Synthesis of Fault-Tolerant Concurrent Programs

PAUL C. ATTIE
Northeastern University and MIT Computer Science and Artificial Intelligence Laboratory
ANISH ARORA
The Ohio State University
and
E. ALLEN EMERSON
The University of Texas at Austin

Methods for mechanically synthesizing concurrent programs from temporal logic specifications obviate the need to manually construct a program and compose a proof of its correctness. A serious drawback of extant synthesis methods, however, is that they produce concurrent programs for models of computation that are often unrealistic. In particular, these methods assume completely fault-free operation, that is, the programs they produce are fault-intolerant. In this paper, we show how to mechanically synthesize fault-tolerant concurrent programs for various fault classes. We illustrate our method by synthesizing fault-tolerant solutions to the mutual exclusion and barrier synchronization problems.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]: Fault Tolerance; D.1.2 [**Programming Techniques**]: Automatic Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, formal methods, model checking, reliability*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs, mechanical verification, specification techniques*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Temporal logic*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*

An extended abstract containing some of the results of this paper was presented at the ACM Symposium on Principles of Distributed Computing, Puerto Vallarta, Mexico, 1998.
P. C. Attie was supported in part by the National Science Foundation under grant number CCR-0204432. A. Arora was supported in part by DARPA-NEST contract number F33615-01-C-1901, NSF grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, and an unrestricted grant from Microsoft Research. E. A. Emerson was supported in part by NSF grants CCR-009-8141 and ITR-CCR-020-5483, and SRC Contract No. 2002-TJ-1026.
Authors' addresses: P. C. Attie, College of Computer and Information Science, Northeastern University, Cullinane Hall, 360 Huntington Ave., Boston, MA 02115; email: attie@ccs.neu.edu; A. Arora, Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210; email: anish@cis.ohio-state.edu; E. A. Emerson, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: emerson@cs.utexas.edu.

General Terms: Design, Languages, Reliability, Theory, Verification

Additional Key Words and Phrases: Concurrent programs, fault-tolerance, program synthesis, specification, temporal logic

1. INTRODUCTION

Methods for synthesizing concurrent programs from temporal logic specifications based on the use of a decision procedure for testing temporal satisfiability have been proposed by Emerson and Clarke [1982] and Manna and Wolper [1984]. An important advantage of these synthesis methods is that they obviate the need to manually compose a program and manually construct a proof of its correctness. One only has to formulate a precise problem specification; the synthesis method then mechanically constructs a correct solution. A serious drawback of these methods, however, is that they deal only with functional correctness properties. Nonfunctional properties such as *fault-tolerance* are not addressed. For example, the method of Manna and Wolper [1984] produces CSP programs in which all communication takes place between a central synchronizer process and one of its satellite processes. Thus, failure of the central synchronizer blocks the entire system.

In this paper, we present a sound and complete method for the synthesis of fault-tolerant programs. In our method, the properties of the program in the absence of faults are described in a *problem specification*, and the fault-tolerance properties of the program are described in terms of the behavior of the program when subjected to the occurrence of faults [Arora and Kulkarni 1998]. The faults themselves are specified as a set of actions (guarded commands) that perturb the state of the program [Arora and Gouda 1993].

Our synthesis method is based on a decision procedure for the branching-time temporal logic CTL [Emerson and Clarke 1982]. We apply this decision procedure to synthesize both “normal” behavior in the absence of faults, and “recovery” behavior, after the occurrence of a fault. Soundness of our method means that both normal and recovery behavior conform to the given specification. Completeness means that if some fault-tolerant program exists which satisfies the specification, then our method produces such a program. A byproduct of completeness is the ability to mechanically generate *impossibility results*: if the method fails, then we can conclude that the specified problem has no solution, for example, because the required recovery is not attainable in the presence of the specified faults.

Our method has time complexity exponential in the size of the problem description. Roughly speaking, this is because our method generates a global-state transition diagram which contains exactly the behaviors of the program to be synthesized. We note that all extant synthesis methods (except those of Attie and Emerson [1998], Attie [1999], and Emerson et al. [1992]) rely on exhaustive state-space search and thus also have exponential (at least) time complexity. We outline in the conclusions a way of circumventing this exponential complexity by combining the method presented here with that of Attie and Emerson [1998] and Attie [1999].

We also show how our method accommodates *multitolerance* [Arora and Kulkarni 1998], in which different faults may have to be tolerated in different ways, that is, the required fault-tolerance properties depend not only on the problem specification, but also on the particular fault that occurs.

The paper is as follows. Section 2 defines the model of computation, specification language (CTL), fault-model, and fault-tolerance properties that we consider. Section 3 formally defines the synthesis problem for fault-tolerant concurrent programs. Section 4 reviews the CTL decision procedure [Emerson 1981; Emerson and Clarke 1982]. Section 5 presents our synthesis method. Section 6 illustrates our method by applying it to synthesize fault-tolerant solutions for the mutual exclusion and barrier synchronization problems, and also to automatically produce an impossibility result. Section 7 establishes the soundness, completeness, and complexity of the method. Section 8 discusses the method's scope, extends the method to deal with multitolerance, and outlines an alternative synthesis method that guarantees stronger correctness properties but has a narrower range of application. Section 9 discusses related work, proposes future research, and concludes the paper.

2. TECHNICAL PRELIMINARIES

We now present some technical preliminaries. First, we describe our model of concurrent computation, and the temporal logic CTL (mostly taken from Emerson and Clarke [1982]). We then give some of the concepts and technical details needed in order to model faults. We first give a general model of faults, and then define a special type of Kripke structure that incorporates the transitions that arise from the occurrence of a fault (which we call *fault-transitions*). We finally outline some of the fault-tolerance properties that our method can deal with. One of the contributions of this paper is the definition of a formal model of faults within the model-theoretic setting, which enables mechanical reasoning about programs, specifically, synthesis of a program from a specification (our topic in this paper) and model-checking a program against a specification (a topic we leave to another occasion, but certainly one that our framework can address).

2.1 Model of Concurrent Computation

We consider nonterminating concurrent programs of the form $P = P_1 \| \cdots \| P_I$ which consist of a finite number of fixed sequential processes $P_1, \ldots, P_I$ running in parallel. With every process P_i, we associate a single, unique index, namely i. Formally, each process P_i is a directed graph where each node is labeled by a unique name (s_i), and each arc is labeled with an action $B \rightarrow A$ consisting of an enabling condition (i.e., guard) B and corresponding statement A to be performed (i.e., a guarded command [Dijkstra 1976]). A *global state* is a tuple of the form $(s_1, \ldots, s_I, x_1, \ldots, x_m)$ where each node s_i is the current local state of P_i and $x_1, \ldots, x_m$ is a list (possibly empty) of shared synchronization variables. A guard B is a predicate on global states and a statement A is a parallel assignment which updates the values of the shared variables. If the guard B is omitted from an action, it is interpreted as *true* and we simply write the action as A. If the statement A is omitted, the shared variables are unaltered and we write the action as B.

We model concurrency in the usual way by the nondeterministic interleaving of the "atomic" transitions of the individual processes P_i. Hence, at each step of the computation, some process with an enabled action is nondeterministically selected to be executed next. Assume that $s = (s_1, \ldots, s_i, \ldots, s_I, x_1, \ldots, x_m)$ is the current global state, and that P_i contains an arc from node s_i to s'_i labeled by the action $B \to A$. If B is true in the current state then a transition can be made to the next state $s' = (s_1, \ldots, s'_i, \ldots, s_I, x'_1, \ldots, x'_m)$ where $x'_1, \ldots, x'_m$ is the list of updated shared variables resulting from execution of statement A (we notate this transition as $s \xrightarrow{i,A} s'$). A *computation* is any sequence of states where each successive pair of states is related by the above next state transition relation.

The synthesis task thus amounts to supplying the actions to label the arcs of each process so that the resulting computation tree of the entire program $P_1 \| \cdots \| P_I$ meets a given temporal logic specification.

2.2 The Specification Language CTL

We have the following syntax for CTL, where p denotes an atomic proposition, and f, g denote (sub-)formulae. The atomic propositions are drawn from a set $\mathcal{AP}$ that is partitioned into sets $\mathcal{AP}_1, \ldots, \mathcal{AP}_I$. $\mathcal{AP}_i$ contains the atomic propositions local to process i. Other processes can read propositions in $\mathcal{AP}_i$, but only process i can modify these propositions (which collectively define the local state of process i).

—Each of p, $f \wedge g$, and $\neg f$ is a formula (where $\wedge$, $\neg$ indicate conjunction and negation, respectively).

—$\mathsf{EX}_j f$ is a formula which means that there is an immediate successor state reachable by executing one step of process P_j in which formula f holds.

—$\mathsf{A}[f \mathsf{U} g]$ is a formula which means that for every computation path, there is some state along the path where g holds, and f holds at every state along the path until that state.

—$\mathsf{E}[f \mathsf{U} g]$ is a formula which means that for some computation path, there is some state along the path where g holds, and f holds at every state along the path until that state.

Formally, we define the semantics of CTL formulae with respect to a Kripke structure $M = (S_0, S, A, L)$ consisting of the following:

S, a countable set of global states.

$S_0 \subseteq S$, a nonempty set of initial states.

$A \subseteq S \times [1\!:\!I] \times S$. a transition relation. A is partitioned into relations $A_1, \ldots, A_I$, where A_i gives the transitions of process i.[1]

$L : S \mapsto 2^{\mathcal{AP}}$, a labeling function which labels each state with the set of atomic propositions true in that state.

[1]We use $[m : n]$ for the set of natural numbers m through n inclusive ($\emptyset$ if $m > n$), and $[m : n)$ for the set of natural numbers m through $n-1$ inclusive ($\emptyset$ if $m \geq n$).

A *path* is a sequence of states where each pair of successive states is related by the transition relation A. A *fullpath* is a maximal path, that is, a path that is either infinite, or ends in a state with no outgoing transitions. If π is a fullpath, then define $|\pi|$, the length of π, to be ω when π is infinite and k when π is finite and of the form $s^0 \rightarrow \cdots \rightarrow s^k$. We use the usual notation for truth in a structure: $M, s^0 \models f$ means that f is true at state s^0 in structure M. When the structure M is understood, we write $s^0 \models f$. We define $\models$ inductively:

$M, s^0 \models p$	iff	$p \in L(s^0)$ for atomic proposition p;
$M, s^0 \models \neg f$	iff	not($M, s^0 \models f$);
$M, s^0 \models f \wedge g$	iff	$M, s^0 \models f$ and $M, s^0 \models g$;
$M, s^0 \models \mathsf{EX}_j f$	iff	for some state t, $(s^0, t) \in A_j$ and $M, t \models f$;
$M, s^0 \models \mathsf{A}[f \mathsf{U} g]$	iff	for all fullpaths $\pi = (s^0, s^1, \ldots)$ in M that start in s^0, there exists $i \in [0 : |\pi|]$ such that $M, s^i \models g$ and for all $j \in [1 : (i-1)]$: $M, s^j \models f$;
$M, s^0 \models \mathsf{E}[f \mathsf{U} g]$	iff	for some fullpath $\pi = (s^0, s^1, \ldots)$ in M that starts in s^0, there exists $i \in [0 : |\pi|]$ such that $M, s^i \models g$ and for all $j \in [1 : (i-1)]$: $M, s^j \models f$.

We say that a formula f is *satisfiable* if and only if there exists a structure M and state s of M such that $M, s \models f$. In this case, we say that M is a *model* of f. We say that a formula f is *valid* if and only if $M, s \models f$ for all structures M and states s of M.

We use the notation $M, U \models f$ as an abbreviation of $\forall s \in U : M, s \models f$, where U is a set of global states. We introduce the abbreviations $f \vee g$ for $\neg(\neg f \wedge \neg g)$, $f \Rightarrow g$ for $\neg f \vee g$, $f \equiv g$ for $(f \Rightarrow g) \wedge (g \Rightarrow f)$, $\mathsf{AF} f$ for $\mathsf{A}[true \mathsf{U} f]$, $\mathsf{EF} f$ for $\mathsf{E}[true \mathsf{U} f]$, $\mathsf{A}[f \mathsf{W} g]$ for $\neg\mathsf{E}[\neg f \mathsf{U} \neg g]$, $\mathsf{E}[f \mathsf{W} g]$ for $\neg\mathsf{A}[\neg f \mathsf{U} \neg g]$, $\mathsf{AG} f$ for $\mathsf{A}[false \mathsf{W} f]$, $\mathsf{EG} f$ for $\mathsf{E}[false \mathsf{W} f]$, $\mathsf{AX}_i f$ for $\neg\mathsf{EX}_i \neg f$, $\mathsf{EX} f$ for $\mathsf{EX}_1 f \vee \cdots \vee \mathsf{EX}_k f$, and $\mathsf{AX} f$ for $\mathsf{AX}_1 f \wedge \cdots \wedge \mathsf{AX}_k f$. Note that $\mathsf{A}[g \mathsf{W} h] \equiv \mathsf{A}[h \mathsf{U_w} (g \wedge h)]$ and $\mathsf{E}[g \mathsf{W} h] \equiv \mathsf{E}[h \mathsf{U_w} (g \wedge h)]$, where $\mathsf{U_w}$ is the "weak until" modality: $\mathsf{A}[f' \mathsf{U_w} f'']$, $(\mathsf{E}[f' \mathsf{U_w} f''])$ mean that along all paths (along some path), either f' holds forever, or that f'' eventually holds, and f' holds at all states up to (but not necessarily including) the first state in which f'' holds. When omitting the subformulae, we will write $\mathsf{A}[f \mathsf{U} g]$, $\mathsf{E}[f \mathsf{U} g]$, $\mathsf{A}[f \mathsf{W} g]$, $\mathsf{E}[f \mathsf{W} g]$ as AU, EU, AW, EW, respectively. In CTL, every occurrence of a *path quantifier* (A or E) is paired with one of the *linear time modalities* X_j, F, G, U, W. We call such a pair a *CTL modality*.

A formula of the form $\mathsf{A}[f \mathsf{U} g]$ or $\mathsf{E}[f \mathsf{U} g]$ is an *eventuality* formula. An eventuality corresponds to a liveness property in that it makes a promise that something does happen. This promise must be *fulfilled*. The eventuality $\mathsf{A}[f \mathsf{U} g]$ $(\mathsf{E}[f \mathsf{U} g])$ is fulfilled for s in M provided that for every (respectively, for some) path starting at s, there exists a finite prefix of the path in M whose last state satisfies g and all of whose other states satisfy f. For all the states of this finite prefix except the last, we say that the eventuality is *pending*, since g does not hold in these states (otherwise a shorter prefix could have been used). Since $\mathsf{AF} g$ and $\mathsf{EF} g$ are special cases of $\mathsf{A}[f \mathsf{U} g]$ and $\mathsf{E}[f \mathsf{U} g]$, respectively, they are also eventualities. In contrast, $\mathsf{A}[f \mathsf{W} g]$, $\mathsf{E}[f \mathsf{W} g]$ (and their special cases $\mathsf{AG} f$ and $\mathsf{EG} f$) are *invariance* formulae. An invariance corresponds to a safety property since it asserts that whatever happens to occur (if anything) will meet certain conditions.

Since our programs are in general finite state, the propositional version of temporal logic can be used to specify their properties. This is essential, since only propositional temporal logics enjoy the finite-model property, which is the underlying basis of the CTL decision procedure of Emerson and Clarke [1982] that this paper builds upon.

An example CTL specification is that for the two-process mutual exclusion problem (here $i \in \{1, 2\}$):

(1) Initial state (both processes are initially in their noncritical region): $N_1 \wedge N_2$.

(2) It is always the case that any move P_i makes from its noncritical region is into its trying region and such a move is always possible:
$$\mathsf{AG}(N_i \Rightarrow (\mathsf{AX}_i T_i \wedge \mathsf{EX}_i T_i)).$$

(3) It is always the case that any move P_i makes from its trying region is into its critical region:
$$\mathsf{AG}(T_i \Rightarrow \mathsf{AX}_i C_i).$$

(4) It is always the case that any move P_i makes from its critical region is into its noncritical region and such a move is always possible:
$$\mathsf{AG}(C_i \Rightarrow (\mathsf{AX}_i N_i \wedge \mathsf{EX}_i N_i)).$$

(5) P_i is in at most one of N_i, T_i, or C_i:
$$\mathsf{AG}(N_i \Rightarrow \neg(T_i \vee C_i)) \wedge \mathsf{AG}(T_i \Rightarrow \neg(N_i \vee C_i)) \wedge \mathsf{AG}(C_i \Rightarrow \neg(N_i \vee T_i)).$$

(6) A transition by one process cannot cause a transition by another (interleaving model of concurrency):
$$\mathsf{AG}((N_1 \Rightarrow \mathsf{AX}_2 N_1) \wedge (N_2 \Rightarrow \mathsf{AX}_1 N_2)),$$
$$\mathsf{AG}((T_1 \Rightarrow \mathsf{AX}_2 T_1) \wedge (T_2 \Rightarrow \mathsf{AX}_1 T_2)),$$
$$\mathsf{AG}((C_1 \Rightarrow \mathsf{AX}_2 C_1) \wedge (C_2 \Rightarrow \mathsf{AX}_1 C_2)).$$

(7) P_i does not starve: $\mathsf{AG}(T_i \Rightarrow \mathsf{AF} C_i)$.

(8) P_1, P_2 do not access critical resources together: $\mathsf{AG}(\neg(C_1 \wedge C_2))$.

(9) It is always the case that some process can move: $\mathsf{AGEX}\mathit{true}$.

We call the specification that expresses the required properties of the program in the absence of faults the *problem specification*. We assume, in the sequel, that the problem specification is expressed in the form $\mathit{init\text{–}spec} \wedge \mathsf{AG}(\mathit{global\text{–}spec})$, where *init–spec* contains only atomic propositions and Boolean operators. *init–spec* specifies the initial state, and *global–spec* specifies correctness properties that are required to hold at all states that are reachable from an initial state in the absence of faults. We call these the *initial specification* and *global specification*, respectively.

For the mutual exclusion specification above, *init–spec* is clause 1, and *global–spec* is the conjunction of Clauses 2–9.

2.3 Model of Faults

The faults that a concurrent program is subject to may be categorized in a variety of ways:

(1) *Type*, for example, the faults are stuck-at, fail-stop, crash, omission, timing, performance, or Byzantine.
(2) *Duration*, for example, the faults are permanent, intermittent, or transient.
(3) *Observability*, for example, the faults are detectable or not by the program.
(4) *Repair*, for example, the faults are correctable or not by the program.

Toward developing a uniform and general method for fault-tolerant concurrent program synthesis that accommodates these various categories of faults, we recall a uniform and general representation of faults (cf. Arora and Gouda [1993]). In this representation, faults are modeled as actions (guarded commands) whose execution perturbs the program state. Consider for example a fault that corrupts the state of a wire. The wire itself is represented by the following program action over two one-bit variables in and out: $out \neq in \rightarrow out := in$. The fault that corrupts the state of the wire is represented by the fault action: $out \neq in \rightarrow out :=\ ?$, where ? denotes a nondeterministically chosen binary value.

For this representation to capture all of the categories mentioned above sometimes requires the use of auxiliary state variables. For example, consider the fault by which the wire is stuck-at-low-voltage. In this case, the correct behavior of the wire is represented by using an auxiliary atomic proposition $broken$ and the program action: $out \neq in \wedge \neg\, broken \rightarrow out := in$. The incorrect behavior of the wire, once a fault occurs, is represented by the program action that sets out to 0 provided that the state of the wire is broken: $broken \rightarrow out := 0$. The stuck-at-low-voltage fault is represented by the fault action: $\neg\, broken \rightarrow broken := true$.

Should it be of interest to capture that only a bounded number k of wires can be stuck-at-low-voltage, an auxiliary variable $brokencount$ can be used to strengthen the stuck-at-low-voltage fault action to

$$\neg broken \wedge brokencount < k \rightarrow broken := true, brokencount := brokencount + 1.$$

To reinforce the use of fault actions, here are several more examples:

—A fault action that captures repair of the wire is: $broken \rightarrow broken := false$.
—Taken together, the stuck-at-low-voltage and the repair fault action capture intermittent stuck-at faults.
—Consider an *omission fault* by which a buffer loses its content. In this case, letting the proposition is_full denote that the buffer has content, the omission fault is represented by the action: $is\_full \rightarrow is\_full := false$.
—Consider a *timing fault* by which access to the contents of a buffer is delayed. By introducing an auxiliary proposition $is\_delayed$, the timing fault is represented by the two actions:

$$is\_full \rightarrow is\_full := false, is\_delayed := true, \text{ and}$$
$$\neg is\_full \wedge is\_delayed \rightarrow is\_full := true, is\_delayed := false.$$

—*Fail-stop faults* [Schneider 1984, 1990]: A fault in this class stops a process from executing any actions, possibly forever. Thus, fail-stop faults effectively corrupt program processes in a detectable (i.e., other processes are explicitly notified of the failure), uncorrectable, and potentially permanent manner. Fail-stop faults may thus be distinguished from *crash* faults, which are undetectable in purely asynchronous systems. That is, we assume some underlying failure detection mechanism [Chandra and Toueg 1996] which provides the explicit notification of the fault to the other processes. Thus, fail-stop faults are more benign than crash faults. The assumption of failure detection, of course, implies a departure from a purely asynchronous model of concurrent computation [Fischer et al. 1985]. If some processes fail permanently, then the remaining processes can be thought of as a "subprogram" of the original program.

—*General state faults*: A fault in this class arbitrarily perturbs the state of a process or a shared variable, without being detected by any process. As a result, the program may be placed in a state it would not have reached under normal computation of the processes. Such state faults are general in the sense that by a sequence of these faults the program may reach arbitrary global states [Jayaram and Varghese 1996]; thus, general state faults effectively corrupt global state in an undetectable, correctable, and transient manner.

We will use fail-stop and general state faults in the mechanical synthesis examples we present in this paper. Needless to say, our synthesis method suffices for the many other concurrent program faults that are captured by fault actions.

