Designing Run-Time Fault-Tolerance Using Dynamic Updates*

Ali Ebnenasir
Department of Computer Science
Michigan Technological University
Houghton MI 49931, USA
aebnenas@mtu.edu

Abstract

We present a framework for designing run-time fault-tolerance using dynamic program updates triggered by faults. This is an important problem in the design of autonomous systems as it is often the case that a running program needs to be upgraded to its fault-tolerant version once faults occur. We formally state fault-triggered program updates as a design problem. We then present a sound and complete algorithm that automates the design of fault-triggered updates for replacing a program that does not tolerate faults with a fault-tolerant version thereof at run-time. We also define three classes of fault-triggered dynamic updates that tolerate faults during the update. We demonstrate our approach in the context of a fault-triggered update for the gate controller of a parking lot.

1 Introduction

Tolerating *unanticipated* faults at run-time is an important property of autonomous systems (e.g., Mars Exploration Rover) as such systems should guarantee a minimum level of behavioral correctness in the presence of unexpected environmental events. Since it is difficult (if not impossible) to anticipate all types of faults and to design systems that tolerate all fault-types, it is highly desirable to dynamically upgrade the controlling software of autonomous systems after the first occurrence of an unanticipated type of faults is detected so that subsequent occurrences of faults are tolerated. Such a run-time update should take place while satisfying (possibly a weaker version of) the program specification during the update. In this paper, we report our ongoing work on the development of a framework for the design of run-time fault-tolerance based on fault-triggered updates.

Several approaches formalize and present solutions for dynamic replacement of an *old* running program with a *new* program [18, 14, 15, 26, 2] most of which focus on the architectural and programming issues with less emphasis on the effect of faults and fault-tolerance during updates. For example, Kramer and Magee [18] present a framework for specifying and implementing dynamic updates at the architectural (components/connectors) level, thereby separating the functional updates from structural updates. Gupta and Jalote [14] present a functional approach for transferring the state of the old program to a state of the new program where the transfer functions are developed offline. Gupta *et al.* [15] propose a formal framework for designing and reasoning about dynamic updates, where online change is considered to be an *instantaneous* process using a transfer function (defined by developers). Duggan [12] presents a type-based hot swapping approach where programmers define two-way mappings that are reflected as type-sharing constraints during update. Stoyle *et al.* [26] introduce a safe transformer for dynamic update of imperative programs. Ajmani *et al.* [1, 2] present a middleware to support dynamic updates in distributed systems. All aforementioned approaches focus on updating programs and their components with less emphasis on automated design and dependability of such dynamic updates in the presence of faults.

We propose a *formal* framework for modeling and designing run-time fault-tolerance, where a program that does not provide any guarantees about its behavior when unanticipated faults occur (called the *fault-intolerant* program) is replaced with a fault-tolerant version thereof at run-time. Our proposed approach extends the notion of dynamic program updates in that updates are triggered by faults and tolerate faults. More specifically, we first formally define what we mean by an upgrade program in the context of fault-tolerance. We then define the problem of designing fault-triggered updates and present a sound and complete algorithm for automated design of such updates. Subsequently, we define three classes of fault-tolerant dynamic updates.

Our proposed approach for the design of run-time fault-tolerance comprises three main steps once a new type of faults is detected: (i) designing the new (upgrade) pro-

*This work was sponsored by a grant from Michigan Technological University.

gram (which is a fault-tolerant version of the running fault-intolerant program), (ii) designing a *fault-triggered update program* for dynamic (i.e., run-time) replacement of the fault-intolerant program with its fault-tolerant version due to the occurrence of faults, and (iii) adding fault-tolerance to the update program. In our previous work [13, 24], we have investigated the first step. The third step is currently under investigation. The focus of this paper is on the second step. Specifically, in the absence of faults, the running fault-intolerant program satisfies its safety and liveness specifications [4], where a safety specification states that nothing bad ever happens and a liveness specification stipulates that something good will eventually occur. We present a sound and complete algorithm that takes the design of the running fault-intolerant program (represented as a finite state machine) and the design of its fault-tolerant version (which is designed in an offline fashion), and determines whether there exists an *update program* that enables the dynamic update of the fault-intolerant program with its fault-tolerant version when faults occur. This way, subsequent occurrences of faults would be tolerated by the fault-tolerant program. The update program should meet three properties: *progress, safeness* and *interference-freedom*. The progress property ensures that the update eventually occurs. The safeness property guarantees that during update the safety specification of the fault-intolerant program will be preserved. Moreover, in the absence of faults, the update program must not *interfere* with the execution of the fault-intolerant/tolerant program. The soundness of our algorithm guarantees that the generated update program is correct-by-construction (i.e., meets safeness, progress and interference-freedom). The completeness of our algorithm ensures that if our algorithm fails to find an update program, then the fault-triggered update of the fault-intolerant program to its fault-tolerant version is not possible (under safeness, progress and non-interference constraints).

We demonstrate our proposed approach in the context of a parking lot example, where we design a program for updating the controlling software of a parking lot gate controller with its fault-tolerant version. The rest of this paper is organized as follows: In Section 2, we present some preliminary concepts. Then in Section 3, we specify the requirements of run-time fault-tolerance and define the problem of designing run-time fault-tolerance using fault-triggered updates. In Section 4, we present an algorithm for automated design of fault-triggered updates. We demonstrate our algorithm in the context of the parking lot example in Section 5. In Section 6, we define three classes of dynamic updates that tolerate faults during update. We discuss related work in Section 7. Finally, we make concluding remarks in Section 8.

2 Preliminaries

In this section, we represent the formal definitions of programs, specifications and faults from [5, 22, 24, 13].

Program. A *program* p is of the form $\langle \mathcal{V}_p, \delta_p \rangle$, where $\mathcal{V}_p$ is a finite set of variables $\{v_1, \cdots, v_n\}$ $(n \geq 1)$ and δ_p denotes the set of program transitions. Each variable v_i has a finite domain D_i. A state of p is a valuation $\langle d_1, \cdots, d_n \rangle$ of the variables of p, where d_i is a value in the domain D_i. The set of all possible states of p comprises the state space of p, denoted S_p. The set of transitions of p, denoted δ_p , is a subset of $S_p \times S_p$. A *state predicate* of p is any subset of S_p. A state predicate X is *closed* in a program p (respectively, δ_p) iff (if and only if) $\forall s_0, s_1 : (s_0, s_1) \in \delta_p : (s_0 \in X \Rightarrow s_1 \in X)$.

A sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$ is a *computation* of p iff the following two conditions are satisfied: (1) if σ is infinite, then $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$ holds, and (2) if σ is finite and terminates in state s_f then there does not exist state s, $s \neq s_f$, such that $(s_f, s) \in \delta_p$. A sequence of states, $\langle s_0, s_1, ..., s_n \rangle$, is a *computation prefix* of p iff $\forall j : 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p$.

The *projection* of a program p on a state predicate X, denoted as $p|X$, is the program $\langle \mathcal{V}_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \ \wedge \ s_0, s_1 \in X\} \rangle$. In other words, $p|X$ consists of transitions of p that start in X and end in X.

The union of two programs $p_1 = \langle \mathcal{X}, \delta_1 \rangle$ and $p_2 = \langle \mathcal{Y}, \delta_2 \rangle$ is a program $p_u = \langle \mathcal{X} \cup \mathcal{Y}, \delta_u \rangle$, where $\mathcal{X} = \{x_1, \cdots, x_n\}$ and $\mathcal{Y} = \{y_1, \cdots, y_m\}$ $(m, n \geq 1)$ and $\mathcal{X} \cap \mathcal{Y} = \emptyset$. Each state in the state space of p_u, denoted S_u, is a valuation of $\langle x_1, \cdots, x_n, y_1, \cdots, y_m \rangle$. We denote by $s \uparrow_{S_u}$ the *image* of a state $s \in S_{p_1}$ (respectively, $s \in S_{p_2}$) in S_u, where $s = \langle d_1, \cdots, d_n \rangle$ (respectively, $s = \langle c_1, \cdots, c_m \rangle$). The image of $s \in S_{p_1}$ (respectively, $s \in S_{p_2}$) is a set of states $\langle d_1, \cdots, d_n, y_1, \cdots, y_m \rangle$ (respectively, $\langle x_1, \cdots, x_n, c_1, \cdots, c_m \rangle$) in S_u such that for all possible valuations of $\langle y_1, \cdots, y_m \rangle$ (respectively, $\langle x_1, \cdots, x_n \rangle$) the valuation $\langle d_1, \cdots, d_n \rangle$ (respectively, $\langle c_1, \cdots, c_m \rangle$) remains the same. In other words, the image of s is a state predicate in S_u. The image of a state predicate $X \subseteq S_{p_1}$ (respectively, $X \subseteq S_{p_2}$) is also a state predicate in S_u, denoted $X \uparrow_{S_u}$, that is the union of the images of all states s in X. The image of a transition $(s_0, s_1) \in \delta_1$ (respectively, $(s_0, s_1) \in \delta_2$) is a set of transitions in $S_u \times S_u$ from $s_0 \uparrow_{S_u}$ to $s_1 \uparrow_{S_u}$ such that no variable y_j $(1 \leq j \leq m)$ (respectively, x_i $(1 \leq i \leq n)$) is changed by transitions of that set. We represent the image of a transition $(s_0, s_1) \in \delta_1$ (respectively, $(s_0, s_1) \in \delta_2$) by $(s_0, s_1) \uparrow_{S_u \times S_u}$. The image of a transition $(s_0, s_1) \in \delta_1$ (respectively, $(s_0, s_1) \in \delta_2$) is a set of transitions in $S_u \times S_u$. Let $\delta_1 \uparrow_{S_u \times S_u}$ denote the set of transitions in $S_u \times S_u$ that is the image of the set of transitions δ_1. The set of transitions of the union program (i.e., δ_u) is equal to $(\delta_1 \uparrow_{S_u \times S_u}) \cup (\delta_2 \uparrow_{S_u \times S_u})$. If $\mathcal{X} = \mathcal{Y}$ (i.e., $S_{p_1} = S_{p_2}$), then the union program is equal to $\langle \mathcal{X}, \delta_1 \cup \delta_2 \rangle$.

Properties and specifications. A *property* is a set of

infinite sequences of states that is *suffix closed* and *fusion closed*. *Suffix closure* of the set means that if a state sequence σ is in that set, then so are all the suffixes of σ. *Fusion closure* of the set means that if state sequences $\langle \alpha, s, \gamma \rangle$ and $\langle \beta, s, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta \rangle$ and $\langle \beta, s, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state. Following Alpern and Schneider [4], we rewrite a property as the intersection of a *safety property* and a *liveness property*. Intuitively, a safety property states that nothing bad ever happens. A liveness property specifies that something good will eventually occur. For a suffix closed and fusion closed property, a safety property can be specified as a set of bad transitions (see Page 26, Lemma 3.6 of [21]), that is, for program p, a safety property is a subset of $S_p \times S_p$. A *safety specification* of a program p is a conjunction of all safety properties of p. Hence, for simplicity, in this paper, we assume that a safety specification is represented as a set of transitions $\mathcal{B} \subseteq S_p \times S_p$ that must not occur in any program computation. Also, we say a transition (s_0, s_1) *violates* the safety specification iff $(s_0, s_1) \in \mathcal{B}$. A *computation satisfies the safety specification* iff none of its transitions violates the safety specification. A liveness property is represented by a set of *infinite* sequences of states, denoted $\mathcal{L}$. A computation σ *satisfies a liveness* property $\mathcal{L}$ iff $\sigma \in \mathcal{L}$. The *liveness specification* of a program is the conjunction of all its liveness properties. An example of a liveness property is a *leads-to* property, denoted $P \mapsto Q$, where P and Q are state predicates. A computation $\sigma = \langle s_0, s_1, \cdots \rangle$ satisfies $P \mapsto Q$ iff for every state s_i, if P holds in s_i, then there exists a state s_j, for $j \geq i$, in which Q holds.

A computation $\sigma = \langle s_0, s_1, ... \rangle$ *satisfies* a specification $spec$ iff σ satisfies safety and liveness of $spec$. We say a state predicate I_p is an *invariant* of a program p iff the following conditions are satisfied: (1) I_p is closed in δ_p, and (2) starting from every state in I_p, every computation of p satisfies $spec$.

