corrects C within δ in $\mathcal{P}[]\mathcal{C}$ from U.*

Proof. The proof of *Safeness*, *Progress*, and *Stability* is identical to the proof of Theorem 5.8. Thus, we only need to show *Strong Convergence* within δ time units. Observe that the first condition of of the theorem guarantees that $\mathcal{P}$ preserves closure of W. Moreover, since $\mathcal{C}$ is atomic the witness predicate is established as soon as $\mathcal{C}$ has a chance to execute. Finally, since $\mathcal{P}$ is delay-compatible with $\mathcal{C}$, the composed programs never takes illegal time delays. Hence, *Strong Convergence* within δ is satisfied by $\mathcal{P}[]\mathcal{C}$. ∎

Theorem 5.11 *Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{C} = \langle V_\mathcal{C}, X_\mathcal{C}, GC_\mathcal{C}\rangle$ be a strong δ-corrector component in which W strongly corrects C within δ from U. It is the case that W strongly corrects C within δ in $\mathcal{C}; (W \triangleright \mathcal{P})$ from U.*

Proof. We need to show that $\mathcal{C}; (W \triangleright \mathcal{P})$ satisfies *Safeness*, *Stability*, *Progress*, and *Strong Convergence*. Consider a computation $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ of $\Pi_{\mathcal{C};(W \triangleright \mathcal{P})}$, where $\overline{\sigma} = \overline{\alpha}\overline{\beta}$ such that $\overline{\alpha}$ is a computation prefix of $\mathcal{C}$ and $\overline{\beta}$ is in $\Pi_\mathcal{P}$.

- (*Safeness*) Consider a state σ_i, $i \geq 0$, of $\overline{\sigma}$. If σ_i is in $\overline{\alpha}$, then *Safeness* is trivially satisfied. If σ_i is in $\overline{\beta}$, then $\sigma_i \models (U \wedge W)$ in all states along $\overline{\beta}$. Now, if $\sigma_i \not\models U \wedge C$, then it must be the case that $\mathcal{C}; (W \triangleright \mathcal{P})$ has reached a state that neither $W \triangleright \mathcal{P}$ nor $\mathcal{C}$ could reach. However, this is in contradiction with the fact that U is closed in $\mathcal{P}$ and $\mathcal{C}$. Hence, $\sigma_i \models (U \Rightarrow (W \Rightarrow C))$.
- (*Progress*) Let σ_i, $i \geq 0$, be a state of $\overline{\sigma}$ where $\sigma_i \models C$. Since, $\mathcal{P}$ starts working when W holds, it has to be the case that $\mathcal{C}$ establishes W before reaching the first state in $\overline{\beta}$. Hence, *Progress* is met.
- (*Stability*) This condition is trivially satisfied due to the first condition of the theorem.
- (*Strong Convergence*) This condition is also trivially met since when $\mathcal{C}$ is executing, $\mathcal{P}$ does not interfere in any ways with $\mathcal{C}$. Thus, it has to be the case that $\mathcal{C}$ reaches a state in C within δ time units. Hence, *Strong Convergence* is satisfied.

∎

The proof of Theorems 5.12 and 5.13 are identical to the proof of Theorem 5.11, except the redundant conditions (i.e., *Strong Convergence* for detectors, and, *Stability* and closure of witness predicate for weak δ-correctors) can be omitted.

Theorem 5.14 *Let $\mathcal{P}$ be a program, U and V be two state predicates closed in $\mathcal{P}$, and $\mathcal{D}$ be a detector component in which W detects D from U.*
If

- *$\mathcal{P}[]\mathcal{D}$ refines $U \mapsto_{\leq \infty} V$,*

then

- *W detects D in $(\neg V \triangleright \mathcal{P})[]\mathcal{D}$ from U.*

Proof. We need to show that $(\neg V \triangleright \mathcal{P})[]\mathcal{D}$ satisfies *Safeness*, *Stability*, and *Progress*. Consider a computation $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ of $\Pi_{(\neg V \triangleright \mathcal{P})[]\mathcal{D}}$:

- (*Safeness*) Consider a state σ_i, $i \geq 0$, of $\overline{\sigma}$. If $\sigma_i \models (U \wedge W)$, but $\sigma_i \not\models U \wedge D$, then it must be the case that $(\neg V \triangleright \mathcal{P})[]\mathcal{D}$ has reached a state that neither $\mathcal{P}$ nor $\mathcal{D}$ could reach. However, this is in contradiction with the fact that U is closed in $\mathcal{P}$ and $\mathcal{D}$. Hence, $\sigma_i \models (U \Rightarrow (W \Rightarrow D))$.
- (*Progress*) Let σ_i, $i \geq 0$, be a state of $\overline{\sigma}$ where $\sigma_i \models D$. Since, $U \mapsto_{\leq \infty} V$, there exists a state σ_k, where V becomes true. At this point $\mathcal{P}$ stops working and $\mathcal{D}$ is executed in isolation. Now, since $\mathcal{D}$ satisfies *Progress*, there has to be a state σ_j such that $j \geq i$ and $\sigma_j \models W$. Hence, *Progress* is met.

- (*Stability*) This condition is trivially satisfied since $\mathcal{P}$ is is closed in V and when V is established, $\mathcal{D}$ is executed in isolation, which in turn guarantees *Stability*.

∎

Theorem 5.15 *Let $\mathcal{P}$ be a program, U and V be two state predicates closed in $\mathcal{P}$, and $\mathcal{C}$ be a weak/strong δ-corrector component in which W weakly/strongly corrects C within δ from U.*
If

- *$\mathcal{P}[]\mathcal{C}$ refines $U \mapsto_{\leq\theta} V$, for some $\theta \in \mathbb{Z}_{\geq 0}$,*

then

- *W weakly/strongly corrects C within $\delta+\theta$ in $(\neg V \triangleright \mathcal{P})[]\mathcal{C}$ from U.*

Proof. We need to show that $(\neg V \triangleright \mathcal{P})[]\mathcal{C}$ satisfies *Safeness*, *Stability*, *Progress*, and *Weak/Strong Convergence*. The proof of *Safeness*, *Progress*, and *Stability* are identical to those of Theorem 5.14. Thus, we only need to show that $(\neg V \triangleright \mathcal{P})[]\mathcal{C}$ satisfies *Convergence* in $\delta + \theta$. Consider a computation $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ of $\Pi_{(\neg V \triangleright \mathcal{P})[]\mathcal{C}}$. Since $\mathcal{P}[]\mathcal{C}$ refines $U \mapsto_{\leq\theta} V$ and $\sigma_0 \models U$, then there exists a state σ_k such that $\sigma_k \models V$ and $\tau_k \leq \theta$. At this point, $\mathcal{P}$ stops working and $\mathcal{C}$ proceeds in isolation. Since $\mathcal{C}$ satisfies *Weak/Strong Convergence* to C within δ time units, $(\neg V \triangleright \mathcal{P})[]\mathcal{C}$ also reaches a state σ_j such that $\sigma_j \models C$ and $\tau_j - \tau_k \leq \delta$. Hence, $(\neg V \triangleright \mathcal{P})[]\mathcal{C}$ satisfies *Weak/Strong Convergence* within $\theta + \delta$. ∎

Theorem 5.16 *Let $\mathcal{P} = \langle V_\mathcal{P}, X_\mathcal{P}, GC_\mathcal{P}\rangle$ be a program, U be a state predicate such that U is closed in $\mathcal{P}$, and $\mathcal{D} = \langle V_\mathcal{D}, X_\mathcal{D}, GC_\mathcal{D}\rangle$ be a detector component in which W detects D from U. If $\Pi_\mathcal{P} \subseteq \Pi_\mathcal{D}$, then W detects D in $\mathcal{P}[]\mathcal{D}$ from U.*

Proof. The proof is trivial. Since $\Pi_\mathcal{P} \subseteq \Pi_\mathcal{D}$, we have $\Pi_{\mathcal{P}[]\mathcal{D}} = \Pi_\mathcal{D}$. Hence, '$W$ detects D from U' holds in all computations of $\mathcal{P}[]\mathcal{D}$. ∎

The proof of Theorem 5.17 is identical to the proof of Theorem 5.16.