In general, we will need to specify constraints on the update of the auxiliary atomic propositions,[2] and their relation to the "regular" atomic propositions, that is, those appearing in the problem specification. This is provided by a *problem-fault coupling specification*, which is a CTL formula $\mathsf{AG}(coupling\text{–}spec)$, where *coupling–spec* itself can be any CTL formula. The coupling specification could, for instance, restrict the local states which a process can be in when an auxiliary proposition is true, or express that an auxiliary atomic proposition cannot be changed by any process. For example, a problem-fault coupling specification for the wire example with the stuck-at-low-voltage fault is $\mathsf{AG}(broken \Rightarrow \mathsf{AG}broken) \wedge \mathsf{AG}((broken \wedge \neg out) \Rightarrow \mathsf{AG}\neg out)$, that is, once broken, always broken, and once broken and output is low, then output stays low forever.

2.4 Fault-Tolerant Kripke Structures

To model the occurrence of faults, we use a *fault-tolerant Kripke structure* $M_F = (S_0, S, A, A_F, L)$, where S_0, S, A, L are as before, and $A_F \subseteq S \times F \times S$ is a set of *fault-transitions*. F is a set of fault-actions as discussed above. A fault-transition labeled with $a \in F$ models the occurrence of fault action a. Note that A and A_F are disjoint, by definition.

[2]We assume that all auxiliary state variables are represented as a finite number of auxiliary atomic propositions.

A fullpath (path) in M_F can contain transitions drawn from A and from A_F. A *fault-free fullpath* (*fault-free path*) is a fullpath (path) that contains no fault-transitions, that is, its transitions are drawn only from A. An *initialized fullpath* (*initialized path*) is a fullpath (path) whose first state is an initial state (i.e., in S_0).

A global state is *normal* iff it lies on some fault-free initialized fullpath. A global state that (1) lies on no initialized fault-free fullpath, and (2) is the final state of an initialized path that ends in a fault-transition, is *perturbed*. All other states are *recovery* states. Thus, perturbed states can only be reached from initial states via paths that contain at least one fault-transition. In particular, a state that can be reached by both a fault-free initialized path, and an initialized path that ends in a fault-transition, is a normal (and not a perturbed) state. Let S_F denote that set of all perturbed states.

The set of transitions A is partitioned into *normal-transitions*—those that start in a normal state, and *recovery-transitions*—those that start in a perturbed or recovery state.

The appropriate notion of satisfaction in a fault-tolerant Kripke structure is given by $\models_n$, a version of the $\models$ relation that is relativized to fault-free fullpaths.[3] The definition of $\models_n$ is verbatim identical to that of $\models$ above, except that every occurrence of "fullpath" is replaced by "fault-free fullpath." We give the clauses that differ from the above definition of $\models$:

$M, s^0 \models_n \mathsf{A}[f \mathsf{U} g]$ iff for all fault-free fullpaths $\pi = (s^0, s^1, \ldots)$ in M that start in s^0,
  there exists $i \in [0 : |\pi|]$ such that
    $M, s^i \models g$ and for all $j \in [1 : (i-1)]$: $M, s^j \models f$;

$M, s^0 \models_n \mathsf{E}[f \mathsf{U} g]$ iff for some fault-free fullpath $\pi = (s^0, s^1, \ldots)$ in M that starts in s^0,
  there exists $i \in [0 : |\pi|]$ such that
    $M, s^i \models g$ and for all $j \in [1 : (i-1)]$: $M, s^j \models f$.

2.5 Fault-Tolerance Properties

In the presence of faults, a concurrent program need not always satisfy its given specification. But it is desirable then that when faults occur, the program at least satisfy some "tolerance" property, which may be potentially weaker than the given specification. The choice of the tolerance is, of course, dependent on the context and application; nevertheless it is generally possible to classify the tolerance property in terms of how (and whether) the safety and the liveness parts of the given specification are respected in the presence of the faults. In one class, *masking tolerance*, both the safety and the liveness parts are always respected; in another, *fail-safe tolerance*, only the safety part but not necessarily the liveness part is respected; and in yet another, *nonmasking tolerance*, the liveness part is always respected but the safety part is only eventually

[3]The idea of relativized satisfaction comes from Emerson and Lei [1985] where it was used to handle fairness in CTL model checking.

respected. (The alternative that only the liveness part is respected but the safety is not ever respected appears to be uncommon.)

Let P be a concurrent program which satisfies a problem specification $problem\text{–}spec = init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec)$, where $init\text{–}spec$ and $global\text{–}spec$ can be any CTL formulae, and let F be a set of fault actions. It follows that in all states reached by program execution in the absence of faults (i.e., the normal states) the CTL formula $global\text{–}spec$ holds. In states reached by program execution in the presence of faults (i.e., the perturbed states), however, $global\text{–}spec$ need not hold in general. We define the following:

—P is *masking tolerant* to F for $problem\text{–}spec$ if and only if $\mathsf{AG}(global\text{–}spec)$ holds at all perturbed states. That is, subsequent execution of P from these states satisfies the desired correctness properties of P.

—P is *nonmasking tolerant* to F for $problem\text{–}spec$ if and only if $\mathsf{AFAG}(global\text{–}spec)$ holds at all perturbed states. That is, subsequent execution of P from these states eventually reaches a state from where the desired correctness properties of P are satisfied.

—P is *fail-safe tolerant* to F for $problem\text{–}spec$ if and only if $\mathsf{AG}(global\text{–}safety\text{–}spec)$ holds at all perturbed states, where $global\text{–}safety\text{–}spec$ consists of all the safety properties in $global\text{–}spec$. That is, subsequent execution of P from these states satisfies the desired safety properties—but not necessarily the liveness properties—of P. We assume that specifications are written in such a way that the safety component of the specification can be extracted. This assumption must be made by any method that guarantees fail-safe tolerance only.[4]

Let $spec \overset{\mathrm{df}}{=} problem\text{–}spec \wedge \mathsf{AG}(coupling\text{–}spec)$, where $problem\text{–}spec = init\text{-}spec \wedge \mathsf{AG}(global\text{–}spec)$ is a problem specification, and $\mathsf{AG}(coupling\text{–}spec)$ is a problem-fault coupling specification. We shall call $spec$ a temporal specification.

Definition 2.1 ($Label_{TOL}$). Given a temporal specification $spec = init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$, define $Label_{TOL}(spec)$ as follows, where $TOL \in \{masking, nonmasking, fail\text{–}safe\}$:

—If $TOL = masking$, $Label_{TOL}(spec)$ is the CTL formula $\mathsf{AG}(global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$. For masking tolerance, the global specification must hold in all perturbed states.

—If $TOL = nonmasking$, $Label_{TOL}(spec)$ is the CTL formula $\mathsf{AFAG}\ (global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$. For nonmasking tolerance, the global specification must eventually hold in all computations starting in perturbed states (provided that faults stop occurring).

—If $TOL = fail\text{–}safe$, $Label_{TOL}(spec)$ is the CTL formula $\mathsf{AG}\ (global\text{–}safety\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$. For fail-safe tolerance, the safety component of the global specification must hold in all perturbed states. Note that recovery to states where $global\text{–}spec$ holds is not required.

[4]See Manolios and Trefler [2001] for a discussion of how a branching time specification can be expressed as a conjunction of a safety specification and a liveness specification.

$Label_{TOL}(spec)$ gives the formula that must be satisfied by perturbed states in order for the synthesized program to have the desired fault-tolerance properties. In all cases, the coupling specification must hold in all perturbed states.

Just as our representation of faults is general enough to capture extant fault-classes, our definition of tolerance properties is general enough to capture the fault-tolerance requirements of extant computing systems. (The interested reader is referred to Arora and Gouda [1993] for a detailed discussion of how these tolerance properties suffice for fault-tolerance in distributed systems, networks, circuits, database management, etc.) To mention but a few examples, systems based on consensus, agreement, voting, or commitment require masking tolerance—or at least failsafe tolerance—whereas those based on reset, checkpointing/recovery, or exception handling typically require nonmasking tolerance.

3. THE SYNTHESIS PROBLEM

The problem of synthesis of fault-tolerant concurrent programs is as follows. Given are

(1) A *problem specification*, which is a CTL formula *problem–spec* of the form $init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec)$, where *init–spec* and *global–spec* can be any CTL formulae. Section 2.2 gives an example problem specification for mutual exclusion.

(2) A *fault specification*, which consists of (1) a set of auxiliary atomic propositions, and (2) a set F of fault actions (guarded commands) over the atomic propositions (including the auxiliary ones). We assume, for the time being, that fault actions cannot reference the shared synchronization variables $x_1, \ldots, x_m$. We show how to remove this restriction in Section 5.3 below. We also assume that fault actions always terminate.

For example, the fault specification of the stuck-at-low-voltage-fault, as given in Section 2.3, is $\{broken\}$ and $\{\neg broken \rightarrow broken := true\}$. Execution of the fault actions models the occurrence of faults.

(3) A *problem-fault coupling specification*, $\mathsf{AG}(coupling\text{–}spec)$, where *coupling–spec* can be any CTL formula. It relates the atomic propositions in the problem specification with those in the fault specification. For example, a problem-fault coupling specification for the wire example of Section 2.3 with the stuck-at-low-voltage fault is $\mathsf{AG}(broken \Rightarrow \mathsf{AG}broken) \wedge \mathsf{AG}((broken \wedge \neg out) \Rightarrow \mathsf{AG}\neg out)$, that is, once broken, always broken, and once broken and output is low, then output stays low forever.

(4) A *type of tolerance* $TOL \in \{masking, nonmasking, fail\text{–}safe\}$, which specifies the desired tolerance property.

Required is to synthesize a concurrent program that

(1) satisfies $init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec)$ in the absence of faults, and
(2) satisfies $\mathsf{AG}(coupling\text{–}spec)$ in the absence of faults, and
(3) is TOL-tolerant to F for $init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec)$.

Let $M_F = (S_0, S, A, A_F, L)$ be the Kripke structure generated by the execution of the synthesized program in the presence of the set of faults F, and let S_F be the set of perturbed states in M_F. Then, we require

(1) $M_F, S_0 \models_n \mathit{init\text{–}spec} \wedge \mathsf{AG}(\mathit{global\text{–}spec}) \wedge \mathsf{AG}(\mathit{coupling\text{–}spec})$, and

(2) $M_F, S_F \models_n \mathit{Label}_{TOL}(\mathit{spec})$.

Note that $\mathsf{AG}(\mathit{coupling\text{–}spec})$ is required to be satisfied in all states, since it is a conjunct of $\mathit{Label}_{TOL}(\mathit{spec})$ for all tolerances TOL.

4. THE CTL DECISION PROCEDURE

We provide here an overview of the CTL decision procedure [Emerson 1981; Emerson and Clarke 1982; Emerson 1990], together with necessary technical definitions (taken from Emerson [1981, chapter 4] and Emerson and Clarke [1982]). Since Emerson [1981, chapter 4] deals with the logic UB, which is obtained from CTL by replacing AU, EU, AW, EW by AF, EF, AG, EG respectively, we make the necessary extensions needed to account for the until modality of CTL.

A CTL formula f is in *positive normal form* iff any negations within f are applied only to atomic propositions. Any CTL formula can be converted to positive normal form by "pushing" the negations inwards, using the appropriate dualities for the abbreviations $\vee$, AW, EW, and AX_i, for example, $\neg\mathsf{A}[g\mathsf{U}h] \equiv \mathsf{E}[\neg g\mathsf{W}\neg h]$.

Definition 4.1 (*Fisher-Ladner Closure*). If f is a CTL formula, then $cl(f)$, the generalized Fisher-Ladner closure of f, is given by

$$\begin{aligned}
&cl(p) = p \text{ for atomic proposition } p,\\
&cl(g \wedge h) = \{g \wedge h\} \cup cl(g) \cup cl(h),\\
&cl(\neg f) = \{\neg f\} \cup cl(f),\\
&cl(\mathsf{A}[g\mathsf{U}h]) = \{\mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h]\} \cup cl(g) \cup cl(h),\\
&cl(\mathsf{E}[g\mathsf{U}h]) = \{\mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h]\} \cup cl(g) \cup cl(h),\\
&cl(\mathsf{AF}g) = \{\mathsf{AF}g, \mathsf{AXAF}g\} \cup cl(g),\\
&cl(\mathsf{EF}g) = \{\mathsf{EF}g, \mathsf{EXEF}g\} \cup cl(g),\\
&cl(\mathsf{EX}_i g) = \{\mathsf{EX}_i g\} \cup cl(g),\\
&cl(\mathsf{A}[g\mathsf{W}h]) = \{\mathsf{A}[g\mathsf{W}h], \mathsf{AXA}[g\mathsf{W}h]\} \cup cl(g) \cup cl(h),\\
&cl(\mathsf{E}[g\mathsf{W}h]) = \{\mathsf{E}[g\mathsf{W}h], \mathsf{EXE}[g\mathsf{W}h]\} \cup cl(g) \cup cl(h),\\
&cl(\mathsf{AG}g) = \{\mathsf{AG}g, \mathsf{AXAG}g\} \cup cl(g),\\
&cl(\mathsf{EG}g) = \{\mathsf{EG}g, \mathsf{EXEG}g\} \cup cl(g),\\
&cl(\mathsf{AX}_i g) = \{\mathsf{AX}_i g\} \cup cl(g).
\end{aligned}$$

Let $|f|$, the length of f, be the sum of the number of occurrences of atomic propositions, propositional connectives, and CTL modalities, in f (with multiple occurrences of the same proposition, connective, or modality counting). Then, $|cl(f)| \leq 2|f|$.

A CTL formula is *elementary* iff it is an atomic proposition, the negation of an atomic proposition, or has either AX_j or EX_j as its main connective. We classify a nonelementary formula as either a conjunctive formula $\alpha \equiv \alpha_1 \wedge \alpha_2$ or a disjunctive formula $\beta \equiv \beta_1 \vee \beta_2$, as follows:

$\alpha = g \wedge h$	$\alpha_1 = g$	$\alpha_2 = h$
$\alpha = \mathsf{A}[g\mathsf{W}h]$	$\alpha_1 = h$	$\alpha_2 = g \vee \mathsf{AXA}[g\mathsf{W}h]$
$\alpha = \mathsf{E}[g\mathsf{W}h]$	$\alpha_1 = h$	$\alpha_2 = g \vee \mathsf{EXE}[g\mathsf{W}h]$
$\alpha = \mathsf{AG}g$	$\alpha_1 = g$	$\alpha_2 = \mathsf{AXAG}g$
$\alpha = \mathsf{EG}g$	$\alpha_1 = g$	$\alpha_2 = \mathsf{EXEG}g$
$\alpha = \mathsf{AX}g$	$\alpha_1 = \mathsf{AX}_1 g \quad \ldots$	$\alpha_I = \mathsf{AX}_I g$
$\beta = g \vee h$	$\beta_1 = g$	$\beta_2 = h$
$\beta = \mathsf{A}[g\mathsf{U}h]$	$\beta_1 = h$	$\beta_2 = g \wedge \mathsf{AXA}[g\mathsf{U}h]$
$\beta = \mathsf{E}[g\mathsf{U}h]$	$\beta_1 = h$	$\beta_2 = g \wedge \mathsf{EXE}[g\mathsf{U}h]$
$\beta = \mathsf{AF}g$	$\beta_1 = g$	$\beta_2 = \mathsf{AXAF}g$
$\beta = \mathsf{EF}g$	$\beta_1 = g$	$\beta_2 = \mathsf{EXEF}g$
$\beta = \mathsf{EX}g$	$\beta_1 = \mathsf{EX}_1 g \quad \ldots$	$\beta_I = \mathsf{EX}_I g$

Note that nonelementary formulae whose main connective is a temporal modality are classified according to the fixpoint characterization of the modality, for example: $\mathsf{AF}g \equiv g \vee \mathsf{AXAF}g$, so $\mathsf{AF}g$ is a β formula. Also, the expansions of AX, EX involve I formulae (recall that I is the number of processes) rather than just two. In subsequent discussion we shall, for sake of brevity, assume that all of these expansions produce exactly two formulae. The generalization to deal with I formulae will always be straightforward. We refer to the above transformations as *α-β expansions*.

A set of formulae $\mathcal{F}$ is *downward closed* iff (1) if $\alpha \in \mathcal{F}$ then $\alpha_1, \alpha_2 \in \mathcal{F}$, and (2) if $\beta \in \mathcal{F}$ then $\beta_1 \in \mathcal{F}$ or $\beta_2 \in \mathcal{F}$.

Definition 4.2 (*AND/OR Graph, Fullgraph*). An *AND/OR graph K* is a tuple $(V_C, V_D, A_{CD}, A_{DC}, L)$ with the following components:

(1) V_C, a set of AND-nodes;
(2) V_D, a set of OR-nodes;
(3) $A_{CD} \subseteq V_C \times [1{:}I] \times V_D$, a set of AND-OR transitions;
(4) $A_{DC} \subseteq V_D \times V_C$, a set of OR-AND transitions;
(5) $L : V_C \cup V_D \mapsto 2^{cl(f)}$, a labeling function which labels each node in $V_C \cup V_D$ with a subset of $cl(f)$.

A *fullgraph* is an AND/OR graph in which A_{CD} is a function from V_C to V_D, that is, every AND-node has exactly one successor. We abuse notation and write $(u, v) \in A_{CD}$ for "there exists $i \in [1{:}I]$ such that $(u, i, v) \in A_{CD}$."

Given a CTL formula f_0 (which has first been rewritten into positive normal form), the CTL decision procedure first constructs a particular kind of AND/OR graph (a tableau) T_0 for f_0. We use $c, c', \ldots$ to denote AND-nodes, $d, d', \ldots$ to denote OR-nodes, and $e, e', \ldots$ to denote nodes of either type. Each node is labeled with a subset of $cl(f_0)$, and no two AND-nodes (OR-nodes) have the same label. A model for f_0 is extracted from the tableau by taking the AND-nodes of the tableau as states of the model, and preserving the local transition structure of the tableau. Soundness of the CTL decision procedure is established by showing that, in the final model, every state satisfies all the formulae in its label.

The CTL decision procedure constructs the tableau T_0 by starting with a single OR-node d_0 labeled with $\{f_0\}$, and repeatedly constructing successors of "frontier" nodes until there is no more change. The set of AND-node successors $Blocks(d)$ of an OR-node d is determined as follows. d is "expanded" into a tree using the above characterization of nonelementary formulae as α or β. Suppose e is a leaf in the tree constructed so far, and $f \in L(e)$. If $f \equiv \alpha_1 \wedge \alpha_2$ is an α formula, then add a single son to e with label $L(e) - \{f\} \cup \{\alpha_1, \alpha_2\}$. If $f \equiv \beta_1 \vee \beta_2$ is a β formula, then add two sons to e with labels $L(e) - \{f\} \cup \{\beta_1\}$, $L(e) - \{f\} \cup \{\beta_2\}$. For example, $\mathsf{AF}g \equiv g \vee \mathsf{AXAF}g$, so $\mathsf{AF}g$ "generates" two successors, one with g in its label and one with $\mathsf{AXAF}g$ in its label. These successors correspond to the two different ways of satisfying an eventuality. The successor labeled with g certifies that the eventuality $\mathsf{AF}g$ is fulfilled, while the successor labeled with $\mathsf{AXAF}g$ *propagates* $\mathsf{AF}g$. On the other hand, $\mathsf{AG}g \equiv g \wedge \mathsf{AXAG}g$, and so $\mathsf{AG}g$ generates only one successor, labeled with both g and $\mathsf{AXAG}g$. This tree construction terminates when all leaves contain only elementary formulae in their labels. This must happen, since each expansion removes one nonelementary formula and replaces it with one or two smaller formulae. Upon termination, let $Blocks(d)$ contain one AND-node c for each leaf node, and let the label of each c be the union of all node labels along the path from the corresponding leaf back to the root d of the tree. Clearly, $L(c)$ is downward-closed by virtue of the tree construction algorithm. The nodes in $Blocks(d)$ can be regarded as embodying all of the different ways in which the (conjunction of the) formulae in the label of d can be satisfied. The reader is referred to Emerson [1981] and Emerson and Clarke [1982] for full details, where it is also shown that (1) $L(d)$ is satisfiable iff $L(c)$ is satisfiable for at least one $c \in Blocks(d)$, and (2) $L(c)$ is satisfiable iff $LE(c)$ is satisfiable, for $c \in Blocks(d)$ and $LE(c) = \{f \in L(c) \mid f \text{ is elementary}\}$.

The set $Tiles(c)$ of OR-node successors of an AND-node c is defined to be $\bigcup_{i \in [1:I]} Tiles_i(c)$, where $Tiles_i(c)$ is the set of OR-node successors of c that are associated with process i. Assume c is labeled with n formulae of the form $\mathsf{AX}_i g$, namely, $\mathsf{AX}_i g_1, \ldots, \mathsf{AX}_i g_n$, and m formulae of the form $\mathsf{EX}_i h$, namely, $\mathsf{EX}_i h_1, \ldots, \mathsf{EX}_i h_n$. Then, $Tiles_i(c) \overset{\text{df}}{=} \{D_i^1, \ldots, D_i^m\}$, where $D_i^j = \{\mathsf{AX}_i g_1, \ldots, \mathsf{AX}_i g_n\} \cup \{\mathsf{EX}_i h_j\}$, for $j \in [1:m]$. Finally, the edge from c to every node in $Tiles_i(c)$ is labeled with the process index i, to indicate that this successor is associated with process i. There are two special cases in the definition of $Tiles(c)$. First, if c has no nexttime formulae in its label, then $Tiles(c) = \{d\}$, where $L(d) = L(c)$, and $Blocks(d) = \{c\}$, that is, c is given a single "dummy" successor labeled with the same formulae. Second, if only EX_i-formulae are missing, then c is split into I AND-nodes $c_1, \ldots, c_I$ (which have the same incoming edges) and $L(c_i)$ is set to $L(c) \cup \{\mathsf{EX}_i true\}$, for all $i \in [1:I]$. The OR-node successors of each c_i are then computed as above. From the above discussion, we see that $Tiles(c)$ is exactly the set of successors required to satisfy all of the nexttime formulae in the label of c. In Emerson [1981] it was shown that $L(c)$ is satisfiable iff $L(d)$ is satisfiable for all $d \in Tiles(c)$, and $LP(c)$ is satisfiable, where $LP(c) = \{f \in L(c) \mid f \text{ is a proposition or its negation}\}$.