Faults. We follow previous work [10, 5, 22, 24] in using the notion of state perturbation for modeling different types of faults as state perturbation provides a sufficiently expressive means to represent a variety of fault-types (e.g., Byzantine, failstop, message loss, etc.). Formally, a type of *faults* f for a program $p = \langle \mathcal{V}_p, \delta_p \rangle$ is a subset of the set $\{(s_0, s_1) : (s_0, s_1) \in S_p \times S_p\}$. We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in p and the transitions in f. We say that a state predicate $\mathcal{FS}_p$ is an f-span (read as *fault-span*) of p from the invariant I_p iff the following two conditions are satisfied: (1) $I_p \subseteq \mathcal{FS}_p$, and (2) $\mathcal{FS}_p$ is closed in $p[]f$. Observe that for all computations of p that start in states where I_p is true, $\mathcal{FS}_p$ is a boundary in the state space of p to which (but not beyond which) the state of p may be perturbed by the occurrence of the transitions in $p[]f$. A subset of $\mathcal{FS}_p - I_p$ that is reachable from I_p by a sequence of f transitions *alone* is called the *sub* f-span of p from I_p, denoted $\mathcal{SFS}_p$.

We say that a sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$, is a *computation of p in the presence of f* iff the following three conditions are satisfied: (1) if σ is infinite then $\forall j : 0 < j : (s_{j-1}, s_j) \in (\delta_p \cup f)$, (2) if σ is finite and terminates in state s_f then there does not exist state s, $s \neq s_f$, such that $(s_f, s) \in \delta_p$, and (3) $\exists n : n \geq 0 : (\forall j : n < j : (s_{j-1}, s_j) \in \delta_p)$. The first requirement captures that in each step, either a program transition or a fault transition is executed. The second requirement captures that faults do not have to execute. Finally, the third requirement captures that the number of fault occurrences in a computation is finite. This requirement is the same as that made in previous work (e.g., [10, 5, 27]) to ensure that eventually recovery can occur.

Fault-tolerance. We represent three levels of fault-tolerance (from [5, 21]) depending on the extent to which a program satisfies its safety or liveness specifications in the presence of faults. Intuitively, a *failsafe* fault-tolerant program guarantees to satisfy its safety specification even when faults occur. Formally, we say that p is *failsafe f-tolerant (*read as *fault-tolerant) from its invariant I_p for its specification $spec$* iff the following conditions hold: (1) in the absence of faults, p satisfies $spec$ from I_p, and (2) there exists $\mathcal{FS}_p$ such that (a) $\mathcal{FS}_p$ is an f-span of p from I_p, and (b) every computation of $p[]f$ satisfies the safety of $spec$ from $\mathcal{FS}_p$.

A *nonmasking* fault-tolerant program guarantees to recover to its invariant after faults stop occurring, however, its safety specification may be violated during recovery. We say that a program p is *nonmasking f-tolerant from I_p for $spec$* iff the following conditions hold: (1) in the absence of faults, p satisfies $spec$ from I_p, and (2) there exists $\mathcal{FS}_p$ such that (a) $\mathcal{FS}_p$ is an f-span of p from I_p, and (b) every computation of $p[]f$ that starts from a state in $\mathcal{FS}_p$ contains a state of I_p.

A *masking* fault-tolerant program always satisfies its safety specification (even in the presence of faults), and eventually recovers to its invariant. More precisely, a program p is *masking f-tolerant from I_p for $spec$* iff the following conditions hold: (1) in the absence of faults, p satisfies $spec$ from I_p, and (2) there exists $\mathcal{FS}_p$ such that (a) $\mathcal{FS}_p$ is an f-span of p from I_p, (b) $p[]f$ satisfies the safety of $spec$ from $\mathcal{FS}_p$, and (c) every computation of $p[]f$ that starts from a state in $\mathcal{FS}_p$ contains a state of I_p.

3 Requirements of Run-Time Fault-Tolerance

In this section, we formalize the requirements of designing run-time fault-tolerance against *unanticipated* faults. Consider a running program $p_o = \langle \mathcal{V}_o, \delta_o \rangle$, with an invariant I_o and the specification $spec$, that is perturbed by

an unanticipated fault-type f. Let $\mathcal{FS}_o$ be the f-span of p_o from I_o. To meet the requirements of failsafe/masking fault-tolerance at run-time, computations of $p_o[]f$ must satisfy the safety of $spec$. However, in the presence of f, the safety of $spec$ may be violated in two ways. First, the computations of $p_o|I_o$ may reach states inside I_o from where transitions of f may directly violate safety. Clearly, with the lack of knowledge about the behavioral nature of f, it is difficult to identify such states while designing p_o. Second, even if transitions of f do not directly violate the safety of $spec$, the safety of $spec$ may be violated by the transitions of $p_o|(\mathcal{FS}_o - I_o)$ once the state of p_o is perturbed outside I_o. This is because, in the design of p_o, no guarantees are provided for computations of p_o that start outside I_o.

In order to meet the requirements of nonmasking/masking fault-tolerance at run-time, every computation of p_o that starts in a state in $(\mathcal{FS}_o - I_o)$ must reach a state in I_o. In other words, after f stops occurring, recovery from $(\mathcal{FS}_o - I_o)$ to I_o should be provided. To ensure recovery, we have to resolve two classes of problems: non-progress cycles and deadlocks. More specifically, the computations of $p_o|(\mathcal{FS}_o - I_o)$ may reach cycles that prevent recovery. Likewise, if the computations of $p_o|(\mathcal{FS}_o - I_o)$ reach a state with no outgoing transitions (i.e., a deadlocked state), then the recovery to I_o cannot be achieved. It is obvious that without knowing the behavior of the fault-type f while designing p_o, it is difficult to bring about provisions that guarantee recovery from $(\mathcal{FS}_o - I_o)$.

In order to deal with the complexity of designing run-time fault-tolerance in the presence of unanticipated faults, we propose a hybrid design method by combining two existing approaches: dynamic program updates [18, 14, 15, 26, 2] and our previous work [13, 24] on offline synthesis of fault-tolerant programs from their fault-intolerant version. Specifically, since it is difficult (if not impossible) to guarantee that p_o meets the requirements of (failsafe/nonmasking/masking) fault-tolerance against *unanticipated* faults f, we make the following design decision to support run-time fault-tolerance:

> *After the fault-type f stops occurring, p_o will eventually be replaced with its (failsafe/nonmasking/masking) fault-tolerant version at run-time.*

In order to realize the above design decision, we use our offline synthesis algorithms to automatically generate a (failsafe/nonmasking/masking) fault-tolerant version of p_o , denoted p_n. A dynamic update of p_o with p_n due to the occurrence of faults is called a *fault-triggered update* in the state space S_u of the union of p_o and p_n.