We continue the process of generating successors of frontier nodes (which we refer to as "expanding" a node, in the sequel) until there are no more frontier nodes, that is, every node in T_0 has at least one successor. If, when a node e is being expanded, some successor e' of e has the same label as an already present node e'' of the same type (i.e., AND or OR), then we identify e' and e'', that is, delete e' and add a transition from e to e''. This ensures that every AND-node (OR-node) in T_0 has a unique label. Since there are at most $2^{|cl(f_0)|}$ different labels, the expansion process must terminate.

Thus, the tableau T_0 for CTL formula f_0 is an AND/OR graph with a root d_0 which is an OR-node with label $\{f_0\}$. Every AND-node c (OR-node d) in T_0 has successors given by $Tiles(c)$ ($Blocks(d)$). A tableau T is written as a tuple $(d, V_C, V_D, A_{CD}, A_{DC}, L)$, where d is the root, and the remaining components are as in Definition 4.2. Before continuing the description of the CTL decision procedure, we need a few more technical definitions.

Definition 4.3 (*Prestructure*). A *prestructure* $G = (V, A, L)$ for a CTL formula f consists of a set of nodes V, a set of transitions $A \subseteq V \times V$, and a labeling $L : V \mapsto cl(f)$ of each node with a set of formulae.

We use the generic term *graph* to refer to any object which is a prestructure or an AND/OR graph. Let G be a graph with labeling L. We define *frontier*(G), the frontier of G, to be the set of nodes of G that have no successor in G, and *interior*(G), the interior of G, to be the set of all other nodes of G.

Definition 4.4 (*Generated*). A fullgraph $K = (V'_C, V'_D, A'_{CD}, A'_{DC}, L')$ is *generated* by tableau $T = (d, V_C, V_D, A_{CD}, A_{DC}, L)$ iff there exists a generation function $E : V'_C \cup V'_D \mapsto V_C \cup V_D$ such that

(1) $E[V'_C] \subseteq V_C$;
(2) $E[V'_D] \subseteq V_D$;
(3) $L' = L \circ E$;
(4) if (u, i, w) is an edge in A'_{CD}, then $(E(u), i, E(w))$ is an edge in A_{CD};
(5) if (w, u) is an edge in A'_{DC}, then $(E(w), E(u))$ is an edge in A_{DC};
(6) if an AND-node u of K is an interior node, then for every OR-node d (of T_0) in $Blocks(E(u))$, there exists an OR-node w of K such that $(u, w) \in A'_{CD}$ and $E(w) = d$;
(7) every OR-node w of K has at least one successor in K.

A prestructure $G = (V'', A'', L'')$ is *generated* by tableau T iff there exists a fullgraph $K = (V'_C, V'_D, A'_{CD}, A'_{DC}, L')$ generated by tableau T and such that

(1) $V'' = V'_C$;
(2) $A'' = \{(u, i, v) \in V'' \times [1:I] \times V'' \mid \exists w \in V'_D,\ (u, i, w) \in A'_{CD} \text{ and } (w, v) \in A'_{DC}\}$;
(3) $L'' = L'$ restricted to V'_C.

In a prestructure or an AND/OR graph, whenever there is an edge from node u to node v that is labeled with process index i, then we say that v is a P_i-*successor* of u.

DeleteP. Delete any propositionally inconsistent node.

DeleteOR. Delete any OR-node all of whose successors are already deleted.

DeleteAND. Delete any AND-node one of whose successors is already deleted.

*Delete*AU. Delete any node e such that $\mathsf{A}[g\mathsf{U}h] \in L(e)$ and there does not exist a full subdag rooted at e such that $h \in L(c')$ for every frontier node c' and $g \in L(c'')$ for every interior AND-node c''.

*Delete*EU. Delete any node e such that $\mathsf{E}[g\mathsf{U}h] \in L(e)$ and there does not exist an AND-node c' reachable from e via a path π such that $h \in L(c')$ and for all AND-nodes c'' along π up to but not necessarily including c', $g \in L(c'')$.

Fig. 1. The deletion rules for the CTL decision procedure.

Definition 4.5 (*Directly Embedded*). A fullgraph K is *directly embedded* in tableau T_0 if K is generated by T_0 and the generation function is one-to-one.

Definition 4.6 (*Fulfilled*). The eventuality $\mathsf{A}[g\mathsf{U}h]$ is *fulfilled* for node e in graph G provided that, for every path starting at e in G, there is some node e' along the path such that $h \in L(e')$, and for every node e'' on the path up to (but not necessarily including) node e', $g \in L(e'')$.

The eventuality $\mathsf{E}[g\mathsf{U}h]$ is *fulfilled* for node e in graph G provided that, for some path starting at e in G, there is some node e' along the path such that $h \in L(e')$, and for every node e'' on the path up to (but not necessarily including) node e', $g \in L(e'')$.

Definition 4.7 (*Full Subdag*). A *full subdag D* rooted at node e in T_0 is a directed acyclic subgraph of T_0 satisfying all of the following conditions:

(1) node e is the unique node from which all other nodes in D are reachable;
(2) for every AND-node c in D, if c has any sons in D, then every successor of c in T_0 is a son of c in D;
(3) for every OR-node d, there exists precisely one AND-node c in T_0 such that c is a son of d in D.

The next step of the CTL decision procedure is to apply the set of *deletion rules* given in Figure 1 to T_0. Roughly speaking, these rules remove all nodes that are either propositionally inconsistent, or do not have enough successors, or are labeled with an eventuality formula which is not fulfilled. The presence of a suitable full subdag rooted at e serves to certify the fulfillment of the corresponding eventuality in $L(e)$. We repeatedly apply the deletion rules until there is no change. Since each application removes one node, and T_0 is finite, this process must terminate. Upon termination, if the root of T_0 is has been removed, then f_0 is unsatisfiable. Otherwise f_0 is satisfiable, in which case let T^* be the tableau induced by the remaining nodes. In Emerson and Clarke [1982] it was shown how to extract an actual model from T^*, by a process of "unraveling." From this model, a correct concurrent program can be produced by projecting onto the individual processes.

The unraveling step is as follows. For each AND-node c in T^*, a "fragment" FRAG[c] is constructed. FRAG[c] is a directed acyclic graph whose nodes are

AND-nodes, and whose local structure is taken from T^*, that is, $c' \to c''$ in FRAG[c] only if $c' \to d \to c''$ in T^* for some OR-node d. In addition, c is the root of FRAG[c] and all eventualities in the label of c are fulfilled in FRAG[c]. Given that an AND-node c was not removed by the deletion rules, it follows that, for each eventuality $g \in L(c)$, T^* contains at least one full subdag D with root c and in which g is fulfilled. Let DAG[c, g] be the directed acyclic prestructure that results from removing all the OR-nodes in D and connecting up the AND-nodes so that $c' \to c''$ in DAG[c, g] only if $c' \to d \to c''$ in D for some OR-node d. By construction, DAG[c, g] is generated by T^*.

We construct FRAG[c] from the DAG's in T^* as follows. Let $g_1, \ldots, g_m$ be all of the eventualities in $L(c)$, and let *frontier*(FRAG$_j$) denote the frontier of fragment FRAG$_j$. Then

```
let FRAG$_1$ be a copy of DAG[$c, g_1$];
to obtain FRAG$_{j+1}$ from FRAG$_j$, do
    identify any two nodes on the frontier of FRAG$_j$ that have the same
    label;
    **forall** $s' \in frontier$(FRAG$_j$) do
        /* let $c'$ be the AND-node in $T^*$ that $s'$ is a copy of */
        **if** $g_{j+1} \in L(s')$ **then**
            attach a copy of DAG[$c', g_{i+1}$] to FRAG$_j$ at $s'$
    **endfor**;
/* call the resulting directed acyclic graph FRAG$_{j+1}$ */
let FRAG[$c$] be the directed acyclic graph obtained from FRAG$_m$ by
identifying any two nodes in $frontier$(FRAG$_m$) with the same label
```

If $L(c)$ contains no eventualities, then FRAG[c] consists of c together with enough local successors to satisfy all of the formulae that have AX_j or EX_j as main connective. That is, for each $d \in Tiles(c)$, choose a $c' \in Blocks(d)$ and add c' as a successor of c. Identify all such c' with the same label. Then FRAG[c] consists of c together with all such c'.

In Emerson [1981] it was shown that FRAG[c] is an acyclic prestructure generated by T^* whose root node s_0 is a copy of c, and that all eventualities in $Label(s_0)$ ($= Label(c)$) are fulfilled for s_0 in FRAG[c].

Finally, the FRAGs are connected together (essentially, the frontier nodes of a FRAG are identified with root nodes of other FRAGs) to form a Kripke structure in which all eventualities are fulfilled. This structure is a model of f_0. The procedure is as follows:

```
choose $c_0 \in Blocks(d_0)$ arbitrarily (recall that $d_0$ is the root of $T$);
let $M_1$ = FRAG[$c_0$];
to obtain $M_{i+1}$ from $M_i$, do
    **forall** $s \in frontier(M_i)$ **do**
        /* let $c$ be the AND-node in $T^*$ that $s$ is a copy of */
        **if** there exists $s' \in interior(M_i)$ such that $s'$ is also a copy
            of $c$, and a copy of FRAG[$c$] is directly embedded in $M_i$ with
                root $s'$, **then** identify $s$ and $s'$
        **else**
```

```
            replace $s$ by a copy of FRAG[$c$]
        **endif**
    **endfor**
/* call the resulting graph $M_{i+1}$ */
The construction halts with $i = N$ when $frontier(M_N)$ is empty. Let
$M = M_N$.
Let $M = (S, A, L)$ and let $M_0 = (S, A, L_0)$ where $L_0$ is $L$ restricted to the
propositions occurring in $f_0$. $M_0$ is a model of $f_0$.
```

By construction, each fragment occurs at most once in M. Each step (i.e., obtaining M_{i+1} from M_i) either adds a new fragment or reduces the number of frontier nodes by 1. Since there is one fragment for each AND-node, the number of fragments is the same as the number of AND-nodes, which is bounded by $2^{|cl(f_0)|}$. Thus, after enough steps, no new fragments can be added, and eventually the frontier must become empty. Thus, the above procedure is guaranteed to terminate. It was shown in Emerson [1981] that $M_0, s \models f$ for every $s \in S$ and $f \in L(s)$. In particular, since $f_0 \in L(c_0)$ by definition of c_0 and $Blocks(d_0)$, we have $M_0, c_0 \models f_0$, and so M_0 is indeed a model of f_0.

From the model M_0, a program can be extracted by projecting onto the individual process indices. For example, Figure 8 shows a Kripke structure and a process P_1 extracted from it. The arc from N_1 to T_1 labeled with $N_2 \vee C_2$ is derived by projecting the transitions from $[N_1\ N_2]$ to $[T_1\ N_2]$ and $[N_1\ C_2]$ to $[T_1\ C_2]$ in the Kripke structure onto the process index 1.

5. SYNTHESIS OF FAULT-TOLERANT CONCURRENT PROGRAMS

To synthesize a program that has guaranteed behavioral properties after the occurrence of faults, we have to (1) represent the occurrence of faults, and (2) synthesize the recovery behavior that conforms to the required behavioral properties. We represent the occurrence of faults by fault-transitions, and we represent the appropriate recovery behavior by recovery-transitions (see Section 2.4 above). Thus, our synthesis method first generates a tableau that, in addition to the normal-transitions that represent the behavior of the program in the absence of faults (and which are the only transitions that the previous synthesis method of Emerson and Clarke [1982] produces), also contains the fault-transitions that represent the occurrence of all the faults given in the synthesis problem specification (see Section 3), and the recovery-transitions that generate a recovery behavior that satisfies the required tolerance property (e.g., masking, fail-safe, or nonmasking). The required recovery behavior is enforced by labeling the perturbed states appropriately. The suitable labeling is generated from the problem-fault coupling specification and the type of tolerance required, both of which are part of the synthesis problem specification. The role of the problem-fault coupling specification is to characterize the information retained by a process after the occurrence of a fault, and its relationship to the state of the process when the fault occurred. As such, it will usually be fairly straightforward to write. We further discuss the role of the problem-fault coupling specification in the examples given in Section 6 below.

Section 5.1 presents technical definitions that we use to model the specified faults and fault-tolerance properties. Section 5.2 presents our synthesis method.

5.1 Technical Definitions for Modeling Faults

Recall that faults are modeled as nondeterministic actions (guarded commands) whose execution perturbs the current global state (Section 2.3). If a is a fault action, then $a.guard$, $a.body$ denote the guard, body, respectively, of the guarded command that models a. If c is an AND-node, let $L(c)\!\uparrow\!\mathcal{AP}$ denote the set of atomic propositions that are true in c, and $c(a.guard)$ the truth-value of $a.guard$ in c. Let $\varphi, \psi \subseteq \mathcal{AP}$. Then define $\{\varphi\}\, a.body\, \{\psi\}$ to mean that, if $a.body$ is executed in a state in which exactly the propositions in φ are true, then one possible outcome is a state in which exactly the propositions in ψ are true.

Definition 5.1.1 ($\xrightarrow{a,TOL}$). Let c be an AND-node, d be an OR-node, a a fault action, and TOL a tolerance. Then

$$c \xrightarrow{a,TOL} d \quad \text{if and only if} \quad \exists\varphi \subseteq \mathcal{AP} : c(a.guard) = true \text{ and } \{L(c)\!\uparrow\!\mathcal{AP}\}\, a.body\, \{\varphi\} \text{ and } L(d) = \varphi \cup Label_{TOL}(spec).$$

$c \xrightarrow{a,TOL} d$ intuitively means that fault action a can occur in AND-node c, and that its occurrence can lead to OR-node d. (Recall that AND-nodes correspond to states in the final model.) $L(d)$ is the label of d. The propositional component of $L(d)$ results from applying $a.body$ to the propositions in $L(c)$, while the temporal component $Label_{TOL}(spec)$ of $L(d)$ is determined solely by the problem specification and the desired type of fault-tolerance.

Definition 5.1.2 ($FaultStates$, $FaultTrans$). Given a set F of fault actions, a set V of AND-nodes, and a tolerance TOL, define

$$FaultStates(F, TOL, V) = \{d \mid \exists a \in F,\ \exists c \in V : c \xrightarrow{a,TOL} d\},$$
$$FaultTrans(F, TOL, V) = \{(c, F, d) \mid \exists a \in F,\ \exists c \in V : c \xrightarrow{a,TOL} d\}.$$

$FaultStates(F, TOL, V)$ is the set of OR-nodes reached by executing some fault action of F in some AND-node of V, and $FaultTrans(F, TOL, V)$ is the set of fault-transitions generated by executing some fault action of F in some AND-node of V.

A fault-free path in a tableau is a path that contains no fault-transitions. Analogously to Section 2.4, we define a node to be *normal* iff it lies on some fault-free initialized path. A node that (1) lies on no initialized fault-free path, and (2) is the final state of an initialized path that ends in a fault-transition, is *perturbed*. All other nodes are *recovery* nodes.

In a structure that incorporates fault-transitions, we need to redefine our notions of fulfillment of eventualities.

Definition 5.1.3 (*Fault-Free-Fulfilled*). The eventuality $\mathsf{A}[g\mathsf{U}h]$ is *fault-free-fulfilled* for node e in graph G provided that, for every *fault-free* path starting at e in G, there is some node e' along the path such that $h \in L(e')$, and for every node e'' on the path up to (but not necessarily including) node e', $g \in L(e'')$.

The eventuality $\mathsf{E}[g \mathsf{U} h]$ is *fault-free-fulfilled* for node e in graph G provided that, for some *fault-free* path starting at e in G, there is some node e' along the path such that $h \in L(e')$, and for every node e'' on the path up to (but not necessarily including) node e', $g \in L(e'')$.

5.2 The Synthesis Method

Our synthesis method first generates a tableau that contains normal-transitions, which represent the behavior of the program in the absence of faults, fault-transitions, which represent the occurrence of all the faults given in the problem specification, and recovery-transitions, which generate a recovery behavior that satisfies the required tolerance property.

In the absence of faults, we require both the problem specification $init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec)$ and the problem-fault coupling specification $\mathsf{AG}(coupling\text{–}spec)$ to hold. Thus, the initial OR-node d_0 of the tableau will be labeled with the temporal specification $spec = init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$. We start with d_0 and construct the tableau $T_0 = (d_0, V_C^0, V_D^0, A_{CD}^0, A_{DC}^0, L^0)$ for *spec* in a similar way to the CTL decision procedure, except that the fault- and recovery-transitions are also generated.

The tableau generation is done incrementally. At each stage, an "unexpanded" node is selected, and its successors are constructed. If the node is an AND-node c, then, in addition to all the successors required to satisfy all the formulae of the forms $\mathsf{AX}_i\, g$ and $\mathsf{EX}_i\, h$ (the "nexttime" formulae) in the label of c, we also add all the successors that can be generated by applying fault actions to c. This is because AND-nodes correspond to states in the final model, and so we must represent all the faults that can occur in the state corresponding to c. This is done by applying all the fault-actions to c. Applying a fault-action to c generates (when the fault action's guard is true in c) fault-transitions leading to *fault-successor* OR-nodes of c. An OR-node d (including the fault-successor OR-nodes) is expanded as discussed above in Section 4 for the CTL decision procedure, that is, by computing $Blocks(d)$. The AND-node successors of d (i.e., the AND-nodes in $Blocks(d)$) correspond to perturbed states in the final model (i.e., states that result from the occurrence of a fault) provided that they do not lie on a fault-free path. Otherwise, the occurrence of the fault-action generates an AND-node that could also have resulted from normal execution, and so this AND-node does not correspond to a perturbed state in the final model. We call such occurrences of fault actions *insignificant*, and all other occurrences *significant*. The expansion of AND-nodes that do correspond to perturbed states in the final model then results in recovery-transitions, which generate the required behavior of the synthesized program after a significant fault occurrence. Program transitions from a normal state, that is, a state reachable from d_0 via a fault-free path, are normal transitions (see also Section 2.4).

The OR-node successors of an AND-node c are therefore computed as $Tiles(c) \cup FaultStates(F, TOL, \{c\})$, that is, as the union of (1) $Tiles(c)$, the "non-fault-successors," that are required to satisfy all nexttime formulae in $L(c)$, and (2) the fault-successor OR-nodes that arise from applying the fault-actions in F to c. Note that the labels of fault-successors are computed as described above (in the definition of *FaultStates*), and that the tolerance properties of the program are defined by the requirement that all formulae in the labels of (the AND-node successors of) the fault-successor OR-nodes hold in the final model.

The modeling of fault-occurrence (by means of fault-transitions), and the generation of the recovery-transitions, are both intertwined with the generation of normal-transitions that constitute the program behavior in the absence of faults. The tableau T_0 encodes a model (state transition graph) for a fault-tolerant program that satisfies the synthesis problem specification, provided that some such program exists.

As stated in Section 3, we only require the formulae in the label of a state (whether perturbed or not) to hold under a notion of satisfaction that is relativized to fault-free fullpaths. Thus, only fault-free fullpaths are considered when evaluating the truth of a formula in the label of a node of the tableau T_0. This obviously affects the application of the deletion rules, which eliminate portions of T_0 that cause a violation of the specification. Thus, the appropriate notion of full subdag to be used for both applying the deletion rules, and then for use in the "unwinding" procedure (constructing the fragments and then pasting them together to obtain the final model), is one in which the AND-nodes must have all of their non-fault-successors (but fault-successors may be absent).

Definition 5.2.1 (*Fault-Free Full Subdag*). A *fault-free full subdag D* rooted at node e in T_0 is a directed acyclic subgraph of T_0 satisfying all of the following conditions:

(1) Node e is the unique node from which all other nodes in D are reachable.
(2) For every AND-node c in D, if c has any sons in D, then every non-fault-successor of c in T_0 (i.e., every $d \in \mathit{Tiles}(c)$) is a son of c in D.
(3) For every OR-node d in D, there exists precisely one AND-node c in T_0 such that c is a son of d in D.

We now describe the steps of our synthesis method. We give pseudocode for each step, followed by some discussion. The first step is to construct the tableau $T_0 = (d_0, V_C^0, V_D^0, A_{CD}^0, A_{DC}^0, L^0)$:

```
(1) Let $d_0$ be an OR-node with label $\{spec\}$;
    $T_0 := d_0$;
    **repeat** until $frontier(T_0) = \emptyset$
    (a) Select a node $e \in frontier(T_0)$;
    (b) **if** $\exists e' \in V_D^0 : L(e) = L(e')$ **then**
            merge $e$ and $e'$
        **else**
            attach all $e' \in Succ(e)$ as successors of $e$ and mark $e$ as expanded
        **endif**;
        Update $V_C^0, V_D^0, A_{CD}^0, A_{DC}^0$ appropriately.
```

DeleteP. Delete any node whose label is propositionally inconsistent (i.e., the propositional component of the label is equivalent to *false*).

DeleteOR. Delete any OR-node all of whose successors are already deleted.

DeleteAND. Delete any AND-node one of whose successors (including fault-successors) is already deleted.[5]

*Delete*AU. Delete any node e such that $\mathsf{A}[g\mathsf{U}h] \in L(e)$ and there does not exist a fault-free full subdag rooted at e such that $h \in L(c')$ for every frontier node c' and $g \in L(c'')$ for every interior AND-node c''.

*Delete*EU. Delete any node e such that $\mathsf{E}[g\mathsf{U}h] \in L(e)$ and there does not exist an AND-node c' reachable from e via a fault-free path π such that $h \in L(c')$ and for all AND-nodes c'' along π up to but not necessarily including c', $g \in L(c'')$.

Fig. 2. The deletion rules for our synthesis method.

where the successors $Succ(e)$ of a node of either type are defined as follows: if e is an OR-node, then $Succ(e) = Blocks(d)$, and if e is an AND-node, then $Succ(e) = Tiles(e) \cup FaultStates(F, TOL, \{e\})$.
Let T_0 be the resulting tableau.

We now repeatedly apply the deletion rules in Figure 2 to T_0, until there is no change. These are similar to the CTL decision procedure [Emerson and Clarke 1982] rules, except that they require the existence of a *fault-free full subdag* to certify fulfillment of $\mathsf{A}[g\mathsf{U}h]$, and the existence of a *fault-free path* to certify fulfillment of $\mathsf{E}[g\mathsf{U}h]$.

(2) Repeatedly apply the deletion rules in Figure 2 to T_0 until no deletion rule is applicable. If d_0 is deleted, then return an impossibility result and halt. Otherwise, let T_F be the tableau induced by the nodes that are still reachable (via normal, fault, and recovery transitions) from d_0.

Upon termination, if d_0, the root of T_0, has been removed, then no program exists that satisfies $spec = init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$ under normal operation, and that has the required tolerance properties after a fault has occurred. In this case, we obtain an impossibility result. If d_0 is not removed, then we extract a model from T_F by a process of "unraveling."

For each AND-node c in T_F, a "fragment" FFRAG[c] is constructed. FFRAG[c] is a directed acyclic graph whose nodes are AND-nodes, and whose local structure is taken from T_F, that is, $c' \to c''$ in FFRAG[c] only if $c' \to d \to c''$ in T_F for some OR-node d. In addition, c is the root of FFRAG[c] and all eventualities in the label of c are fault-free-fulfilled in FFRAG[c]. Given that c was not removed by the deletion rules, it follows that, for each eventuality $g \in L(c)$, T_F contains a fault-free full subdag with root c and in which g is fulfilled. Let FDAG[c, g] be the directed acyclic prestructure that results from removing all the OR-nodes from this subdag, and connecting the AND-nodes up appropriately, that is, if $c \xrightarrow{i} d$ and $d \to c'$ are edges in the full subdag for some OR-node d, then $c \xrightarrow{i} c'$

[5]This is mainly where our deletion rules differ from those of the CTL decision procedure. The DeleteAU and DeleteEU rules are also modified to apply to fault-free subdags, fault-free paths, respectively.

is an edge in FDAG[c, g]. We construct FFRAG[c] as follows. Let $g_1, \ldots, g_m$ be all of the eventualities in $L(c)$.

```
(3) (a) Let FFRAG$_1$ be a copy of FDAG[$c$, $g_1$]. To obtain FFRAG$_{j+1}$ from
        FFRAG$_j$, do
        i. identify any two nodes on the frontier of FFRAG$_j$ that have the same
           label;
        ii. **forall** $s' \in frontier(\text{FFRAG}_j)$ **do**
              /* let $c'$ be the AND-node in $T_F$ that $s'$ is a copy of */
              **if** $g_{j+1} \in L(s')$ **then**
                 attach a copy of FDAG[$c'$, $g_{i+1}$] to FFRAG$_j$ at $s'$
        /* call the resulting directed acyclic graph FFRAG$_{j+1}$ */
    (b) Obtain FFRAG$'$[$c$] from FFRAG$_m$ by identifying any two nodes in
        $frontier(\text{FFRAG}_m)$ with the same label
    (c) To obtain FFRAG[$c$] from FFRAG$'$[$c$], do:
        i. **forall** AND-nodes $c'$ in FFRAG$'$[$c$] and $a \in F$:
              **forall** $d$ such that $c' \xrightarrow{a,TOL} d$
                 attach a copy of at least one node $c'' \in Blocks(d)$ as
                 successor of $c'$;[6]
                 label the transition from $c'$ to $c''$ as a fault-transition
```

If $L(c)$ contains no eventualities, then FFRAG$'$[c] consists of c together with enough local successors to satisfy all of the formulae that have AX_j or EX_j as main connective. That is, for each $d \in Tiles(c)$, choose a $c' \in Blocks(d)$ and add c' as a successor of c. Identify all such c' with the same label. Then FFRAG$'$[c] consists of c together with all such c'. Obtain FFRAG[c] from FFRAG$'$[c] by attaching fault-successors as in Step 3(c).

Note, in particular, that Step 3(c) adds the fault-successors of every AND-node in FFRAG[c] to its frontier. We prove in the sequel that FFRAG[c] is an acyclic prestructure generated by T_F whose root node s_0 is a copy of c, and that all eventualities in $Label(s_0)$ ($= Label(c)$) are fault-free-fulfilled for s_0 in FFRAG[c].

Finally, the FFRAGs are connected together (essentially, the frontier nodes of a FFRAG are identified with root nodes of other FFRAGs) to form a Kripke structure in which all eventualities are fault-free-fulfilled. This structure is a model of *spec*. The procedure is as follows:

```
(4) (a) Choose $c_0 \in Blocks(d_0)$ arbitrarily (recall that $d_0$ is the root of $T$);
        Let $M_1 = \text{FFRAG}[c_0]$;
    (b) To obtain $M_{i+1}$ from $M_i$, do
        i. **forall** $s \in frontier(M_i)$ **do**
              /* let $c$ be the AND-node in $T_F$ that $s$ is a copy of */
```

[6]It is permissible to attach a copy of more than one $c'' \in Blocks(d)$ as successor of c'. This increases the nondeterminism of the final model that is built, since execution of a fault-action a leads to one of several states nondeterministically, rather than to a single state that is "precomputed" statically by the model construction step. This extra nondeterminism may be appropriate, in certain situations.

```
        if there exists $s' \in interior(M_i)$ such that $s'$ is also a copy of $c$,
          and a copy of FFRAG[$c$] is directly embedded in $M_i$ with
          root $s'$,
        then
            identify $s$ and $s'$
        else
            replace $s$ by a copy of FFRAG[$c$]
        endif
    /* call the resulting graph $M_{i+1}$ */
```

(c) The construction halts with $i = N$ when $frontier(M_N)$ is empty. Let $M = M_N$. We write $M = (c_0, S, A, A_F, L)$, where c_0 is given in Step 4(a) (we write c_0 instead of $\{c_0\}$), L is given by the labels of each node, S is the set of all nodes in M_N, A is the set of all transitions in M_N that are labeled with a process index (i.e., normal or recovery transitions), and A_F is the set of all transitions in M_N that have label (a, TOL) for some $a \in F$ (i.e., the fault transitions).

Let $M_F = (c_0, S, A, A_F, L_0)$ where L_0 is L restricted to the propositions occurring in *spec*. M_F is a model of *spec*.

We show in the sequel that $M_F, s \models_n f$ for every $s \in S$ and $f \in L(s)$. In particular, since $spec \in L(c_0)$ by definition of c_0 and $Blocks(d_0)$, we have $M_F, c_0 \models_n spec$, and so the synthesized program satisfies *spec* under normal operation.

From the model M_F, we extract a program as follows. In going from M to M_F, we remove all of the nonpropositional formulae in the label of each node. This may result in several nodes (states) having the same label in M_F. As in Emerson and Clarke [1982], we introduce shared variables to distinguish such states. If this is not done, the extracted program will exhibit the same future behavior from all such states, even though they have different labels (in M), thus allowing pending eventualities to remain unfulfilled.[7]

Suppose that a set φ of propositional formulae occurs as the label of states $s_1, \ldots, s_n$ in M. We introduce a new shared variable x_φ, and add the proposition $x_\varphi = k$ to the label of s_k, for $k \in [1 : n]$. We also add the assignment $x_\varphi := k$ to the label of each transition in M_F that enters s_k, for $k \in [1 : n]$. Once all necessary shared variables have been added to M_F, we extract a program $P = P_1 \| \cdots \| P_I$ by projecting onto the individual process indices as follows: add an arc to (the synchronization skeleton) P_i going from local state s_i to local state t_i and labeled with the guarded command $B \rightarrow A$ iff there exists a transition M_F from state s to state t labeled with assignment A,[8] and such that $s_i = s{\uparrow}i$, $t_i = t{\uparrow}i$, and $B = \wedge(L(s){\downarrow}i)$. Here $s{\uparrow}i$ denotes the projection global state s onto P_i (i.e., the component of s that gives the local state of P_i), and $L(s){\downarrow}i$ denotes the set of all formulae in $L(s)$ of the form p_j or the form $\neg p_j$ (where p_j is an atomic proposition in $\mathcal{AP}_j$, and $j \in [1 : I] - \{i\}$) or the form $x = k$ (where x is a shared variable

[7] For example, not introducing a shared variable in the mutual exclusion example in Figure 8 gives rise to a cycle $[N_1\ T_2] \xrightarrow{1} [T_1\ T_2] \xrightarrow{1} [C_1\ T_2] \xrightarrow{1} [N_1\ T_2]$ that causes violation of the absence of starvation specification $\mathsf{AG}(T_2 \Rightarrow \mathsf{AF}C_2)$.

[8] If the transition is not labeled with an assignment, we take A to be *skip*.

and k is a natural number). $\wedge(L(s)\downarrow i)$ then denotes the conjunction of all formulae in the set $L(s)\downarrow i$. In other words, the guard B checks all of the components of global state s except for $s\uparrow i$. The pseudocode for the extraction step is as follows:

```
(5) (a) for every maximal set $\{s_1, \ldots, s_n\}$ of states in $M_F$ such that $L_0(s_1) = L_0(s_2) = \cdots = L_0(s_n)$
          i. introduce a new shared variable $x$
         ii. add the proposition $x = k$ to the label of $s_k$, $k \in [1 : n]$
        iii. label each transition of $M_F$ that enters $s_k$ with the assignment $x := k$,
             for $k \in [1 : n]$
    (b) forall transitions $s \xrightarrow{i,A} t$ in $M_F$
             add an arc to $P_i$ going from $s\uparrow i$ to $t\uparrow i$, and with the label $\wedge(L(s)\downarrow i)\ ar\ A$
```

Appendix 9.2 gives the entire pseudocode for our method.

5.3 Allowing Faults to Corrupt Shared Synchronization Variables

We have assumed up to now that fault actions cannot reference shared variables. Let x be one such variable, and let $\bar{t}$ be the set of propositionally identical states that x disambiguates, and let $|\bar{t}| = n$. Then, without loss of generality, we may assume that the domain of x is $[1 : n]$. By construction of our method, x is set to a constant upon entry to a state in $\bar{t}$ (to record which state in $\bar{t}$ is being entered), and is read only upon exit from states in $\bar{t}$ (to determine which state in $\bar{t}$ is in fact the current global state). Hence, the value of x in states outside $\bar{t}$ has no effect whatsoever on any future computation path.[9] Bearing this in mind, suppose the occurrence of some fault action f changes the current global state to some global state s. There are several cases to consider. If $s \notin \bar{t}$, then corrupting x has no effect, since the value of x in s doesn't affect any future computation. If $s \in \bar{t}$ and x is corrupted to some value in $[1 : n]$, then the effect is that of changing the final state from some $s' \in \bar{t}$ (which would otherwise have been entered) to s. Since s is an already present state from which recovery is guaranteed, this is also not a problem. If $s \in \bar{t}$ and x is corrupted to some value outside $[1 : n]$, then we simply interpret the new value of x as some "default" value within $[1 : n]$, for example, as 1.[10] Then the effect is as if x had been corrupted to 1, which is dealt with by the previous case.

Note that the above reasoning also deals with fault actions that corrupt several shared variables at once. Finally note that although we allow fault actions to corrupt (i.e., overwrite) shared variables, we do not allow fault actions to read shared variables. This restriction is needed (for technical reasons) to ensure the completeness of our method. In particular, this means that our method cannot deal with an adversary that chooses its strategy based on the value of the shared variables. Extending our method to deal with such adversaries is a topic for future work.

[9]In fact, the Emerson and Clarke [1982] synthesis method does not even record the value of x in states outside $\bar{t}$.

[10]We can do this since the domain of x is known in advance.

6. EXAMPLES

6.1 Mutual Exclusion Subject to Fail-Stop Failures

Our first example is the mutual exclusion problem subject to fail-stop failures. The mutual exclusion specification is given in Section 2.2. The fault specification is, for each process P_i, the auxiliary proposition D_i (denoting P_i is "down") and four fault actions: one that truthifies D_i and falsifies all other propositions of P_i (denoting the fail-stop of P_i), and three that truthify N_i, T_i, C_i (respectively) and falsify all other propositions of P_i (denoting the repair of P_i).[11] The problem-fault coupling specification is as follows, where $i \in \{1, 2\}$:

(1) A fail-stopped process is not in any of the states N_i, T_i or C_i:
$$\mathsf{AG}(D_i \equiv \neg(N_i \vee T_i \vee C_i)).$$
(2) A fail-stopped process may stay down forever:
$$\mathsf{AG}(D_i \Rightarrow \mathsf{EG} D_i).$$
(3) A transition by one process cannot cause a fault or recovery in another:
$$\mathsf{AG}((D_1 \Rightarrow \mathsf{AX}_2 D_1) \wedge (D_2 \Rightarrow \mathsf{AX}_1 D_2)).$$

Finally, the type of fault-tolerance we require is masking. The introduction of an auxiliary proposition D_i which is set to *true* when P_i is "down," implies an assumption of failure-detection [Chandra and Toueg 1996], since the other process can read D_i and thus detect that P_i is down.

We now illustrate the tableaux T_0 and T_F generated by our method for this problem. Throughout, AND-nodes are shown as rectangles, and OR-nodes as hexagons. Also, we include in the labels only as much information as needed to uniquely identify each node. So, for example, for all conjuncts of the form $\mathsf{AG} f$ that occur in the problem specification we assume implicitly that f appears in the label of all normal nodes, and for all conjuncts of the form $\mathsf{AG} f$ that occur in the coupling specification we assume implicitly that f appears in the label of all normal, perturbed, and recovery nodes. The fault-free portion of T_F was given by Figure 9 in Emerson and Clarke [1982]. Figure 3 in our paper shows some of the fault-transitions in T_0 and T_F. The top group are fault-transitions arising from fail-stop failures, and the bottom group are fault-transitions arising from repairs (i.e., when a process "comes back up"). Note that repairs are considered to be fault-transitions in our model. Figure 4 shows the portion of T_0, T_F that corresponds to the fail-stop of P_1 in local state N_1. Since no nodes are deleted, this portion is the same in T_0 and T_F. Figure 5 shows the portion of T_0 that corresponds to the fail-stop of P_1 followed by the fail-stop of P_2, from the initial state $[N_1\ N_2]$. Here, two of the OR-nodes are propositionally inconsistent, and are therefore deleted, by rule *DeleteP* of Figure 2 (that is, the OR-node with label $\{D_1, \mathsf{EG} D_1\}$ is propositionally inconsistent since the global specification requires that exactly one of the propositions N_2, T_2, C_2, D_2 be true in each node, and none of these propositions is true in this node). This leads to the deletion of three of the AND-nodes, by rule *DeleteAND* of Figure 2. Note that, for clarity

[11] C_i is truthified only when mutual exclusion would not be violated.

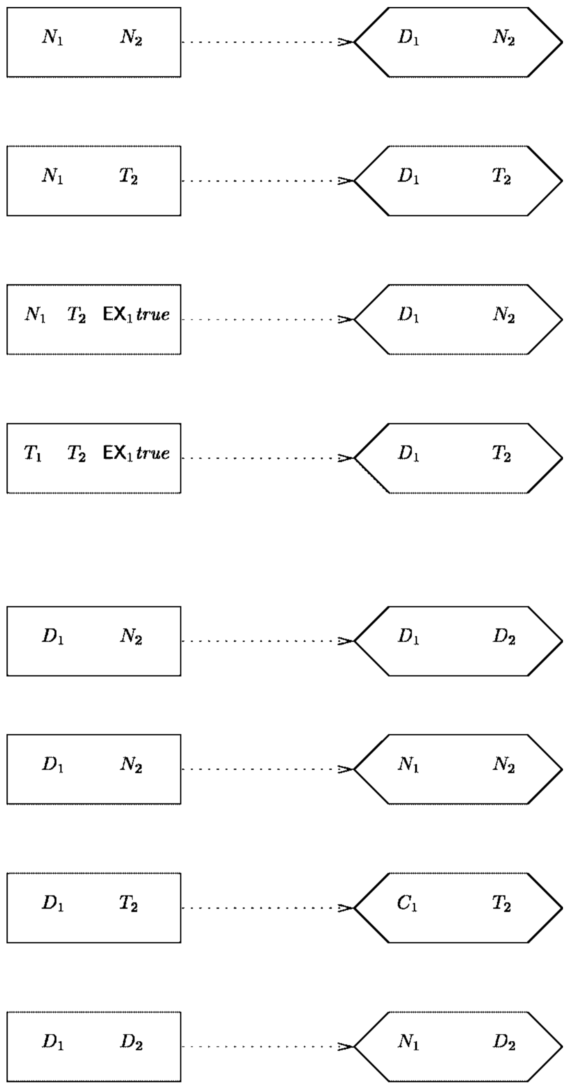

Fig. 3. Some fault-transitions in the tableaux T_0, T_F for the two-process mutual exclusion problem with fail-stop failures.

of Figure 5, the OR-AND transitions leaving the lower row of OR-nodes have all been omitted. Figure 6 shows the result of performing the deletions, that is, the portion of T_F that corresponds to the portion of T_0 in Figure 5. Figure 7 shows some of the FFRAGs that are extracted from the tableau portions in Figures 4 and 6. We have labeled some of the AND-nodes (OR-nodes) with $\mathtt{AND}_n$ ($\mathtt{OR}_n$) in these figures so as to illustrate which nodes are used in constructing the FFRAGs. Note that in Figure 4, the nodes labeled $\mathtt{OR}_1$ and $\mathtt{OR}_5$ are the same node, and are duplicated for clarity of the figure. Finally, in Figures 3–7, we have used solid, dashed, and dotted lines to denote those AND-OR and OR-AND

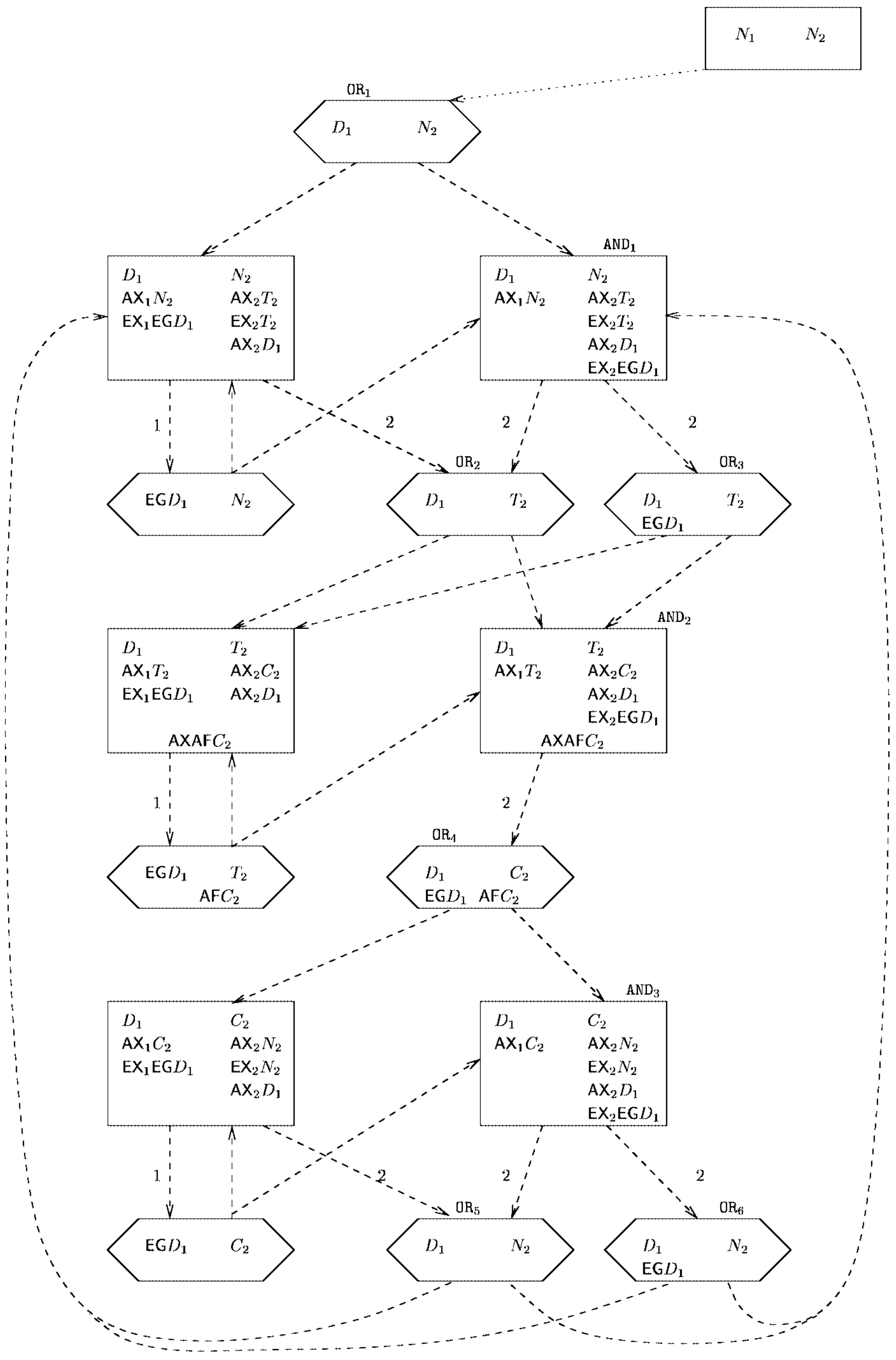
N_1 N_2
OR_1
D_1 N_2
D_1 N_2 $\mathsf{AX}_1 N_2$ $\mathsf{AX}_2 T_2$ $\mathsf{EX}_1\mathsf{EG}D_1$ $\mathsf{EX}_2 T_2$ $\mathsf{AX}_2 D_1$
AND_1
D_1 N_2 $\mathsf{AX}_1 N_2$ $\mathsf{AX}_2 T_2$ $\mathsf{EX}_2 T_2$ $\mathsf{AX}_2 D_1$ $\mathsf{EX}_2\mathsf{EG}D_1$
1
2
2
2
$\mathsf{EG}D_1$ N_2
OR_2
D_1 T_2
OR_3
D_1 T_2 $\mathsf{EG}D_1$
D_1 T_2 $\mathsf{AX}_1 T_2$ $\mathsf{AX}_2 C_2$ $\mathsf{EX}_1\mathsf{EG}D_1$ $\mathsf{AX}_2 D_1$ $\mathsf{AXAF}C_2$
AND_2
D_1 T_2 $\mathsf{AX}_1 T_2$ $\mathsf{AX}_2 C_2$ $\mathsf{AX}_2 D_1$ $\mathsf{EX}_2\mathsf{EG}D_1$ $\mathsf{AXAF}C_2$
1
2
$\mathsf{EG}D_1$ T_2 $\mathsf{AF}C_2$
OR_4
D_1 C_2 $\mathsf{EG}D_1$ $\mathsf{AF}C_2$
D_1 C_2 $\mathsf{AX}_1 C_2$ $\mathsf{AX}_2 N_2$ $\mathsf{EX}_1\mathsf{EG}D_1$ $\mathsf{EX}_2 N_2$ $\mathsf{AX}_2 D_1$
AND_3
D_1 C_2 $\mathsf{AX}_1 C_2$ $\mathsf{AX}_2 N_2$ $\mathsf{EX}_2 N_2$ $\mathsf{AX}_2 D_1$ $\mathsf{EX}_2\mathsf{EG}D_1$
1
2
2
2
$\mathsf{EG}D_1$ C_2
OR_5
D_1 N_2
OR_6
D_1 N_2 $\mathsf{EG}D_1$

Fig. 4. Portion of T_0, T_F corresponding to the fail-stop of P_1 in local state N_1.