In order to guarantee that a fault-triggered update eventually occurs, we first ensure that, once faults f stop occurring, the execution of p_o does not stay in a non-progress cycles in $((\mathcal{FS}_o - I_o) \uparrow S_u)$. One way to meet this requirement is to add monitors (detectors) to p_o that verify whether or not I_o has been violated at run-time. Such an addition of monitors to p_o can be done while designing p_o using existing approaches in the literature [9] that enable such a run-time monitoring. This is feasible because I_o is known at design time. After the addition of such monitors, the program p_o can stop executing once its state is perturbed outside I_o. As a result, only $\mathcal{SFS}_o$ is reachable from I_o by fault transitions, where $\mathcal{SFS}_o$ is the sub f-span of p_o from I_o.

Second, we design a *fault-triggered update program* (briefly called the *update program*) to ensure that p_o will eventually be replaced by p_n after f stops occurring. A fault-triggered update program $p = \langle \mathcal{V}_o \cup \mathcal{V}_n, \delta \rangle$ is defined in S_u. The set of transitions of an update program should satisfy three properties, namely *safeness, progress* and *interference-freedom*. The safeness requires that, during update, the safety of $spec$ is satisfied by an update program. The *progress* requires that a state in the invariant of the fault-tolerant program (i.e., $(I_n \uparrow S_u)$) is eventually reached. Intuitively, the *interference-freedom* requires that, in the absence of faults, the execution of the update program does not violate either the specification or the invariant of the fault-intolerant (respectively, fault-tolerant) program.

In order to meet the progress requirement, we must ensure that a state in $(I_n \uparrow S_u)$ is eventually reached from any state in $\mathcal{SFS}_o \uparrow S_u$. This is in fact a *leads-to* requirement that an update program must fulfill. Formally, we require that any program designed for updating p_o with its fault-tolerant version p_n satisfies $(\mathcal{SFS}_o \uparrow S_u) \mapsto (I_n \uparrow S_u)$. Intuitively, starting from any state that is an image of $\mathcal{SFS}_o$ in S_u, the update program will eventually reach a state that is an image of I_n in S_u.

In order to ensure interference-freedom, the update program must not include transitions that execute inside $I_o \uparrow S_u$ or $I_n \uparrow S_u$ (i.e., $\{(s_0, s_1) | (s_0, s_1 \in S_u) \wedge (s_0 \in (I_o \uparrow S_u) \wedge s_1 \in (I_o \uparrow S_u)) \vee (s_0 \in (I_n \uparrow S_u) \wedge s_1 \in (I_n \uparrow S_u))\}$). Moreover, the update program must not violate the closure of $I_o \uparrow S_u$ or $I_n \uparrow S_u$; i.e., the update program must not include the set of transitions $\{(s_0, s_1) | (s_0, s_1 \in S_u) \wedge (s_0 \in (I_o \uparrow S_u) \wedge s_1 \notin (I_o \uparrow S_u)) \vee (s_0 \in (I_n \uparrow S_u) \wedge s_1 \notin (I_n \uparrow S_u))\}$. The conjunction of the above requirements results in excluding any transition that starts in either $I_o \uparrow S_u$ or $I_n \uparrow S_u$ (i.e., $\{(s_0, s_1) | (s_0, s_1 \in S_u) \wedge (s_0 \in (I_o \uparrow S_u)) \vee (s_0 \in (I_n \uparrow S_u))\}$). While ensuring the reachability of $I_n \uparrow S_u$, the update program must not reach the invariant of the old program (i.e., $I_o \uparrow S_u$) because the execution of the update program would stay in $I_o \uparrow S_u$ due to the closure of $I_o \uparrow S_u$ in p_o, thereby violating the reachability of $I_n \uparrow S_u$. Therefore, we state the problem of designing fault-triggered update programs as follows:

The Problem of Designing Fault-Triggered Updates. Given are a fault-intolerant program $p_o = \langle \mathcal{V}_o, \delta_o \rangle$, its invariant I_o, its specification $spec$, a fault-type f, a program $p_n = \langle \mathcal{V}_n, \delta_n \rangle$ with its invariant I_n that is a failsafe/nonmasking/masking f-tolerant version of p_o, and $\mathcal{SFS}_o$, which is the sub f-span of p_o from I_o.

Identify an update program $p = \langle \mathcal{V}_o \cup \mathcal{V}_n, \delta \rangle$ in the state space S_u of the union of p_o and p_n such that

(1) p satisfies the safety of $spec$,
(2) p satisfies $(\mathcal{SFS}_o \uparrow S_u) \mapsto (I_n \uparrow S_u)$, and
(3) $\delta \cap \{(s_0, s_1) | (s_0, s_1 \in S_u) \wedge ((s_0 \in I_o \uparrow S_u) \vee (s_1 \in I_o \uparrow S_u) \vee (s_0 \in I_n \uparrow S_u))\} = \emptyset$.

□

Soundness and completeness. An algorithm for solving the update problem is *sound* iff for any given input, its output meets the requirements of the update problem. An algorithm for the update problem is *complete* iff for any given input if there exists a fault-triggered update program that meets the requirements of the update problem, then the algorithm always finds a program with a non-empty set of transitions δ.

Comment on the problem statement. We consider cases where faults f perturb the state of p_o outside its invariant I_o; i.e., $\mathcal{SFS}_o \neq \emptyset$. There are cases in which faults f may directly violate the safety of $spec$ inside I_o without actually perturbing the state of p_o outside I_o. Detecting the occurrence of such faults is more difficult since the state of p_o remains inside its invariant. We are currently investigating the design of run-time fault-tolerance against such types of faults.

4 Designing Fault-Triggered Updates

In this section, we present a sound and complete algorithm to solve the update problem defined in the previous section. Our goal is to design a program p in the state space S_u of the union of p_o and p_n such that p meets the requirements of the update problem. We present the Update algorithm (see Figure 1) for designing fault-triggered updates.