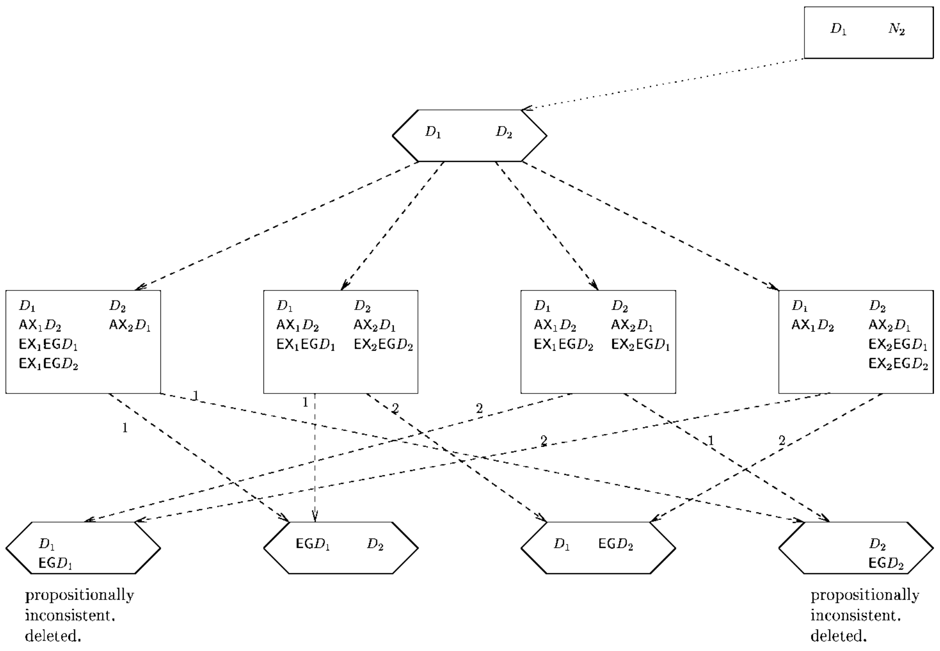
D_1 N_2
D_1 D_2
D_1 D_2 AX_1D_2 AX_2D_1 EX_1EGD_1 EX_1EGD_2
D_1 D_2 AX_1D_2 AX_2D_1 EX_1EGD_1 EX_2EGD_2
D_1 D_2 AX_1D_2 AX_2D_1 EX_1EGD_2 EX_2EGD_1
D_1 D_2 AX_1D_2 AX_2D_1 EX_2EGD_1 EX_2EGD_2
1
2
D_1 EGD_1
EGD_1 D_2
D_1 EGD_2
D_2 EGD_2
propositionally inconsistent. deleted.
propositionally inconsistent. deleted.

Fig. 5. Portion of T_0 corresponding to the fail-stop of P_1 followed by the fail-stop of P_2, from initial state $[N_1\ N_2]$.

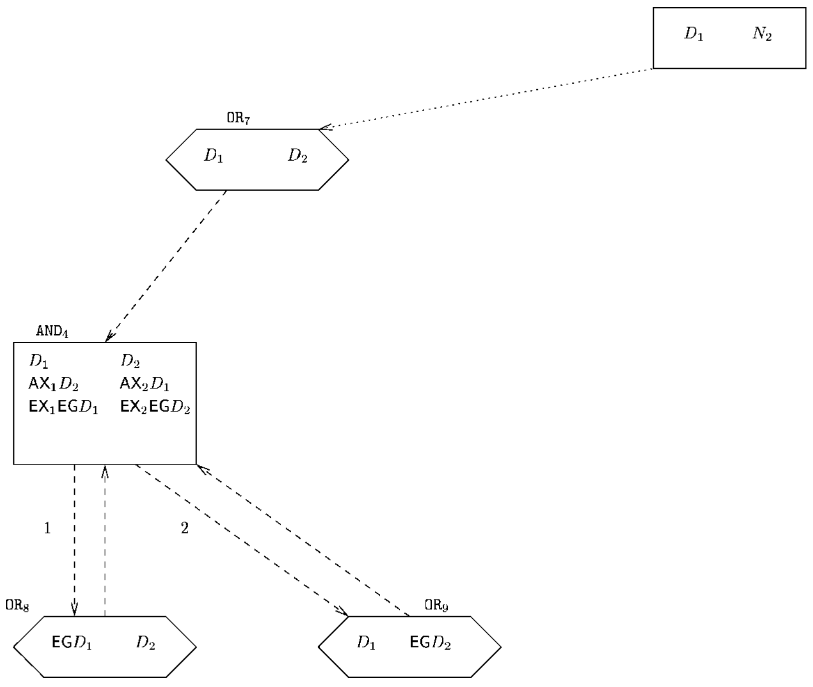

Fig. 6. Portion of T_F corresponding to the fail-stop of P_1 followed by the fail-stop of P_2, from initial state $[N_1\ N_2]$.

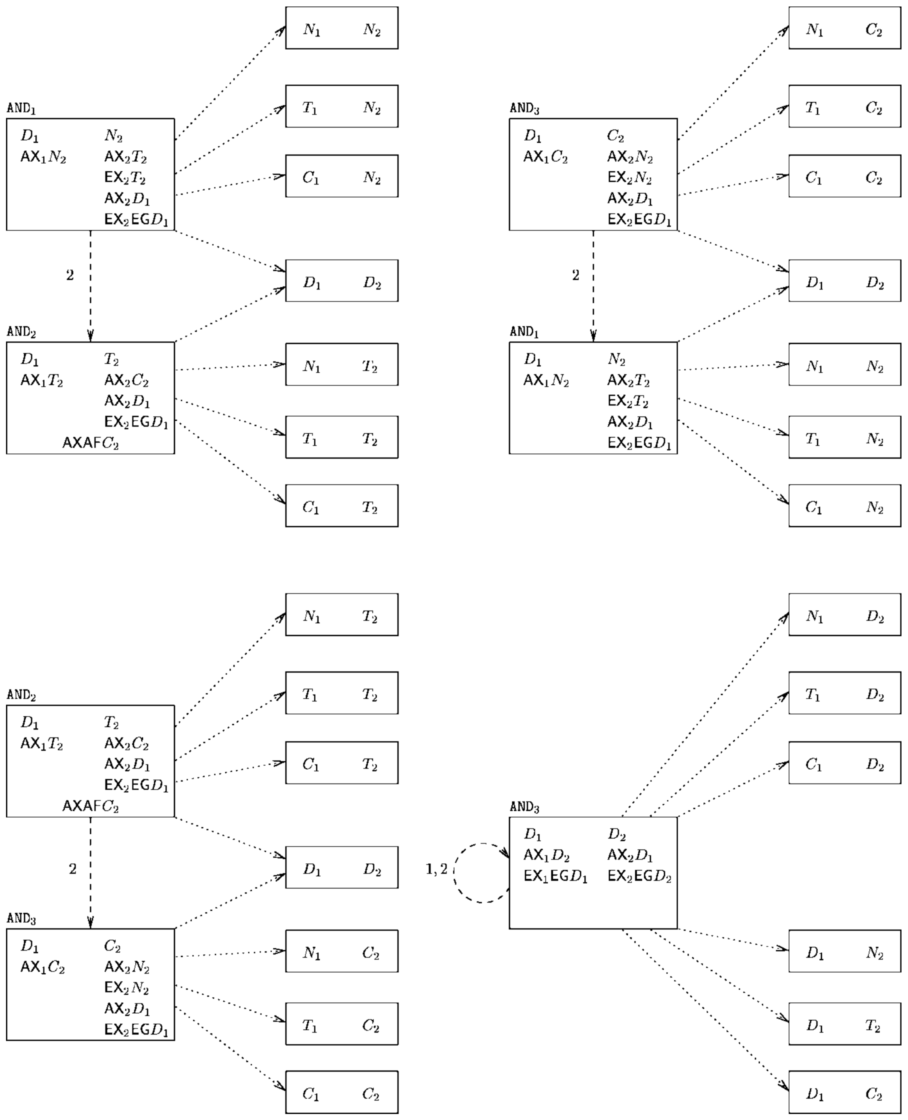

Fig. 7. Some FFRAGs for the two-process mutual exclusion problem with fail-stop failures.

transitions in the tableau that will give rise to normal-transitions, recovery-transitions, and fault-transitions in the final model (Figure 8), respectively.

Figure 8 shows the final model that is obtained by unwinding T_F, and Figure 9 shows the concurrent program $P_1 \| P_2$ that is extracted from this model. This program is a solution to the two-process mutual exclusion problem and exhibits masking tolerance to fail-stop failures.

The portion of the Kripke structure in Figure 8 that is above the dark horizontal line is the model for the mutual exclusion specification that is produced by the CTL decision procedure of Emerson and Clarke [1982] (and from which a fault-intolerant program could be extracted). The entire Kripke structure is

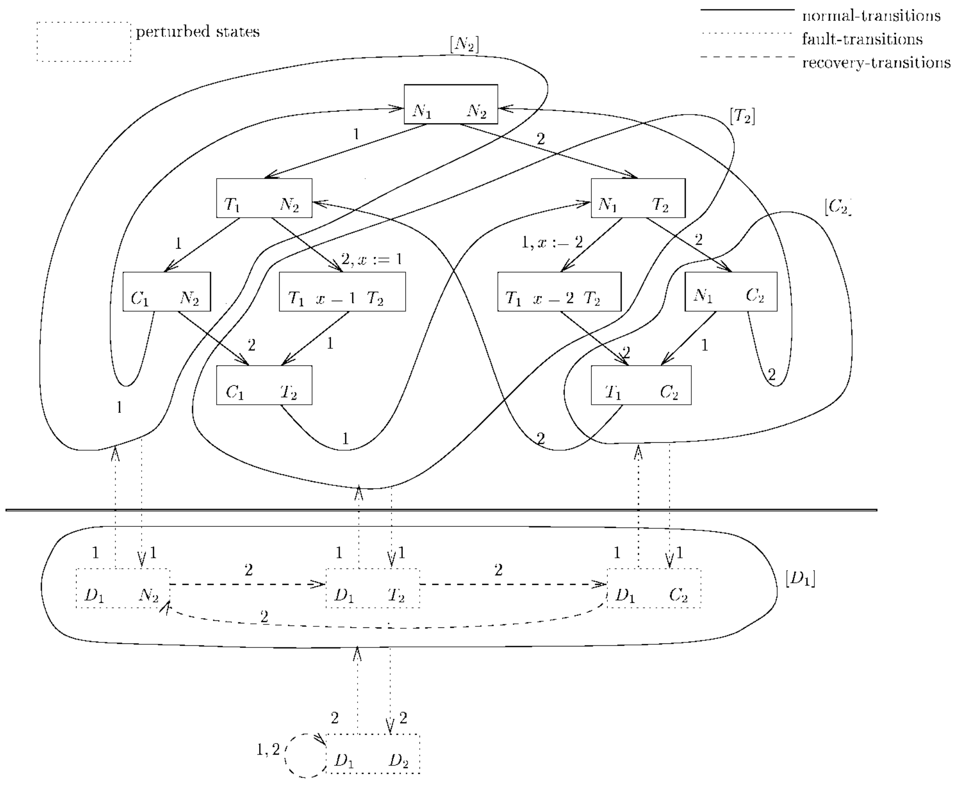

Fig. 8. Two-process mutual exclusion structure for the fail-stop failures model.

the final model produced by our synthesis method. For clarity, only the fault-/recovery-transitions corresponding to the failure of P_1, followed by the failure of P_2, are shown. The transitions corresponding to the other order of failure can be deduced by symmetry considerations. Normal-, fault-, and recovery-transitions are indicated by solid, dotted, and dashed lines, respectively. Perturbed states are indicated by a dotted boundary. Note the grouping of states into the sets $[N_2]$, $[T_2]$, $[C_2]$, $[D_1]$. All states in each set have the indicated fault- and recovery-transitions, which are drawn to the boundary of the set.

6.2 Barrier Synchronization Subject to General State Failures

Our second example is the barrier synchronization problem subject to general state failures. The barrier synchronization specification is as follows:

Each process consists of a cyclic sequence of two terminating phases, phase A and phase B. Process i, $i = 1, 2$, is in exactly one of 4 local states, SA_i, EA_i, SB_i, EB_i, corresponding to the start of phase A, the end of phase A, the start of phase B, and the end of phase B, respectively:

(1) Initial state (both processes are initially at the start of phase A): $SA_1 \wedge SA_2$.

(2) The start of phase A is always followed by the end of phase A:

$$\mathsf{AG}(SA_i \Rightarrow \mathsf{AX}_i EA_i).$$

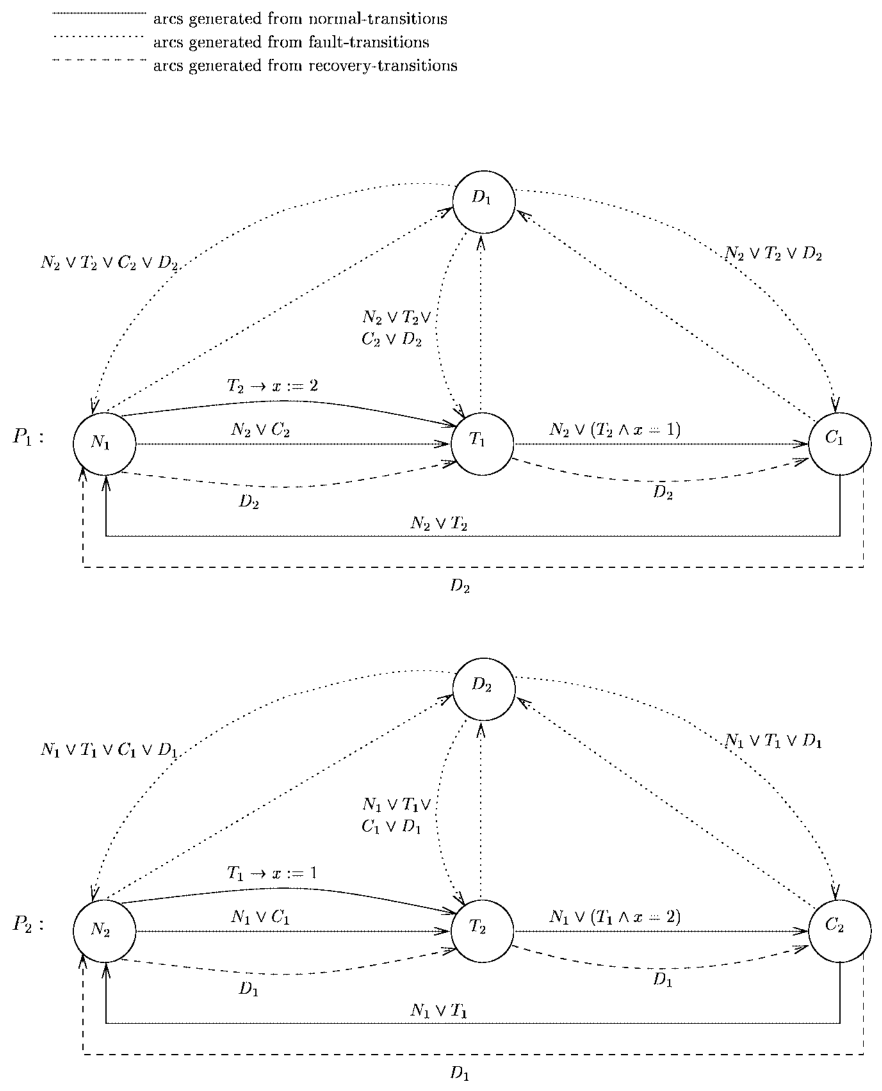
arcs generated from normal-transitions
arcs generated from fault-transitions
arcs generated from recovery-transitions
D_1
$N_2 \vee T_2 \vee C_2 \vee D_2$
$N_2 \vee T_2 \vee D_2$
$N_2 \vee T_2 \vee$ $C_2 \vee D_2$
$T_2 \rightarrow x := 2$
P_1 :
N_1
$N_2 \vee C_2$
T_1
$N_2 \vee (T_2 \wedge x = 1)$
C_1
D_2
D_2
$N_2 \vee T_2$
D_2
D_2
$N_1 \vee T_1 \vee C_1 \vee D_1$
$N_1 \vee T_1 \vee D_1$
$N_1 \vee T_1 \vee$ $C_1 \vee D_1$
$T_1 \rightarrow x := 1$
P_2 :
N_2
$N_1 \vee C_1$
T_2
$N_1 \vee (T_1 \wedge x = 2)$
C_2
D_1
D_1
$N_1 \vee T_1$
D_1

Fig. 9. Fault-tolerant concurrent program $P_1 \| P_2$ extracted from the structure in Figure 8.

(3) The end of phase A is always followed by the start of phase B:

$$\mathsf{AG}(EA_i \Rightarrow \mathsf{AX}_i SB_i).$$

(4) The start of phase B is always followed by the end of phase B:

$$\mathsf{AG}(SB_i \Rightarrow \mathsf{AX}_i EB_i).$$

(5) The end of phase B is always followed by the start of phase A:

$$\mathsf{AG}(EB_i \Rightarrow \mathsf{AX}_i SA_i).$$

(6) P_i is always in exactly one of the states SA_i, EA_i, SB_i, EB_i:

$$\mathsf{AG}(SA_i \equiv \neg(EA_i \vee SB_i \vee EB_i)) \wedge \mathsf{AG}(EA_i \equiv \neg(SA_i \vee SB_i \vee EB_i)) \wedge$$
$$\mathsf{AG}(SB_i \equiv \neg(SA_i \vee EA_i \vee EB_i)) \wedge \mathsf{AG}(EA_i \equiv \neg(SA_i \vee SB_i \vee EB_i)).$$

(7) The processes are never simultaneously at the start of different phases:

$$\mathsf{AG}\neg(SA_1 \wedge SB_2) \wedge \mathsf{AG}\neg(SA_2 \wedge SB_1).$$

(8) The processes are never simultaneously at the end of different phases:

$$\mathsf{AG}\neg(EA_1 \wedge EB_2) \wedge \mathsf{AG}\neg(EA_2 \wedge EB_1).$$

(9) It is always the case that some process can move: $\mathsf{AGEX}\,true$.

The fault specification adds no propositions, and for every combination of truth-values for the atomic propositions of a process, there exists a fault action assigning those values to the propositions. The problem-fault coupling specification is simply *true* (since general state failures are undetectable it is not useful to add extra propositions, and since they are also correctable, there is no need to add any restrictions on the propositions). Finally, the type of fault-tolerance we require is nonmasking.

Figure 10 gives the model generated by our synthesis method for this problem. (The model for the barrier synchronization specification that is produced by the CTL decision procedure is obtained by removing the four perturbed states and all incident transitions.) For clarity, the fault-transitions are omitted. Figure 11 shows the program extracted from the model. Note that the tolerance of the extracted program is a special case of nonmasking, namely, self-stabilizing [Arora and Gouda 1993].

It is interesting that in the fault-intolerant program for barrier synchronization (solid lines only), a process can move if the other process is at the same state or one state “ahead,” whereas in the fault-tolerant program (solid and dashed lines), a process can move if the other process is at the same state or one state ahead, or *two* states ahead. The fault-intolerant program deadlocks in any of the perturbed states, whereas the fault-tolerant program, with the recovery-transitions added, does not deadlock. But note, however, that these recovery-transitions do *not* permit the fault-tolerant program to generate any new states or transitions under normal (fault-free) operation.

6.3 An Impossibility Result

Consider also the barrier synchronization problem subject to fail-stop failures and with nonmasking tolerance required. Suppose P_1 goes down in state $[SA_1\ EA_2]$. If we allow that P_1 may stay down forever ($\mathsf{AG}(D_1 \Rightarrow \mathsf{EG}D_1)$ in the coupling specification), then the resulting perturbed state has a label that is unsatisfiable, and so recovery-transitions from this state cannot be generated. Indeed from the meaning of the barrier synchronization problem—the progress of P_2 requires the concomitant progress of P_1—it is easy to see that if P_1 stays down forever then the original problem specification cannot be satisfied. Hence our synthesis method provides a mechanical way of obtaining such impossibility results.

7. CORRECTNESS AND COMPLEXITY OF THE SYNTHESIS METHOD

There are three aspects to the correctness of the synthesis method: soundness, completeness, and fault-closure. Soundness means that the synthesized program satisfies the specification. Completeness means that, if there is a program

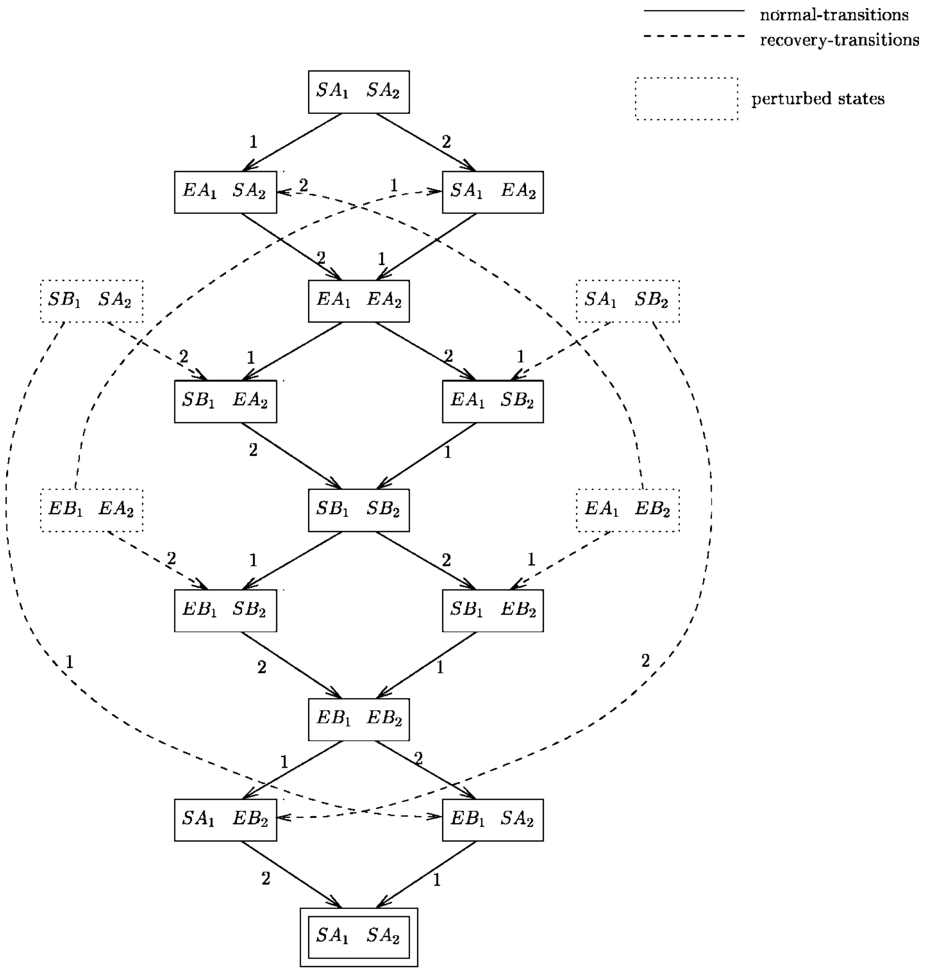

Fig. 10. Barrier synchronization structure for the general state failures model.

which satisfies the specification, then one such program will be synthesized. Fault-closure means that every specified fault-action is faithfully represented in the synthesized program.

7.1 Soundness

Recall that $\models_n$ denotes the $\models$ relation of CTL when path quantification is restricted to fault-free fullpaths, and that $M_F = (s_0, S, A, A_F, L_0)$[12] is the

[12]Since our method produces a single initial state, we use s_0 instead of $\{s_0\}$ to denote the corresponding singleton set.

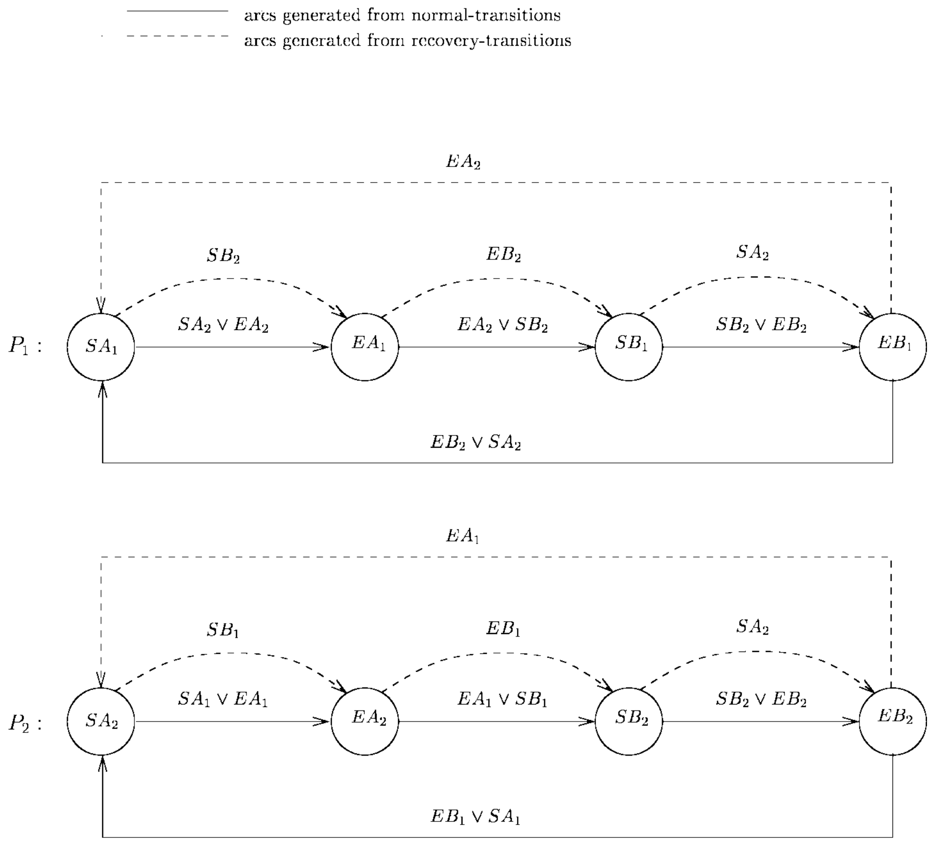

Fig. 11. Fault-tolerant concurrent program $P_1 \| P_2$ extracted from the structure in Figure 10.

fault-tolerant model produced by our method. Soundness means that all formulae in the label of a state hold in that state, that is, for all $s \in S$ and $g \in L(s)$: $M_F, s \models_n g$. Thus, by virtue of the way that labels are computed for nodes that correspond to perturbed states (Sections 4 and 5), the resulting program is a solution of the synthesis problem.

There are two key ideas in establishing soundness. The first is that all formulae in the label of a node are "propagated" correctly. For example, if $\mathsf{E}[g\mathsf{U}h]$ is in the label of a node v, then some successor of v must either contain h in its label (thereby fulfilling the eventuality $\mathsf{E}[g\mathsf{U}h]$), or it must contain g, $\mathsf{E}[g\mathsf{U}h]$, $\mathsf{EXE}[g\mathsf{U}h]$ in its label (thereby propagating the eventuality $\mathsf{E}[g\mathsf{U}h]$). The second key idea is that all eventualities are fulfilled, because every maximal path will eventually "intersect" with the root of a fragment. Fragment roots serve as "checkpoints," which ensure that all pending eventualities are fulfilled.

We start with Propositions 7.1.1–7.1.3 below, which establish useful structural properties of the tableau T_0. Proposition 7.1.1 follows from the definition of *Blocks*. *Blocks*(d) in effect contains all the successor nodes that correspond to different ways of simultaneously satisfying the formulae in $L(d)$. Proposition 7.1.2 follows from the fact that, for any $c \in$ *Blocks*(d), the conjunction of the elementary formulae in $L(c)$ is logically equivalent to the conjunction of all the formulae in $L(c)$. It is easily seen that this equivalence is preserved by each of the "α-β" expansions (Section 4) which construct a child from its parent in the tree that is constructed during the computation of *Blocks*. Proposition 7.1.3 follows from the definition of *Tiles*, in that *Tiles*(c) is the minimum set of successors that are needed in order to satisfy the nexttime ($\mathsf{AX}g$ or $\mathsf{EX}h$) formulae in $L(c)$. By construction, $L(c)$ contains only elementary formulae, and so the only other formulae in $L(c)$ besides the nexttime formulae are either atomic propositions or negations of atomic propositions. See Section 4 for the detailed discussion of *Blocks* and *Tiles*. The full proofs of Propositions 7.1.1–7.1.3 can be found in Emerson [1981, chapter 4], where these are given as Propositions 4.4.1–4.4.3, respectively.[13]

We next establish Proposition 7.1.4, which gives useful structural properties that all nodes (including the perturbed nodes) in tableau T_0 have, for example, that nodes have unique labels, no nodes are without successors, every AND-node has a downward-closed label, and the relationships between satisfiability of a node label and the satisfiability of the labels of its successor nodes are as given in Propositions 7.1.1 and 7.1.3. Most of these follow from the construction of T_0, and by Propositions 7.1.1 and 7.1.3. Clauses 7.1.4 and 7.1.4 are noteworthy in that they establish that the nexttime formulae in the label of a node are propagated appropriately to the successors of the node. This is a crucial first step for showing that all the formulae in a node label actually hold in the final model that is constructed. Proposition 7.1.4 lays the groundwork for Proposition 7.1.5, which establishes similar structural properties for any prestructure that is generated by tableau T_0. Specifically, it shows that the nexttime formulae are propagated appropriately. Thus, nexttime formulae are "satisfied locally" in such a prestructure.

In Proposition 7.1.6, we show that all CTL formulae are propagated appropriately. For example, if $\mathsf{E}[g\mathsf{U}h]$ is in the label of some node v_0, then there must be some maximal fault-free path starting from v_0 such that g, $\mathsf{E}[g\mathsf{U}h]$, $\mathsf{EXE}[g\mathsf{U}h]$ are all propagated, that is, they are in the labels of successive nodes along the path, until a node is reached containing h in its label. If no such node exists, then g, $\mathsf{E}[g\mathsf{U}h]$, $\mathsf{EXE}[g\mathsf{U}h]$ are propagated forever. The propagation is enforced by the presence of $\mathsf{EXE}[g\mathsf{U}h]$ in the node labels, since Proposition 7.1.5 shows that nexttime formulae are propagated appropriately. This would be sufficient to establish that all formulae in the label of a node are true, except for the issue of eventualities. So, in the above example of $\mathsf{E}[g\mathsf{U}h]$, it is possible that every node along the maximal fault-free path is labeled with g, and no node is labeled with h. $\mathsf{E}[g\mathsf{U}h]$ requires that h is actually true at some node along the path. We establish that this must hold by showing that the eventualities in the root

[13]The results in Emerson [1981, chapter 4] were established for the logic UB, which was obtained from CTL by replacing AU, EU, AW, EW by AF, EF, AG, EG, respectively. However, the results can be easily seen to carry over to CTL: to deal with $\mathsf{A}[f\mathsf{U}g]$, $\mathsf{E}[f\mathsf{U}g]$, modify the treatment of $\mathsf{AF}g$, $\mathsf{EF}g$, respectively, to check that f holds along the prefix of the fullpath up to the state where g holds. Likewise, to deal with $\mathsf{A}[f\mathsf{W}g]$, $\mathsf{E}[f\mathsf{W}g]$, modify the treatment of $\mathsf{AG}g$, $\mathsf{EG}g$, respectively, to check that g holds only in all states up to and including the first state in which f holds (rather than in all states of the path).

c of a fragment FFRAG[c] are all fulfilled *within* FFRAG[c] (Proposition 7.1.7). Thus, the root c of a fragment serves as a "checkpoint," which guarantees that all eventualities that are pending in c are actually fulfilled. Since the final model is constructed by pasting together fragments (Step 4 in Section 5.2), we can show that every maximal fault-free path must contain a node which is the root of a fragment. This node serves to certify the fulfillment of all eventualities that are pending in the first state of the path. We do this in Theorem 7.1.9, which shows that, in the final model M, a node satisfies all the formulae in its label.

If φ is a set of propositional formulae, then the notation $\dashv\!|\,\varphi$ means that all the formulae in φ are simultaneously satisfiable, that is, there exists an assignment of truth values to the atomic propositions in these formulae which makes all of the formulae true.

PROPOSITION 7.1.1. *Let d be an OR-node. Then $\dashv\!|\,L(d)$ iff $\dashv\!|\,L(c_1)$ or $\cdots$ or $\dashv\!|\,L(c_k)$, where $Blocks(d) = \{c_1, \ldots, c_k\}$.*

PROPOSITION 7.1.2. *Let d be an OR-node. Then, for each $c_i \in Blocks(d)$, $\dashv\!|\,L(c_i)$ iff $\dashv\!|\,LE(c_i)$, where $LE(c_i) = \{f \in L(c_i) \mid f$ is elementary$\}$.*

PROPOSITION 7.1.3. *Let c be an AND-node. Then $\dashv\!|\,L(c)$ iff $\dashv\!|\,L(d_1)$ and $\cdots$ and $\dashv\!|\,L(d_k)$ and $\dashv\!|\,LP(c)$ where $Tiles(c) = \{d_1, \ldots, d_k\}$ and $LP(c) = \{f \in L(c) \mid f$ is an atomic proposition or its negation$\}$.*

PROPOSITION 7.1.4. *Tableau T_0 satisfies the following:*

(1) *The root of T_0 is an OR-node d_0 such that $L(d_0) = \{f_0\}$.*
(2) *All AND-nodes (OR-nodes) in T_0 have distinct labels.*
(3) *Every node in T_0 has a successor in T_0.*
(4) *$L(c)$ is downward-closed for all AND-nodes c in T_0.*
(5) *For every OR-node d, $L(d)$ iff $\dashv\!|\,L(c_1)$ or $\cdots$ or $\dashv\!|\,L(c_n)$, where $Blocks(d) = \{c_1, \ldots, c_n\}$.*
(6) *For every AND-node c, $\dashv\!|\,L(c)$ iff $\dashv\!|\,L(d_1)$ and $\cdots$ and $\dashv\!|\,L(d_m)$ and $\dashv\!|\,LP(c)$ where $Tiles(c) = \{d_1, \ldots, d_m\}$.*
(7) *For every AND-node c,*

$$\begin{aligned}\mathsf{AX} f &\in L(c) \text{ implies } f \in L(d) \text{ for all } d \in Tiles(c),\\ \mathsf{EX} f &\in L(c) \text{ implies } f \in L(d) \text{ for some } d \in Tiles(c).\end{aligned}$$

(8) *For every AND-node c,*

$$\begin{aligned}\mathsf{AX}_i f &\in L(c) \text{ implies } f \in L(d) \text{ for all } d \in Tiles_i(c),\\ \mathsf{EX}_i f &\in L(c) \text{ implies } f \in L(d) \text{ for some } d \in Tiles_i(c).\end{aligned}$$

PROOF. We deal with each clause of the proposition in turn.

Clause 1. By construction of our method.

Clause 2. Nodes with identical labels are merged in Step 1b, Section 5.2.

Clause 3. By definition, $Succ(e)$ is never empty for any node e. Since every node is expanded at some point, it follows that every node has at least one successor.

Clause 4. Any AND-node c is a member of $Blocks(d)$ for some OR-node d, by construction of our method. By definition of $Blocks$, any node in $Blocks(d)$ has a downward-closed label. (Note that the nodes directly generated by applying a fault action are OR-nodes.)

Clause 5. Follows directly from Proposition 7.1.1.

Clause 6. Follows directly from Proposition 7.1.3.

Clause 7. Holds by definition of $Tiles(c)$.

Clause 8. Holds by definition of $Tiles_i(c)$. □

PROPOSITION 7.1.5. *If prestructure $G = (V, A, L)$ is generated by tableau T_0, then*

(1) *for all nodes v in G, $L(v)$ is downward closed;*
(2) *for all interior nodes v in G, if $\mathsf{AX} f \in L(v)$, then $f \in L(v')$ for all non-fault-successors v' of v in G;*
(3) *for all interior nodes v in G, if $\mathsf{AX}_i f \in L(v)$, then $f \in L(v')$ for all (non-fault) P_i-successors v' of v in G;*
(4) *for all interior nodes v in G, if $\mathsf{EX} f \in L(v)$, then $f \in L(v')$ for some non-fault-successor v' of v in G;*
(5) *for all interior nodes v in G, if $\mathsf{EX}_i f \in L(v)$, then $f \in L(v')$ for some (non-fault) P_i-successor v' of v in G.*

PROOF. Recall that tableau $T_0 = (d_0, V_C^0, V_D^0, A_{CD}^0, A_{DC}^0, L^0)$, let $K = (V_C', V_D', A_{CD}', A_{DC}', L')$ be the fullgraph satisfying the definition of "G is generated by T_0," and let E be the generation function satisfying the definition of "K is generated by T_0." We establish each clause in turn.

Clause 1. Let v be an arbitrary node of G. By definition of generated, $L(v)$ is the label of some AND-node of T_0. Hence, by Proposition 7.1.4, clause 7.1.4, $L(v)$ is downward-closed.

Clause 2. Let v be an arbitrary interior node of G, and let v' be an arbitrary non-fault-successor of v in G. By definition of generated, there exists $w \in V_D'$ such that $(v, w) \in A_{CD}'$ and $(w, v') \in A_{DC}'$. By definition of generated, $(E(v), E(w)) \in A_{CD}$ and $(E(w), E(v')) \in A_{DC}$. By the fact that v' is a non-fault-successor of v, we have $E(w) \in Tiles(E(v))$ and $E(v') \in Blocks(E(w))$. Now suppose $\mathsf{AX} f \in L(v)$. Then, $\mathsf{AX} f \in L^0(E(v))$. Hence, by Proposition 7.1.4, Clause 7, $f \in L^0(E(w))$. Thus, by construction of $Blocks(E(w))$, $f \in L^0(E(v'))$. By definition of generated, we conclude $f \in L(v')$.

Clause 3. Let v be an arbitrary interior node of G, and let v' be an arbitrary non-fault P_i-successor of v in G. By definition of generated, there exists $w \in$

V'_D such that $(v, i, w) \in A'_{CD}$ and $(w, v') \in A'_{DC}$. By definition of generated, $(E(v), i, E(w)) \in A_{CD}$ and $(E(w), E(v')) \in A_{DC}$. By the fact that v' is a non-fault P_i-successor of v, we have $E(w) \in Tiles_i(E(v))$ and $E(v') \in Blocks(E(w))$. Now suppose $\mathsf{AX}_i\, f \in L(v)$. Then, $\mathsf{AX}_i\, f \in L^0(E(v))$. Hence, by Proposition 7.1.4, Clause 8, $f \in L^0(E(w))$. Thus, by construction of $Blocks(E(w))$, $f \in L^0(E(v'))$. By definition of generated, we conclude $f \in L(v')$.

Clause 4. Let v be an arbitrary interior node of G. By definition of generated, v is an interior node of K. Hence $E(v)$ is an interior node of T_0. Now suppose $\mathsf{EX}\, f \in L(v)$. Then, $\mathsf{EX}\, f \in L^0(E(v))$. By Proposition 7.1.4, Clause 7, $f \in L^0(d)$ for some OR-node successor d of $E(v)$ in T_0. By definition of generated, there exists some OR-node w of K such that $(v, w) \in A'_{CD}$, $E(w) = d$, and w has some successor in K. Let v' be some non-fault AND-node successor of w in K, that is, $(w, v') \in A'_{DC}$. By definition of generated, $(E(w), E(v')) \in A^0_{DC}$, that is, $(d, E(v')) \in A^0_{DC}$. Since $f \in L^0(d)$ and $E(v') \in Blocks(d)$, we have $f \in L^0(E(v'))$ by construction of $Blocks$. Hence $f \in L(v')$. From $(v, w) \in A'_{CD}$, $(w, v') \in A'_{DC}$, and the definition of generated, (v, v') is a non-fault edge in G. Since $f \in L(v')$, we are done.

Clause 5. Let v be an arbitrary interior node of G. By definition of generated, v is an interior node of K. Hence $E(v)$ is an interior node of T_0. Now suppose $\mathsf{EX}_i\, f \in L(v)$. Then, $\mathsf{EX}_i\, f \in L^0(E(v))$. By Proposition 7.1.4, Clause 8, $f \in L^0(d)$ for some OR-node P_i-successor d of $E(v)$ in T_0. By definition of generated, there exists some OR-node w of K such that $(v, i, w) \in A'_{CD}$, $E(w) = d$, and w has some successor in K. Let v' be some AND-node successor of w in K, that is, $(w, v') \in A'_{DC}$. By definition of generated, $(E(w), E(v')) \in A^0_{DC}$, that is, $(d, E(v')) \in A^0_{DC}$. Since $f \in L^0(d)$ and $E(v') \in Blocks(d)$, we have $f \in L^0(E(v'))$ by construction of $Blocks$. Hence $f \in L(v')$. From $(v, i, w) \in A'_{CD}$, $(w, v') \in A'_{DC}$, and the definition of generated, (v, i, v') is a non-fault edge in G. Since $f \in L(v')$, we are done. □

PROPOSITION 7.1.6. *Let G be a prestructure generated by tableau T_0. For all nodes v_0 in G, the following all hold:*

(1) *If $\mathsf{A}[g \mathsf{U} h] \in L(v_0)$, then for all maximal fault-free paths $v_0, v_1, v_2, \ldots$ in G,*
for all $j \geq 0$, $g, \mathsf{A}[g \mathsf{U} h], \mathsf{AXA}[g \mathsf{U} h] \in L(v_j)$, or
there exists $i \geq 0$ such that
$h, \mathsf{A}[g \mathsf{U} h] \in L(v_i)$ and for all $j \in [0 : i)$, $g, \mathsf{A}[g \mathsf{U} h], \mathsf{AXA}[g \mathsf{U} h] \in L(v_j)$.