In the Update algorithm, the first three lines construct the images of the sub f-span of p_o from I_o, the invariant of the fault-intolerant program and the invariant of the fault-tolerant program in S_u. Line 4 initializes a state predicate $\mathcal{R}_0$, a set of transitions δ and an integer variable i. The repeat-until loop in Lines 5-8 iteratively constructs the set of transitions of the update program. Specifically, we first identify the set of states $\mathcal{R}_1$ from where a single-step recovery transition to the invariant of the fault-tolerant program (i.e., $\mathcal{R}_0 = \mathcal{I}_n$) is feasible. Such transitions should not violate the safety of $spec$ and their source state s_0 must be outside $\mathcal{I}_o$ and $\mathcal{I}_n$. We include all these transitions in the set of transitions δ. In the next iteration, we identify the set of states $\mathcal{R}_2$ from where a single-step recovery to $\mathcal{R}_1$ can be added. Continuing thus, we reach a point where no more single-step recovery to $\mathcal{R}_i$ is possible. The termination of the algorithm is guaranteed by the fact that, in each iteration, a set of states is excluded. If the image of the sub f-span of p_o in S_u includes states that do not belong to the state predicate $\cup_{k=0}^{i} \mathcal{R}_k$, then it means there are states reachable by fault transitions from where $\mathcal{I}_n$ cannot be reached. Otherwise, the set of transitions in δ constructs the set of transitions of the update program.

Theorem 4.1 The Update algorithm is sound.

Proof. The requirements (1) and (3) of the update problem are satisfied by construction as the Update algorithm includes only transitions that meet those requirements. Consider the following loop invariant for the Lines 5 to 8 (see Figure 1): *$\mathcal{I}_n$ is reachable from every state of $\mathcal{R}_i$*. The loop invariant is clearly true before the first iteration. Steps 5 and 6 preserve the loop invariant as they do not modify $\mathcal{R}_i$. Step 7 satisfies the loop invariant by the construction of the set of states $\mathcal{R}_{i+1}$. At termination, the loop invariant holds since $\mathcal{R}_i = \emptyset$. Thus, $\mathcal{I}_n$ is reachable from every state of $\cup_{k=0}^{i} \mathcal{R}_k$. Therefore, the progress requirements is met since $\mathcal{SFS}$ is a subset of the union of all $\mathcal{R}_i$ predicates. □

Theorem 4.2 The Update algorithm is complete.

Proof. Let $p' = \langle \mathcal{V}_o \cup \mathcal{V}_n, \delta' \rangle$ be a program for updating a fault-intolerant program $p_o = \langle \mathcal{V}_o, \delta_o \rangle$ with $p_n = \langle \mathcal{V}_n, \delta_n \rangle$ and our algorithm fails to find p'. Since p' is an f-triggered update program that meets the requirements of the update problem, all transitions of δ' satisfy the safety of $spec$ (Requirement (1)). Also, no transition in δ' starts in $\mathcal{I}_o$ (respectively, $\mathcal{I}_n$) or terminates in $\mathcal{I}_o$ (Requirement (3)). Moreover, from every state in $\mathcal{SFS}$ (see Figure 1), the program p' should meet the progress requirement. Thus, an arbitrary computation prefix $c = \langle s_0, \cdots, s_k \rangle$ ($k \geq 1$) of p' that starts in a state in $\mathcal{SFS}$ must ensure reachability of a state in $\mathcal{I}_n$; c does not include any repeated and deadlock states (i.e., all s_i states, for $0 \leq i \leq k$, are distinct and have some outgoing transition(s)). Let $s_0 \in \mathcal{SFS}$ and $s_k \in \mathcal{I}_n$. Since p' is a correct update program, the transition (s_{k-1}, s_k) must meet the requirements (1) and (3) of the update problem. Hence, $s_{k-1} \in \mathcal{R}_1$ must hold, i.e., our algorithm would include s_{k-1} in the state predicate $\mathcal{R}_1$ on Line 5. Likewise, the transition (s_{k-1}, s_k) would be included in δ on Line 6. A similar argument holds for s_{k-2} and the transition (s_{k-2}, s_{k-1}). Inductively, s_0 (respectively, (s_0, s_1)) would be included in $\mathcal{R}_k$ (respectively, δ). Since our algorithm declares failure if there exist states in $\mathcal{SFS}$ from where $\mathcal{I}_n$ is not reachable, it follows that our algorithm would have found the prefix c. Therefore, our algorithm is complete. □

Theorem 4.3 The complexity of the Update algorithm is polynomial in the state space of the union program.

Proof. Clearly, each one of the first four steps (see Figure 1) can be done in polynomial time in the state space of the union program, S_u. Moreover, Steps 5 and 6 inside the loop

Update($S_u, I_o, I_n, \mathcal{SFS}_o$: state predicates, $\mathcal{B}$: safety of *spec*)
{
$\mathcal{SFS} := \{s|(s \in S_u) \wedge (\exists s_0 : (s_0 \in \mathcal{SFS}_o) : (s \in s_0 \uparrow S_u))\}$; (1)
$\mathcal{I}_o := \{s|(s \in S_u) \wedge (\exists s_0 : (s_0 \in I_o) : (s \in s_0 \uparrow S_u))\}$; (2)
$\mathcal{I}_n := \{s|(s \in S_u) \wedge (\exists s_0 : (s_0 \in I_n) : (s \in s_0 \uparrow S_u))\}$; (3)
$\mathcal{R}_0 := \mathcal{I}_n$; $\delta := \emptyset$; $i := 0$; // $\mathcal{R}_i$ is a state predicate in S_u (4)
repeat{
$\mathcal{R}_{i+1} := \{s_0 \mid (s_0 \notin \mathcal{I}_o) \wedge (s_0 \notin (\cup_{k=0}^{i} \mathcal{R}_k)) \wedge (\exists s_1 : s_1 \in \mathcal{R}_i : (s_0, s_1) \notin \mathcal{B})\}$; (5)
$\delta := \delta \cup \{(s_0, s_1) : s_0 \in \mathcal{R}_{i+1} \wedge s_1 \in \mathcal{R}_i \wedge s_1 \notin \mathcal{I}_o \wedge (s_0, s_1) \notin \mathcal{B}\}$; (6)
$i := i + 1$; (7)
} until ($\mathcal{R}_i = \emptyset$); (8)
if ($\mathcal{SFS} \subseteq (\cup_{k=0}^{i} \mathcal{R}_k)$) then return δ; (9)
else declare that a fault-triggered update program does not exist;
return $\emptyset$; (10)
}

Figure 1. Automatic design of fault-triggered updates.

can also be computed in polynomial time. The rest of the algorithm consists of polynomial-time steps as well. Therefore, the complexity of the Update algorithm is polynomial in the state space of the union program. □

Comment on the scalability of our algorithm. While the complexity of synthesizing fault-triggered dynamic updates is polynomial in S_u, the state space of the union program has an exponential size with respect to the number of the variables of p_o and p_n. This may limit the applicability of our algorithm for real-world programs. We argue that such an algorithm has the potential to provide insight for developers in generating strategies that should be taken for the implementation of fault-triggered dynamic updates (respectively, adaptation). Moreover, our algorithm can be used for model-driven development of fault-triggered dynamic updates (respectively, adaptation) as the state space of the abstract model of a program is significantly smaller than the state space of the concrete program.

5 Parking Lot Example

In order to illustrate the Update algorithm, we use the parking lot problem (see Figure 2) adapted from [20]. The parking lot contains three spots for cars (marked a, b and c). There are two gates to enter the parking lot. Immediately after each gate, there is a space where the customer can drop the car off (marked l and r). If the gate is open, the customer may leave the car in this spot. The car will then be moved to one of the spots (a, b or c). We assume that if the car is being moved from l or r to a, b or c, no car can enter during this move. At the exit, there is one door. A car parked in spots a, b or c can leave through this door. However, only one car can leave the parking lot at a time. All doors are one-way, i.e., cars in spots l and r cannot leave and a car cannot enter in spots a, b or c.

Thus, in the parking lot problem, there are three possible events: (1) a car could be dropped off at the left gate and

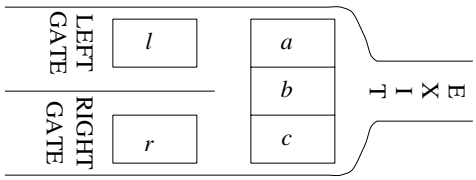

Figure 2. Parking Lot example.

(possibly) moved to the spots a, b or c, (2) a car could be dropped off at the right gate and (possibly) moved to the spots a, b or c, or (3) a car could exit. For simplicity, we assume that each event is atomic. Also, we assume that in each spot there can be at most one car. Now, we describe the state space of the fault-intolerant program, its invariant, its safety specification, and its transitions.

Variables and the state space of the fault-intolerant program (p_o). To model the parking lot, we maintain three variables; x, y and z. The variable x denotes the number of cars that can be let in through the left gate, y denotes the number of cars that can be let in through the right gate, and z denotes the number of cars in the lot. Hence, the left (respectively, right) gate is open when x (respectively, y) is positive. The domain of x and y is $\{0, 1, 2, 3\}$. The domain of z is $\{0, 1, 2, 3, 4, 5\}$. A state of p_o is obtained by assigning each variable a value from its domain. Thus, the state space of p_o contains 4x4x6 (=96) states.

Transitions (δ_o). For brevity, we write program transitions in terms of Dijkstra's guarded commands [11]. A guarded command (also called an *action*) is of the form $grd \longrightarrow st$, where grd is a state predicate and st is a statement that updates a subset of the program variables. The guarded command $grd \longrightarrow st$ corresponds to the set of transitions $\{(s_0, s_1) : grd$ is true in state s_0 and s_1 is obtained by *atomic* execution of st in state $s_0\}$. The fault-intolerant program p_o contains the following four actions representing the set of transitions δ_o:

$$
\begin{array}{lllll}
A_1 :: & x{>}0 \ \wedge \ z{<}5 & \longrightarrow & x, z := x{-}1, z{+}1 \\
A_2 :: & y{>}0 \ \wedge \ z{<}5 & \longrightarrow & y, z := y{-}1, z{+}1 \\
A_3 :: & y{<}3 \ \wedge \ z{>}0 & \longrightarrow & y, z := y{+}1, z{-}1 \\
A_4 :: & x{<}3 \ \wedge \ z{>}0 & \longrightarrow & x, z := x{+}1, z{-}1
\end{array}
$$

The first two actions let a car enter, and the last two actions let a car exit. Upon exit, the value x (or y) is nondeterministically increased so that a new car can enter. For simplicity, we do not model the actions corresponding to the movement of the car inside the parking lot.

Invariant (I_o). We let the invariant of the fault-intolerant program, I_o, be $x{+}y{+}z \leq 3$.

Safety specification. The safety specification $\mathcal{B}_o$ requires that any car in the parking lot should be able to leave. Clearly, a car in spots a, b or c can leave. However, if spots a, b and c are occupied and there is a car in spot l (respectively, r) then that car cannot leave. Hence, it is required that if there exist three or more cars in the lot, then the program p_o must not increase the number of cars. Also, to model the assumption that only one event (entry/exit) can occur at a time, it is required that the value of x, y and z can change at most by 1. Thus, the safety specification rules out the following transitions:

$$
\begin{array}{l}
\mathcal{B}_o = \{(s_0, s_1) : ((s_0, s_1) \in \delta_o) \wedge \\
\quad ((z(s_0) > 3 \ \wedge \ z(s_1) \geq z(s_0)) \ \vee \\
\quad |x(s_1) - x(s_0)|{>}1 \ \vee \ |y(s_1) - y(s_0)|{>}1 \vee \\
\qquad\qquad |z(s_1) - z(s_0)|{>}1)\}
\end{array}
$$

Notation. $v(s)$ represents the value of a variable v in state s.