(2) *If $\mathsf{E}[g \mathsf{U} h] \in L(v_0)$, then for some maximal fault-free path $v_0, v_1, v_2, \ldots$ in G,*
for all $j \geq 0$, $g, \mathsf{E}[g \mathsf{U} h], \mathsf{EXE}[g \mathsf{U} h] \in L(v_j)$, or
there exists $i \geq 0$ such that
$h, \mathsf{E}[g \mathsf{U} h] \in L(v_i)$ and for all $j \in [0 : i)$, $g, \mathsf{E}[g \mathsf{U} h], \mathsf{EXE}[g \mathsf{U} h] \in L(v_j)$.

(3) *If $\mathsf{A}[g \mathsf{W} h] \in L(v_0)$, then for all maximal fault-free paths $v_0, v_1, v_2, \ldots$ in G,*
for all $j \geq 0$, $h, \mathsf{A}[g \mathsf{W} h], \mathsf{AXA}[g \mathsf{W} h] \in L(v_j)$, or
there exists $i \geq 0$ such that
$g, h, \mathsf{A}[g \mathsf{W} h] \in L(v_i)$ and for all $j \in [0 : i)$, $h, \mathsf{A}[g \mathsf{W} h], \mathsf{AXA}[g \mathsf{W} h] \in L(v_j)$.

(4) *If* $\mathsf{E}[g\mathsf{W}h] \in L(v_0)$*, then for some maximal fault-free path* $v_0, v_1, v_2, \ldots$ *in* G*,*
for all $j \geq 0$*,* $h, \mathsf{E}[g\mathsf{W}h], \mathsf{EXE}[g\mathsf{W}h] \in L(v_j)$*, or*
there exists $i \geq 0$ *such that*
$g, h, \mathsf{E}[g\mathsf{W}h] \in L(v_i)$ *and for all* $j \in [0 : i)$*,* $h, \mathsf{E}[g\mathsf{W}h], \mathsf{EXE}[g\mathsf{W}h] \in L(v_j)$*.*

PROOF. The proof is similar to the proof of Proposition 4.5.3 in Emerson [1981, chapter 4] except that Proposition 7.1.5 is invoked instead of Proposition 4.5.2 (of [Emerson 1981, chapter 4]), and the straightforward adjustments need to be made for the modalities AU, EU, AW, EW (as noted previously, Emerson [1981, chapter 4] dealt only with the modalities AF, EF, AG, EG):

Clause 1. Suppose $\mathsf{A}[g\mathsf{U}h] \in L(v_0)$ and $v_0, v_1, v_2, \ldots$ is a maximal fault-free path in G. By Proposition 7.1.5, Clause 1 and the definition of downward-closed, we see that, for each v_j, if $\mathsf{A}[g\mathsf{U}h] \in L(v_j)$ and $h \notin L(v_j)$, then $g, \mathsf{AXA}[g\mathsf{U}h] \in L(v_j)$. Furthermore, unless v_j is the last node on $v_0, v_1, v_2, \ldots$, $\mathsf{A}[g\mathsf{U}h] \in L(v_{j+1})$, by Proposition 7.1.5, Clause 2. Now either $h \notin L(v_i)$ for any i or there is a least i such that $h \in L(v_i)$. In the former case, we have $g, \mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h] \in L(v_j)$ for all $j \geq 0$. In the latter case, we have, for some i, $h, \mathsf{A}[g\mathsf{U}h] \in L(v_i)$ and for all $j \in [0 : i)$, $g, \mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h] \in L(v_j)$.

Clause 2. Suppose $\mathsf{E}[g\mathsf{U}h] \in L(v_0)$. By Proposition 7.1.5, Clause 1, and the definition of downward-closed, we have: (1) for each node v of G: if $\mathsf{E}[g\mathsf{U}h] \in L(v)$ and $h \notin L(v)$, then $g \in L(v)$ and $\mathsf{EXE}[g\mathsf{U}h] \in L(v)$. Also by Proposition 7.1.5, Clause 4, we have: (2) for each internal node v of G: if $\mathsf{EXE}[g\mathsf{U}h] \in L(v)$, then $\mathsf{E}[g\mathsf{U}h] \in L(v')$ for some non-fault-successor v' of v.

Now if $h \in L(v_0)$ then we are done. Otherwise, $h \notin L(v_0)$, and so, by (1), $g \in L(v_0)$ and $\mathsf{EXE}[g\mathsf{U}h] \in L(v_0)$. If v_0 is a frontier node, then we are done. Hence we have: (3) if $h \in L(v_0)$ or v_0 is a frontier node then we are done. If v_0 is not a frontier node, then it is internal, and so by (2) and $\mathsf{EXE}[g\mathsf{U}h] \in L(v_0)$, v_0 has some non-fault-successor v_1 such that $\mathsf{E}[g\mathsf{U}h] \in L(v_1)$. Now, using the same reasoning as in proving (3), we have: if $h \in L(v_1)$ or v_1 is a frontier node then we are done. Otherwise, $h \notin L(v_1)$ and v_1 is internal, and so by (1), $g \in L(v_1)$ and $\mathsf{EXE}[g\mathsf{U}h] \in L(v_1)$. Hence, by (2), $\mathsf{E}[g\mathsf{U}h] \in L(v_2)$ for some non-fault-successor v_2 of v_1. We can continue to generate non-fault-successors in this way until we reach a node v_i such that $h \in L(v_i)$ or v_i is a frontier node. In the first case, we also have $\mathsf{E}[g\mathsf{U}h] \in L(v_i)$, and for all $j \in [0 : i)$, $g, \mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h] \in L(v_j)$, and so we are done. In the second case, we have for all $j \geq 0$, $g, \mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h] \in L(v_j)$, since the path $v_0, v_1, v_2, \ldots$ is maximal. Hence we are done, since $v_0, v_1, v_2, \ldots$ is also fault-free.

Clause 3. Suppose $\mathsf{A}[g\mathsf{W}h] \in L(v_0)$ and $v_0, v_1, v_2, \ldots$ is a maximal fault-free path in G. By Proposition 7.1.5, Clause 1, and the definition of downward-closed, we see that, for each v_j, if $\mathsf{A}[g\mathsf{W}h] \in L(v_j)$, then (1) $h \in L(v_j)$, and (2) if $g \notin L(v_j)$, then $\mathsf{AXA}[g\mathsf{W}h] \in L(v_j)$. Furthermore, unless v_j is the last node on $v_0, v_1, v_2, \ldots$, $\mathsf{A}[g\mathsf{W}h] \in L(v_{j+1})$, by Proposition 7.1.5, Clause 2. Now either $g \notin L(v_i)$ for any i or there is a least i such that $g \in L(v_i)$. In the former case, we have $h, \mathsf{A}[g\mathsf{W}h], \mathsf{AXA}[g\mathsf{W}h] \in L(v_j)$ for all $j \geq 0$. In the latter case, we have, for some i, h, $g, \mathsf{A}[g\mathsf{W}h] \in L(v_i)$ and for all $j \in [0 : i)$, $h, \mathsf{A}[g\mathsf{W}h], \mathsf{AXA}[g\mathsf{W}h] \in L(v_j)$.

Clause 4. Suppose $\mathsf{E}[g\mathsf{W}h] \in L(v_0)$. By Proposition 7.1.5, Clause 1, and the definition of downward-closed, we have (1) for each node v of G: if $\mathsf{E}[g\mathsf{W}h] \in L(v_j)$, then $h \in L(v_j)$, and ($g \notin L(v_j)$ implies $\mathsf{EXE}[g\mathsf{W}h] \in L(v_j)$). Also by Proposition 7.1.5, Clause 4, we have: (2) for each internal node v of G: if $\mathsf{EXE}[g\mathsf{W}h] \in L(v)$, then $\mathsf{E}[g\mathsf{W}h] \in L(v')$ for some non-fault-successor v' of v.

Now if $g \in L(v_0)$ then we are done. Otherwise, $g \notin L(v_0)$, and so, by (1), $h \in L(v_0)$ and $\mathsf{EXE}[g\mathsf{W}h] \in L(v_0)$. If v_0 is a frontier node, then we are done. Hence we have: (3) if $g \in L(v_0)$ or v_0 is a frontier node then we are done. If v_0 is not a frontier node, then it is internal, and so by (2) and $\mathsf{EXE}[g\mathsf{W}h] \in L(v_0)$, v_0 has some non-fault-successor v_1 such that $\mathsf{E}[g\mathsf{W}h] \in L(v_1)$. Now, using the same reasoning as in proving (3), we have: if $g \in L(v_1)$ or v_1 is a frontier node then we are done. Otherwise, $g \notin L(v_1)$ and v_1 is internal, and so by (1), $h \in L(v_1)$ and $\mathsf{EXE}[g\mathsf{W}h] \in L(v_1)$. Hence, by (2), $\mathsf{E}[g\mathsf{W}h] \in L(v_2)$ for some non-fault-successor v_2 of v_1. We can continue to generate non-fault-successors in this way until we reach a node v_i such that $g \in L(v_i)$ or v_i is a frontier node. In the first case, we also have $h, \mathsf{E}[g\mathsf{W}h] \in L(v_i)$, and for all $j \in [0 : i)$, $h, \mathsf{E}[g\mathsf{W}h], \mathsf{EXE}[g\mathsf{W}h] \in L(v_j)$, and so we are done. In the second case, we have for all $j \geq 0$, $h, \mathsf{E}[g\mathsf{W}h], \mathsf{EXE}[g\mathsf{W}h] \in L(v_j)$, since the path $v_0, v_1, v_2, \ldots$ is maximal. Hence we are done, since $v_0, v_1, v_2, \ldots$ is also fault-free. □

PROPOSITION 7.1.7. *For every AND-node c in T_F*, FFRAG[c] *satisfies the following:*

(1) *The fault-free portion of* FFRAG[c] *is an acyclic prestructure generated by T_F whose root node s_0 is a copy of c.*
(2) *All eventuality formulae in $L(s_0)$ are fulfilled for s_0 in the fault-free portion of* FFRAG[c].

PROOF. By construction, the fault-free portion of FFRAG[c] is just FRAG[c]. The proof is then verbatim identical to the proof of Proposition 4.8.1 in Emerson [1981, chapter 4] except that the propositions established above are invoked instead of their analogues in Emerson [1981, chapter 4].

Let FDAG[s, g] denote a copy of FDAG[c, g], where s is a copy of AND-node c. Referring to Step 3 in Section 5.2, we see that the fault-free portion of FFRAG[c] is FFRAG$'$[c]. Also, FFRAG$'$[c] is obtained from FFRAG$_m$ by identifying any two nodes in *frontier*(FFRAG$_m$) with the same label. Thus, if the proposition holds for FFRAG$_m$, then it must also hold for FFRAG$'$[c]. It is left to establish the proposition for FFRAG$_m$. Let $g_1, \ldots, g_m$ be all of the eventualities in $L(c)$.

Referring to Step 3, we establish, by induction on the loop variable j, the following:

(1) The fault-free portion of FFRAG$_j$ is an acyclic prestructure generated by T_F whose root node s_0 is a copy of c.
(2) The eventuality formulae $g_1, \ldots, g_j$ are fulfilled for s_0 in the fault-free portion of FFRAG$_j$

The induction hypothesis holds for FFRAG$_1$ = FDAG[s_0, g_1] by definition of fault-free full-subdag. We assume the induction hypothesis for $j = n$ and establish it for $j = n + 1$.

FFRAG$_{n+1}$ is obtained from FFRAG$_n$ by replacing some nodes s' in its frontier with FDAG[s', g_{n+1}]. By definition, FDAG[s', g_{n+1}] is acyclic, and is generated by T_F. Hence, applying the induction hypothesis to FFRAG$_n$, we conclude that FFRAG$_{n+1}$ is acyclic and is generated by T_F.

By the induction hypothesis, the eventualities $g_1, \ldots, g_j$ are fulfilled for s_0 in the fault-free portion of FFRAG$_n$. Hence, by definition of fulfilled, and the fact that FFRAG$_{n+1}$ is obtained by extending the frontier of FFRAG$_n$, we conclude that $g_1, \ldots, g_j$ are fulfilled for s_0 in the fault-free portion of FFRAG$_{n+1}$. We now show that g_{n+1} is fulfilled for s_0 in the fault-free portion of FFRAG$_{n+1}$. First, we note, by its definition, that FFRAG$_n$ is generated by T_F. Hence, FFRAG$_n$ is also generated by T_0. Hence, Proposition 7.1.6 is applicable to FFRAG$_n$. The rest of the argument is in two cases.

Case 1: g_{n+1} is $\mathsf{E}[g\mathsf{U}h]$ for some formulae g, h. By Proposition 7.1.6, these exists a maximal fault-free path $s_0 = v_0, v_1, v_2, \ldots, v_\ell$ in FFRAG$_n$, where $v_k \in$ *frontier*(FFRAG$_n$) and

(i) there exists $i \in [0:\ell]$ such that
$h, \mathsf{E}[g\mathsf{U}h] \in L(v_i)$ and for all $j \in [0:i)$, $g, \mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h] \in L(v_j)$,

or

(ii) for all $j \in [0:\ell]$, $g, \mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h] \in L(v_j)$.

If (i) holds, then we are done. Otherwise, $\mathsf{E}[g\mathsf{U}h] \in L(v_\ell)$, and so FDAG[$c_\ell, \mathsf{E}[g\mathsf{U}h]$] was attached at c_ℓ in constructing FFRAG$_{n+1}$. Thus $\mathsf{E}[g\mathsf{U}h]$ is fulfilled.

Case 2: g_{n+1} is $\mathsf{A}[g\mathsf{U}h]$ for some formulae g, h. Let $s_0 = c_0, c_1, \ldots, c_k$ be an arbitrary path in FFRAG$_{n+1}$ such that $c_k \in$ *frontier*(FFRAG$_{n+1}$). By construction, *frontier*(FFRAG$_n$) is a cutset of FFRAG$_{n+1}$. Hence, $c_0, c_1, \ldots, c_k$ contains exactly one node (call it c_ℓ) that is in *frontier*(FFRAG$_n$). Now $c_0, c_1, \ldots, c_\ell$ is a maximal path in FFRAG$_n$, Thus, by Proposition 7.1.6:

(i) there exists $i \in [0:\ell]$ such that
$h, \mathsf{A}[g\mathsf{U}h] \in L(c_i)$ and for all $j \in [0:i)$, $g, \mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h] \in L(c_j)$,

or

(ii) for all $j \in [0:\ell]$, $g, \mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h] \in L(c_j)$.

If (i) holds, then we are done. Otherwise, $\mathsf{A}[g\mathsf{U}h] \in L(c_\ell)$, and so FDAG[$c_\ell, \mathsf{A}[g\mathsf{U}h]$] was attached at c_ℓ in constructing FFRAG$_{n+1}$. So, $c_k \in$ *frontier*(FDAG[$c_\ell, \mathsf{A}[g\mathsf{U}h]$]). Thus $h \in L(c_k)$. Since $c_0, c_1, \ldots, c_k$ is an arbitrary maximal path in FFRAG$_{n+1}$, we conclude that $\mathsf{A}[g\mathsf{U}h]$ is fulfilled. □

PROPOSITION 7.1.8. *The synthesis method terminates.*

PROOF. We show in turn that each step of the method terminates.

Step 1: The number of possible distinct labels is bounded by $2^{|cl(spec)|}$. Since nodes of the same type and with identical labels are merged, the construction of T_0 must terminate.

Step 2: Since each application of a deletion rule removes one node, and T_0 is finite, the deletion process must terminate.

Step 3: Since the frontiers of all the fragments involved are finite, it is clear that al of the substeps terminate. Since the number of eventualities in $L(c)$ is finite, the overall step terminates.

Step 4: There are only $O(2^{|cl(spec)|})$ fragments. Each application of substep 4(b)i to a node c either adds FFRAG[c] to the model, or identifies c with some other already-present interior node. Since the number of fragments is finite, after some point, all steps will be to merge a frontier node with an interior node, thereby decreasing the number of frontier nodes. Since, at this point, the number of frontier nodes is finite, it follows that, after enough steps, the frontier becomes empty and the model construction process terminates.

Step 5: Since M_F is finite, it is clear that each substep is iterated only a finite number of times. □

THEOREM 7.1.9 (SOUNDNESS). *Let $M = (s_0, S, A, A_F, L)$ be the structure produced in Step 4 of the method, and let $M_F = (s_0, S, A, A_F, L_0)$ where L_0 is L restricted to the atomic propositions occurring in spec. Then, for all $s \in S$, $f \in L(s)$: $M_F, s \models_n f$.*

PROOF. The proof is by induction on the length of f in positive normal form (i.e., negations are pushed inward using dualities so that only atomic propositions are negated). By construction, M_F is generated by T_F. Since T_F is a subgraph of T_0, M_F is also generated by T_0. Hence Propositions 7.1.5 and 7.1.6 apply to M_F. Let s be an arbitrary state in S, and let f be an arbitrary formula in $L(s)$. We consider all the possible cases for the main modality of f. In each case, we assume $f \in L(s)$, and establish $M_F, s \models_n f$. (For clarity, we underline the case assumption.)

$\underline{f = p, \text{ where } p \text{ is an atomic proposition}}$. $M, s \models_n f$ by definition of $\models_n$.

$\underline{f = \neg p}$. Since $L(s)$ contains no propositional inconsistencies (otherwise s would have been deleted), $p \notin L(s)$. Hence $M, s \not\models_n p$. Hence $M, s \models_n \neg p$ by definition of $\models_n$.

$\underline{f = g \vee h}$. Since $L(s)$ is downward-closed, we conclude $g \in L(s)$ or $h \in L(s)$. Applying the induction hypothesis, we get $M, s \models_n g$ or $M, s \models_n h$. Hence $M, s \models_n g \vee h$ by definition of $\models_n$.

$\underline{f = g \wedge h}$. Since $L(s)$ is downward-closed, we conclude $g \in L(s)$ and $h \in L(s)$. Applying the induction hypothesis, we get $M, s \models_n g$ and $M, s \models_n h$. Hence $M, s \models_n g \wedge h$ by definition of $\models_n$.

$\underline{f = \mathsf{AX}_i g}$. Let t be an arbitrary non-fault P_i-successor of s, that is, $(s, i, t) \in A$. By Proposition 7.1.5, Clause 3, $g \in L(t)$. By the induction hypothesis, $t \models g$. Since t was arbitrarily chosen, we conclude $s \models \mathsf{AX}_i g$.

$\underline{f = \mathsf{EX}_i\, g}$. By Proposition 7.1.5, Clause 5, s has some non-fault P_i-successor t in M such that $g \in L(t)$. By the induction hypothesis, $t \models g$. Hence, $s \models \mathsf{EX}_i\, g$.

$\underline{f = \mathsf{A}[g \mathsf{U} h]}$. We will show that for every maximal fault-free path $s = s_0, s_1, s_2, \ldots$ in M there is an n such that

$$h, \mathsf{A}[g\mathsf{U}h] \in L(s_n) \ \text{ and } \ \text{for all } j \in [0:n),\ \ g, \mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h] \in L(s_j).$$

The induction hypothesis can then be applied to obtain

$$M, s_n \models_n h \quad \text{and for all} \quad j \in [0:n),\ \ M, s_j \models_n g.$$

By definition of $\models_n$, we then conclude $M, s_0 \models_n \mathsf{A}[g\mathsf{U}h]$.

Let $s = s_0, s_1, s_2, \ldots$ be an arbitrary maximal fault-free path in M starting in state s. By construction of M, every node occurs in some fragment embedded in M and at the frontier of each fragment there occurs the root of still another fragment embedded in M. Thus, there must be a least $i \geq 0$ such that s_i is the root of some fragment FFRAG$[s_i]$ embedded in M. If

there exists $\ell \in [0:i)$ such that
$h, \mathsf{A}[g\mathsf{U}h] \in L(s_\ell)$ and for all $j \in [0:\ell)$, $g, \mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h] \in L(s_j)$

then we are done immediately. Otherwise, by Proposition 7.1.6, we have

$$g, \mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h] \in L(s_i).$$

Since s_i is the root of FFRAG$[s_i]$, $\mathsf{A}[g\mathsf{U}h]$ is fulfilled along all maximal fault-free paths starting in s_i, by Proposition 7.1.7. Hence

there exists $\ell \geq 0$ such that
$h, \mathsf{A}[g\mathsf{U}h] \in L(s_\ell)$ and for all $j \in [0:\ell)$, $g, \mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h] \in L(s_j)$

and we are done.

$\underline{f = \mathsf{E}[g\mathsf{U}h]}$. We will show that there exists a finite fault-free path $s = s_0, \ldots, s_n$ in M such that

$$h, \mathsf{A}[g\mathsf{U}h] \in L(s_n) \ \text{ and } \ \text{for all } j \in [0:n),\ g, \mathsf{A}[g\mathsf{U}h], \mathsf{AXA}[g\mathsf{U}h] \in L(s_j).$$

The induction hypothesis can then be applied to obtain

$$M, s_n \models_n h \ \text{ and } \ \text{for all } j \in [0:n),\ M, s_j \models_n g.$$

By definition of $\models_n$, we conclude $M, s_0 \models_n \mathsf{E}[g\mathsf{U}h]$.

From Proposition 7.1.6 and $\mathsf{E}[g\mathsf{U}h] \in L(s_0)$, we have that there exists a maximal fault-free path $s_0, s_1, s_2, \ldots$ in M such that

for all $j \geq 0$, $g, \mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h] \in L(s_j)$, or
there exists $i \geq 0$ such that
$h, \mathsf{E}[g\mathsf{U}h] \in L(s_i)$ and for all $j \in [0:i)$, $g, \mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h] \in L(s_j)$.