Faults. The fault-intolerant program is subject to a fault-type F that allows a car to sneak in. Intuitively, by increasing the value of z, the occurrence of F may prevent the entrance of the cars while there is some room in the parking lot. We model the fault-type F by the following action:

$$
F :: \quad z{<}5 \qquad \longrightarrow \qquad z := z{+}1
$$

When F occurs, the program p_o may reach states in its F-span where any increase of z *by program actions* would violate the safety specification. In this example, we use automated techniques [13, 24] to design a *failsafe* fault-tolerant version of p_o in an offline fashion and then use the Update algorithm presented in Section 4 in order to design an update program for dynamically updating p_o with its fault-tolerant version p_n once F occurs. The failsafe F-tolerant version of p_o is as follows:

$$
\begin{array}{lll}
A'_1 :: \ (x'{>}0) \wedge (z'{<}4) \wedge (\mathcal{I}_n) & \longrightarrow & x', z' := x'{-}1, z'{+}1 \\
A'_2 :: \ (y'{>}0) \wedge (z'{<}4) \wedge (\mathcal{I}_n) & \longrightarrow & y', z' := y'{-}1, z'{+}1 \\
A'_3 :: \ (y'{<}3) \wedge (z'{>}0) & \longrightarrow & y', z' := y'{+}1, z'{-}1 \\
A'_4 :: \ (x'{<}3) \wedge (z'{>}0) & \longrightarrow & x', z' := x'{+}1, z'{-}1
\end{array}
$$

The set of variables of the fault-tolerant program p_n is disjoint from the set of variables of the fault-intolerant program p_o. The safety specification and the invariant of p_n have the same semantics as $\mathcal{B}_o$ and I_o except that each variable is replaced with its primed version. Once p_o is dynamically updated with p_n faults may perturb the value of z' as both z and z' represent the value of an input sensor to the gate controller. However, since p_n does not increase the value of z' if there are more than three cars in the lot, p_n does not violate the safety specification if faults perturb the state of p_n outside its invariant or $z' > 3$.

Designing the F-triggered update program. We use the Update algorithm to design an update program p in S_u. Each state of the union program $p_u = \langle \{x, y, z, x', y', z'\}, \delta_u \rangle$ has the form $\langle x, y, z, x', y', z' \rangle$. The state space of the union program, S_u, has 9216 states and the set of transitions δ_u is represented by all actions $A_1{-}A_4$ and $A'_1{-}A'_4$. The set of variables of the *update* program p is the same as the variables of the union program, however, its set of transitions is automatically generated by the Update algorithm. The transitions of the synthesized update program are as follows:

$$
\begin{array}{l}
U_1 :: \neg\mathcal{I}_o \wedge \neg\mathcal{I}_n \wedge ((4 \leq x' + y' + z' < 6) \vee \\
\quad ((x' + y' + z' = 6) \wedge (x' \neq 0 \wedge y' \neq 0 \wedge z' \neq 0))) \\
\qquad\qquad \longrightarrow \text{if } (x' > 1) \text{ then } x' := x' - 1; \\
\qquad\qquad\quad \text{if } (y' > 1) \text{ then } y' := y' - 1; \\
\qquad\qquad\quad \text{if } (z' \neq 0) \text{ then } z' := z' - 1; \\
U_2 :: \neg\mathcal{I}_o \wedge \neg\mathcal{I}_n \wedge ((6 < x' + y' + z' \leq 9) \vee \\
\quad ((x' + y' + z' = 6) \wedge (x' = 0 \vee y' = 0 \vee z' = 0))) \\
\qquad\qquad \longrightarrow \text{if } (x' > 1) \text{ then } x' := x' - 1; \\
\qquad\qquad\quad \text{if } (y' > 1) \text{ then } y' := y' - 1; \\
\qquad\qquad\quad \text{if } (z' \neq 0) \text{ then } z' := z' - 1;
\end{array}
$$

The action U_1 represents the set of transitions that recover to $\mathcal{I}_n$ while changing x, y and z at most one unit (i.e., preserving the safety of *spec*). The guard of the action U_1 specifies the set of states $\mathcal{R}_1$ from where a single-step recovery to $\mathcal{I}_n$ exist without violating the safety specification. Likewise, the guard of the action U_2 defines the set of states $\mathcal{R}_2$ from where a single-step recovery to $\mathcal{R}_1$ exist. To gain more assurance, we have verified the parking lot example using the Spin model checker [17]. The model comprises all actions $A_1 - A_4$, $A'_1 - A'_4$, and U_1 and U_2 for which safeness, progress and interference-freedom have been verified. The Promela model is available at `http://www.cs.mtu.edu/~aebnenas/research/examples/pklt-rtft.txt`.

6 Defining Fault-Tolerant Updates

The problem of designing fault-triggered updates (see Section 3) does not stipulate any requirements on the computations of the update program in the presence of faults. Specifically, an update program generated by the Update algorithm guarantees progress (i.e., update completes) after faults stop perturbing p_o and provides no guarantees

if faults occur during the update. It is often the case that the same type of faults that triggers the update may also occur during the update, thereby requiring the update program to tolerate the faults. Next, we define three levels of fault-tolerant updates based on the extent to which safety and progress are satisfied. For the following definitions, let $p = \langle \mathcal{V}_o \cup \mathcal{V}_n, \delta \rangle$ be an f-triggered update program (for faults f) with the state space S_u. The program p enables the run-time replacement of a program $p_o = \langle \mathcal{V}_o, \delta_o \rangle$ with a program $p_n = \langle \mathcal{V}_n, \delta_n \rangle$ if f occurs while p_o is executing.

Failsafe Fault-Tolerant Update. Intuitively, a failsafe fault-tolerant update guarantees to preserve safeness even if progress is violated due to the occurrence of faults during the update. Formally, p is a *failsafe f-tolerant update* (i.e., failsafe fault-tolerant update against f) iff the following conditions hold: (1) in the absence of f, any computation of p starting in the image of the sub f-span of p_o (i.e., $\mathcal{SFS}_o \uparrow S_u$) satisfies both its progress and safeness properties, and (2) in the presence of faults, any computation of $p[]f$ satisfies its safeness.

Nonmasking Fault-Tolerant Update. A nonmasking fault-tolerant update guarantees progress even if faults occur during the update, however, safeness may be violated. In formal terms, an update program p is a *nonmasking f-tolerant update* (i.e., nonmasking fault-tolerant update against f) iff the following conditions hold: (1) in the absence of f, any computation of p starting in the image of the sub f-span of p_o (i.e., $\mathcal{SFS}_o \uparrow S_u$) satisfies both its progress and safeness properties, and (2) in the presence of faults, any computation of $p[]f$ satisfies its progress; i.e., any computation of $p[]f$ has a state in $I_n \uparrow S_u$.

Masking Fault-Tolerant Update. A masking fault-tolerant update guarantees progress even if faults occur during the update, and satisfies safeness at all times (even if faults occur). More precisely, a program p is a *masking f-tolerant update* (i.e., masking fault-tolerant update against f) iff the following conditions hold: (1) in the absence of f, any computation of p starting in the image of the sub f-span of p_o (i.e., $\mathcal{SFS}_o \uparrow S_u$) satisfies both its progress and safeness properties, and (2) in the presence of f, any computation of $p[]f$ satisfies both progress and safeness properties.

7 Related Work

Several approaches on dynamic program updates, adaptation and run-time verification have inspired the proposed work in this paper. For example, Gupta *et al.* [15] propose a formal framework in which developers define a transfer function that specifies how a state of the new program is reached instantaneously. Kramer and Magee [18] enable run-time replacement of a program (or a component thereof) with its new version where the update may be triggered once the running program reaches a set of *quiescent* states; i.e., states in which the program is in a passive status where no external entity is using or communicating with the running program. In the context of our work, since faults may trigger the update from arbitrary states, we are not faced with the problem of identifying the set of quiescent states. However, the occurrence of faults introduces new problems such as reachability of states from where a *safe* dynamic update is not possible. Moreover, faults may occur even during the update, which poses a new challenge for designing dynamic updates that are fault-tolerant themselves. While we have defined three classes of fault-tolerant updates (see Section 6), the design of such updates is currently under investigation.

The proposed approach in this paper also intersects with techniques for developing adaptive systems [19, 3, 6, 8, 23] most of which put less emphasis on tolerating faults during adaptation. For instance, Kramer and Magee [19] use process algebra to specify and verify (behavioral and architectural) adaptation. Allen *et al.* [3] separate functional concerns from adaptation by encapsulating dynamic updates in connectors used to arbitrate communication between system components. Chen *et al.* [8] present a graceful adaptation protocol (transparent to application layer) that is used to construct adaptive distributed software. Zhang and Cheng [28] present a systematic model-based approach for specifying adaptation based on the notion of transferring the state of a program before adaptation to the state of a program after adaptation. They use model checking to verify invariant properties that should hold during adaptation. While all aforementioned approaches play an important role in specification, modeling and design of adaptive systems, the proposed approach in this paper focuses on automated design of (fault-tolerant) adaptations triggered by the occurrence of faults.

Run-time monitoring and verification approaches [25, 7, 16] present techniques and programming artifacts for composing monitors with programs to verify program behaviors at run-time without emphasis on run-time updates. Such techniques enable us to *detect* behavioral deviations at run-time without emphasizing on how to remedy the errors due to the occurrence of faults while maintaining some level of service availability. By contrast, our work focuses on how to design mechanisms that enable the replacement of a running program with its fault-tolerant version (that is designed in an offline fashion) while guaranteeing that safety requirements are satisfied during such dynamic updates. As a result, if the same type of faults occurs after the update is complete, then the updated program possesses the necessary means to tolerate the faults.

8 Conclusions and Future Work

We presented a systematic approach for the design of run-time fault-tolerance. The proposed approach extends (1) our previous results [13, 24] in offline synthesis of fault-tolerance, and (2) existing techniques for dynamic program updates [18, 14, 15, 26, 2]. Specifically, we presented a

sound and complete algorithm that takes a fault-intolerant program and its fault-tolerant version, and automatically generates a *fault-triggered update program*. The update program ensures that, once faults occur, the fault-intolerant program is upgraded to its fault-tolerant version at run-time. The completeness of our algorithm is especially useful in that it provides an *existence test* for the design of fault-triggered dynamic updates and programs that adapt in the presence of faults. We also defined three classes of fault-tolerant dynamic updates (respectively, fault-tolerant adaptations) depending on the extent to which system specification is satisfied during update.

As we pursue our ongoing work on the design of fault-triggered and fault-tolerant updates, we plan to implement a software tool for automated design of fault-triggered dynamic updates that tolerate faults. Such a software tool will be integrated in a UML-based framework for model-driven development of *correct adaptation in the presence of faults*. Specifically, given the state diagram of an old program and the state diagram of a new program, we automatically generate the state diagram of a program that would enable the adaptation of the old to the new program in the presence of faults. We are also interested in designing techniques that refine the update programs generated by the Update algorithm (presented in this paper) to support fault-triggered updates under distribution constraints.

References

[1] S. Ajmani. *Automatic Software Upgrades for Distributed Systems*. Ph.D., MIT, Sept. 2004. Also as Technical Report MIT-LCS-TR-1012.

[2] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 452–476, July 2006.

[3] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *FASE*, pages 21–37, 1998.

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

[5] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[6] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *WICSA*, pages 107–126, 1999.

[7] F. Chen and G. Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic. Notes on Theoretical Computer Science*, 89(2), 2003.

[8] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing adaptive software in distributed systems. In *ICDCS*, pages 635–643, 2001.

[9] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.

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

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1990.

[12] D. Duggan. Type-based hot swapping of running modules (extended abstract). In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 62–73, 2001.

[13] A. Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University, 2005.

[14] D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Software Practice and Experience*, 23(9):949–964, Sept. 1993.

[15] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.

[16] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[17] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[18] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[19] J. Kramer and J. Magee. Analysing dynamic change in distributed software architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.

[20] S. Kulkarni, A. Arora, and A. Ebnenasir. *Adding Fault-Tolerance to State Machine-Based Designs*. Software Engineering and Knowledge Engineering. World Scientific Publishing Co. Pte. Ltd, 2007.

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

[22] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *In Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, 2000.

[23] S. S. Kulkarni and K. N. Biyani. Correctness of component-based adaptation. In *CBSE*, pages 48–58, 2004.

[24] S. S. Kulkarni and A. Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(3):201–215, July-September 2005.

[25] J. Levy, H. Saidi, and T. E. Uribe. Combining monitors for run-time system verification. *Electronic Notes in Theoretical Computer Science*, 70(4), December 2002.

[26] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and flexible dynamic software updating. *In Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 183–194, January 2005.

[27] G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT/LCS/TR-583, 1993.

[28] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE*, pages 371–380, 2006.