By the construction of M, every node occurs in some fragment embedded in M and at the frontier of each fragment there occurs the root of still another fragment embedded in M. Thus, there must be a least $i \geq 0$ such that s_i is the root of some fragment FFRAG$[s_i]$ embedded in M. If

there exists $\ell \in [0:i)$ such that
$h, \mathsf{E}[g\mathsf{U}h] \in L(s_\ell)$ and for all $j \in [0:\ell)$, $g, \mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h] \in L(s_j)$

then we are done: let $n = \ell$. Otherwise, we have

$$g, \mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h] \in L(s_i).$$

Since s_i is the root of FFRAG[s_i], $\mathsf{E}[g\mathsf{U}h]$ is fulfilled for s_i in FFRAG[s_i]. Hence there is a fault-free path $s_i = t_0, t_1, \ldots, t_m$ in FFRAG[s_i] such that

$$h, \mathsf{E}[g\mathsf{U}h] \in L(t_m) \text{ and for all } j \in [0:m),\ g, \mathsf{E}[g\mathsf{U}h], \mathsf{EXE}[g\mathsf{U}h] \in L(t_j).$$

Then $s = s_0, s_1, s_2, \ldots, s_i = t_0, t_1, \ldots, t_m$ is the desired finite fault-free path.

$\underline{f = \mathsf{A}[g\mathsf{W}h]}$. By Proposition 7.1.6, Clause 3, we have, for every maximal fault-free path $s = s_0, s_1, s_2, \ldots$ in M,

for all $j \geq 0$, $h, \mathsf{A}[g\mathsf{W}h], \mathsf{AXA}[g\mathsf{W}h] \in L(v_j)$, or
there exists $i \geq 0$ such that
$g, h, \mathsf{A}[g\mathsf{W}h] \in L(v_i)$ and for all $j \in [0:i)$, $h, \mathsf{A}[g\mathsf{W}h], \mathsf{AXA}[g\mathsf{W}h] \in L(v_j)$.

Applying the induction hypothesis to this, we obtain

for all $j \geq 0$, $M, s_j \models_n h$ or
there exists $i \geq 0$ such that
$M, s_i \models_n g \wedge h$ and for all $j \in [0:n)$, $M, s_j \models_n h$.

By definition of $\models_n$, we conclude $M, s_0 \models_n \mathsf{A}[g\mathsf{W}h]$.

$\underline{f = \mathsf{E}[g\mathsf{W}h]}$. By Proposition 7.1.6, Clause 4, we have, for some maximal fault-free path $s = s_0, s_1, s_2, \ldots$ in M,

for all $j \geq 0$, $h, \mathsf{E}[g\mathsf{W}h], \mathsf{EXE}[g\mathsf{W}h] \in L(v_j)$, or
there exists $i \geq 0$ such that
$g, h, \mathsf{E}[g\mathsf{W}h] \in L(v_i)$ and for all $j \in [0:i)$, $h, \mathsf{E}[g\mathsf{W}h], \mathsf{EXE}[g\mathsf{W}h] \in L(v_j)$.

Applying the induction hypothesis to this, we obtain

for all $j \geq 0$, $M, s_j \models_n h$ or
there exists $i \geq 0$ such that
$M, s_i \models_n g \wedge h$ and for all $j \in [0:n)$, $M, s_j \models_n h$.

By definition of $\models_n$, we conclude $M, s_0 \models_n \mathsf{E}[g\mathsf{W}h]$. □

COROLLARY 7.1. *Let $M = (s_0, S, A, A_F, L)$ be the structure produced in Step* 4 *of the method, and let $M_F = (s_0, S, A, A_F, L_0)$ where L_0 is L restricted to the atomic propositions occurring in spec. Also, let S_F be the set of perturbed states in M_F. Then*

(1) $M_F, s_0 \models_n$ *init–spec* $\wedge\ \mathsf{AG}$(*global–spec*) $\wedge\ \mathsf{AG}$(*coupling–spec*),
(2) $M_F, S_F \models_n Label_{TOL}(spec)$.

PROOF. From our semantics of concurrent programs given in Section 2.1 and the pseudocode for Step 5 (the extraction step) of our method given in Section 5.2, we can check that execution of the extracted program P does indeed generate M_F. This follows, since each transition in M_F corresponds to a unique arc in P whose execution generates that transition. Furthermore, the introduction of shared variables ensures that every state of P corresponds to a unique state of M_F (which furthermore agrees with it on all atomic propositions) and vice versa. Formalization of this argument is straightforward, and is omitted. We refer the reader to Attie and Emerson [2001] for examples of formal arguments of this kind.

From the pseudocode for Step 1 of our method given in Section 5.2 (especially Definitions 5.1.1 and 5.1.2), we have

(1) $L(s_0) = \{init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)\}$,
(2) $\forall s \in S_F : Label_{TOL}(spec) \subseteq L(s)$.

The corollary follows immediately by applying Theorem 7.1.9. □

Comparing with Section 3, we confirm that our synthesis method solves the synthesis problem. A slight technicality is that our method produces models with a single initial state, whereas the method is stated in (somewhat more general) terms of models with a finite set of initial states.

7.2 Completeness

Completeness of our method is the requirement that, if some fault-tolerant program satisfying the requirements of Section 3 exists, then our method produces such a program. Since a fault-tolerant program can always be extracted from the model M_F produced by our method, we actually formalize completeness in terms of the existence of M_F: if a model satisfying the requirements of Section 3 exists, then our method produces such a model M_F.

To assure completeness, the deletion rules in Figure 2 are formulated so that a nodes in the tableau is deleted only if necessary. Keeping in mind that, in the final model, the label of every state must be satisfied by that state, we examine all of the deletion rules from Figure 2:

—*DeleteP*: A propositionally inconsistent node label cannot be satisfied.
—*DeleteOR*: Each successor of an OR-node d gives one of the (finitely many) ways in which $L(d)$ may possibly be satisfied. If all these successors are deleted, then there is no way of satisfying $L(d)$.
—*DeleteAND*: If a successor (but not a fault-successor) of an AND-node c is deleted, that means that some elementary formula in $L(c)$ cannot be satisfied. Hence, neither can $L(c)$. If a fault-successor of c is deleted, then from c, some fault-action in the fault-specification can be executed, leading to an unsatisfiable OR-node. Since these faults must be allowed to occur in any state (i.e., AND-node) of the final extracted model, the only recourse is to delete the AND-node itself.
—*Delete*AU: Some AU formula in a node e cannot be fault-free-fulfilled by the normal and recovery transitions leaving e, even when all possible choices for the successors of OR-nodes are considered (i.e., all possible fault-free full subdags rooted at e). Thus, $L(e)$ cannot be satisfied in any model, and so e must be deleted.
—*Delete*EU: Some EU formula in a node e cannot be fault-free-fulfilled by the normal and recovery transitions leaving e, even when all possible choices for the successors of OR-nodes are considered (i.e., all possible fault-free paths starting at e). Thus, $L(e)$ cannot be satisfied in any model, and so e must be deleted.

Thus, we see that each rule only deletes a node if there is no way of satisfying the node's label. In particular, for an AND-node c, this means that there is no model M containing c as a state, such that c satisfies its label ($M, c \models_n L(c)$).

We now define the notion of *fault-subgraph*. A fault-subgraph rooted at node e is the fault-tolerant analogue of an infinite tree-like model rooted at e—it certifies that the label of every node reachable from e (including e itself) is satisfied (in the final model) despite the occurrence of faults.

Even though all formulae in $L(e)$ are satisfied by the fault-free portion of T_F starting at e, it could be that fault-transitions lead from e to another node for which this isn't the case. Thus, $L(e)$ could be satisfiable, but fault-transitions lead from e to a node e' such that $L(e')$ is not satisfiable. In some cases, this may require that e itself is deleted, for example, if e is an AND-node and e' is one of it's fault-successors (as discussed above under the DeleteAND bullet). Thus, nodes that are not deleted in the synthesis method of Emerson and Clarke [1982] could be deleted in our method. However, every node that is deleted by the Emerson and Clarke [1982] method will also be deleted by our method, since its label will be unsatisfiable (by virtue of the completeness of the Emerson and Clarke [1982] method).

Definition 7.2.1 (*Fault-Subgraph*). A *fault-subgraph* D rooted at node e is a directed bipartite AND/OR graph satisfying all of the following conditions:

(1) All other nodes in D are reachable from e.
(2) All nodes in D are propositionally consistent.
(3) For every AND-node c in D, all fault-successors of c are sons of c in D, and all nodes in $Tiles(c)$ are sons of c in D.
(4) For every OR-node d in D, there exists exactly one AND-node c such that $c \in Blocks(d)$ and c is a son of d in D.
(5) For every node e' in D, every eventuality in $L(e')$ is fault-free fulfilled for e' in D.

To establish completeness, we show that if there exists a fault-subgraph rooted at node e, then, for every eventuality in the label of e, there exists a fault-free full subdag rooted at e which certifies the fulfillment of that eventuality. Furthermore, e will retain enough successors so that it is not deleted by virtue of the DeleteOR and DeleteAND rules. We will show below that if a node e is deleted at some point in our method, then there does not exist a fault-subgraph rooted at e.

LEMMA 7.2.2. *Let e be a node in T_0. If $\mathsf{A}[g \mathsf{U} h] \in L(e)$ and there exists a fault-subgraph D whose root is labeled with $L(e)$, then there exists a fault-free full subdag D^* which is rooted at e and fulfills $\mathsf{A}[g \mathsf{U} h]$, and for which the following hold:*

(1) *Every node of D^* is the root of some fault-subgraph, and*
(2) *D^* is embedded in T_0 at e.*

PROOF. By Definition 7.2.1, Clause 5, $\mathsf{A}[g \mathsf{U} h]$ is fault-free-fulfilled for e in D. Hence, there exists a fault-free full subdag D_0 in D whose root is labeled with $L(e)$ and which fulfills $\mathsf{A}[g \mathsf{U} h]$. We can assume that $L(e)$ is unique in D_0. If not, take D_0 to be the subtree of D_0 rooted at the deeper occurrence of $L(e)$. By Definition 7.2.1 and the construction of T_0, it is easy to see that D_0 is generated by T_0.

From Definition 7.2.1, we easily verify that for any fault-subgraph D' and any node e' in D', the subgraph of D' induced by the nodes reachable from e' is also a fault-subgraph. Hence, every node in D is the root of some fault-subgraph. Hence, every node in D_0 is the root of some fault-subgraph.

We construct a series of fault-free full subdags $D_0, D_1, \ldots, D_n = D^*$, where D_{i+1} is obtained from D_i by merging a pair of duplicate nodes, (i.e., AND/OR nodes, respectively, with the same label). Define the depth of a node to be the length of a longest path in D_0 from that node back to e. Thus, $depth(v) = 1 + \max\{depth(v') \mid v' \text{ is a predecessor of } v\}$. Suppose u and v are both AND-nodes or OR-nodes in D_i with the same label, and without loss of generality suppose that $depth(u) \geq depth(v)$. We then replace the shallower node v by the deeper node u to obtain D_{i+1}. That is, we replace each edge (w, v) by the edge (w, u), and remove all nodes that are rendered unreachable from the root.

We show by induction on i that, for each D_i, the following all hold:

(1) D_i is a dag generated by T_0.
(2) $root(D_i) = root(D_0)$.
(3) D_i is a full-subdag that fulfills $\mathsf{A}[g \mathsf{U} h]$.
(4) Every node of D_{i+1} is the root of some fault-subgraph.

Basis. Clauses 1– 4 hold for D_0 by virtue of its construction.

Induction step. Assume Clauses 1–3 hold for D_i and show that they hold for D_{i+1}.

Clause 1. The argument is identical to that in the proof of Lemma 4.9.2 [Emerson 1981].

Clause 2. Since $L(e)$, the label of $root(D_0)$, is unique in D_0, it cannot be deleted. Hence $root(D_{i+1}) = root(D_i) = root(D_0)$.

Clause 3. The duplicate elimination step preserves the successor requirements in Definition 4.7. Hence D_{i+1} is a full subdag. It remains to show that D_{i+1} fulfills $\mathsf{A}[g \mathsf{U} h]$. This argument is the same as that in the proof of Lemma 4.9.2 [Emerson 1981], except that internal nodes also have to be dealt with, since Lemma 4.9.2 [Emerson 1981], is for AF and not AU.

Clause 4. By Definition 7.2.1, we see that the property of being the root of a subgraph depends only on the labeling of a node. Hence this property is clearly preserved by the duplicate elimination step.

Continue eliminating duplicates until there are none left. Let D^* be the resulting full subdag. Since D^* is generated by T_0 and contains no duplicates, it is easy to see that D^* is embedded in T_0. Furthermore, since $root(D^*) = root(D_0)$, and $L(root(D_0)) = L(e)$, we have $L(root(D^*)) = L(e)$. By construction, node labels are unique in T_0, and hence $root(D^*) = e$. □

LEMMA 7.2.3. *Let e be a node in T_0. If $\mathsf{E}[g\mathsf{U}h] \in L(e)$ and there exists a fault-subgraph D whose root is labeled with $L(e)$, then there exists a finite fault-free path π starting in e which fulfills $\mathsf{E}[g\mathsf{U}h]$ and for which the following hold:*

(1) *Every node of π is the root of some fault-subgraph, and*
(2) *π is a path in T_0.*

PROOF. By Definition 7.2.1, Clause 5, $\mathsf{E}[g\mathsf{U}h]$ is fulfilled for e in D. Hence, there exists a finite fault-free path ρ in D whose first state is labeled with $L(e)$ and which fulfills $\mathsf{E}[g\mathsf{U}h]$ (i.e., the last state of ρ is labeled with h and all other states are labeled with g). We can assume that $L(e)$ is unique in ρ. If not, take ρ to be the suffix of ρ starting at the deeper occurrence of $L(e)$. Let path π result from ρ by identifying all duplicate nodes (i.e., π may contain cycles). Since ρ fulfills $\mathsf{E}[g\mathsf{U}h]$, it is easy to see that π also fulfills $\mathsf{E}[g\mathsf{U}h]$. By Definition 7.2.1 and the construction of T_0, it is easy to see that ρ is generated by T_0. Since π contains no duplicates, it is easy to see that π is a path in T_0. Also, since node labels are unique in T_0 and the first state of π is labeled with $L(e)$, the first state of π must be e.

From Definition 7.2.1, we easily verify that, for any fault-subgraph D' and any node e' in D', the subgraph of D' induced by the nodes reachable from e' is also a fault-subgraph. Hence, every node in D is the root of some fault-subgraph. Hence, every node in ρ is the root of some fault-subgraph.

Finally, by Definition 7.2.1, we see that the property of being the root of a fault-subgraph depends only on the labeling of a node. Since every node of π has the same label as some node of ρ, we conclude that every node of π is the root of some fault-subgraph. □

THEOREM 7.2.4 (COMPLETENESS). *Let e be a node in T_0 that is deleted at some point in our method. Then there does not exist a fault-subgraph rooted at e.*

PROOF. We prove the theorem by induction on the "time" at which node e was deleted. Nodes are deleted in Step 2 of the synthesis method (Section 5), as a result of applying one of the deletion rules in Figure 2. We treat each rule in turn.

DeleteP. Hence $L(e)$ is propositionally inconsistent. Hence, by Definition 7.2.1, Clause 2, there does not exist a fault-subgraph rooted at e.

DeleteOR. Hence e is an OR-node all of whose AND-node successors $c_1, \ldots, c_n$ have already been deleted. By the induction hypothesis, for all $i \in [1:n]$, there does not exist a fault-subgraph rooted at c_i. By construction of our synthesis method, $\{c_1, \ldots, c_n\} = \mathit{Blocks}(e)$. Therefore, we conclude by Definition 7.2.1, Clause 4, that there does not exist a fault-subgraph rooted at e.

DeleteAND. Hence e is an AND-node one of whose OR-node successors d has already been deleted. By the induction hypothesis, there does not exist a fault-subgraph rooted at d. By construction of our synthesis method, $d \in \mathit{Tiles}(e)$ or d is a fault-successor of e. In either case, we conclude by Definition 7.2.1, Clause 3, that there does not exist a fault-subgraph rooted at e.

*Delete*AU. We establish the contrapositive. Suppose that $\mathsf{A}[g\mathsf{U}h] \in L(e)$ and there exists a fault-subgraph D rooted at e. Then, by Lemma 7.2.2, there exists a fault-free full subdag D^* rooted at e and which fulfills $\mathsf{A}[g\mathsf{U}h]$ and such that

(1) every node of D^* is the root of some fault-subgraph, and
(2) D^* is embedded in T_0 at e.

By the induction hypothesis, no nodes in D^* have been deleted up to now. Therefore, e cannot be deleted by applying rule *Delete*AU.

*Delete*EU. We establish the contrapositive. Suppose that $\mathsf{E}[g\mathsf{U}h] \in L(e)$ and there exists a fault-subgraph rooted at e. Then, by Lemma 7.2.3, there exists a finite fault-free path π starting in e which fulfills $\mathsf{E}[g\mathsf{U}h]$ and for which the following hold:

(1) every node of π is the root of some fault-subgraph, and
(2) π is a path in T_0.

By the induction hypothesis, no nodes in π have been deleted up to now. Therefore, e cannot be deleted by applying rule *Delete*EU. □

COROLLARY 7.2. *If d_0, the initial OR-node of T_0, is deleted, then no model M_F of the specification exists.*

PROOF. We establish the contrapositive. Suppose there exists a model $M_F = (S_0, S, A, A_F, L)$ of the specification (where each label $L(s)$ gives only the atomic propositions true in s, and S_0 contains at least one state s_0). Then $M_F, s_0 \models spec$. Construct $M' = (S_0, S, A, A_F, L')$ from M_F by labeling each state s of M_F as follows. If s is normal, then $L'(s) \overset{\text{df}}{=} \{f \mid f \in cl(spec) \text{ and } M_F, s \models f\}$. If s is perturbed, then $L'(s) \overset{\text{df}}{=} Label_{TOL}(spec) \cup L(s)$. By definition, $spec \in L'(s_0)$. Take the portion of M' reachable (via both normal and fault-transitions) from s_0 and unwind it into an infinite tree. Classify all the nodes as AND-nodes, and then interject exactly one unique OR-node between every pair of adjacent AND-nodes, that is, replace $c \xrightarrow{i} c'$ by $c \xrightarrow{i} d \rightarrow c'$, where d is a "new," that is, not previously present, node, and the label of d is set to be the label of c'. The result is a fault-subgraph FS with root s_0. Since $L(d_0) = \{spec\} \subseteq L(s_0)$, it is clear that we can replace s_0 in FS by d_0 and the result will still be a fault-subgraph. Thus, there exists a fault-subgraph rooted at d_0. By Theorem 7.2.4, d_0 is not deleted. □

7.3 Fault-Closure

Fault-closure means that all the faults given in the fault specification are included in the final model. Fault-closure follows easily from the construction of our method: for every AND-node of T_0, all possible fault-successors are generated.

PROPOSITION 7.3.1. *Let c be an AND-node in T_F. If $c \xrightarrow{a,TOL} d$ for some fault action $a \in F$, then d is an OR-node in T_F.*

PROOF. The proposition holds in T_0 by construction of our method. T_F is obtained from T_0 by applying the deletion rules in Figure 2. Hence, if any fault-successor d of an AND-node c is deleted, then c will also be deleted by virtue of the DeleteAND rule. Hence, all remaining AND-nodes (i.e., all AND-nodes in T_F) will have all possible fault-successors. □

THEOREM 7.3.2 (FAULT-CLOSURE). *For all $s \in S$, $a \in F$ such that $s(a.guard) = true$, there exists $t \in S$ such that $s \xrightarrow{a} t \in A_F$.*

PROOF. $M_F = (s_0, S, A, A_F, L)$ is built by unraveling T_F. By construction of our method, M_F inherits the local structure of T_F. Since every state of M_F is an AND-node of T_F, we conclude by Proposition 7.3.1 that every state of M_F has all possible fault-successors. □

7.4 Complexity of the Method

We give the time complexity in terms of two parameters: (1) the length of the temporal specification $spec = init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$, that is, the sum of the lengths of the problem specification and the problem-fault coupling specification, and (2) the size of the description of the set of fault-actions F. We assume that each auxiliary atomic proposition is mentioned at least once in the problem-fault coupling specification (since otherwise it can be removed from the fault specification without changing the specification logically), and so the size of the set of auxiliary atomic propositions is always smaller than the size of the problem-fault coupling specification. Hence it is not included as a parameter in the complexity analysis.

The construction of T_0 involves creating nodes whose labels are subsets of $cl(spec)$, and, in the case of perturbed states, subsets of $cl(spec \wedge \mathsf{AFAG}(global\text{–}spec))$, since $\mathsf{AFAG}(global\text{–}spec)$ is the only "new" subformula that can be added by $Label_{TOL}$ (Definition 2.1). Since $\mathsf{AG}(global\text{–}spec)$ is a subformula of $spec$, we have $|cl(spec \wedge \mathsf{AFAG}(global\text{–}spec))| \leq 2|cl(spec)|$. Hence, the number of nodes in T_0 is $O(exp(|cl(spec \wedge \mathsf{AFAG}(global\text{–}spec))|))$, where $exp(n) \overset{\text{df}}{=} 2^n$. This is $O(exp(2|cl(spec)|))$, which is $O(exp(4|spec|))$. Also, for any node e of T_0, $L(e) \subseteq cl(spec \wedge \mathsf{AFAG}(global\text{–}spec))$. Hence $|L(e)| \leq 2|cl(spec)|$, and so $|L(e)| \leq 4|spec|$. Hence, the sum of the lengths of the formulae in $L(e)$ is in $O(|spec|^2)$, since each such formula has length in $O(|spec|)$. Thus, the size of (a reasonable encoding of) each node in T_0 is $O(|spec|^2)$.

Each node e in T_0 is expanded once. Expanding a node involves (1) calculating $Blocks(e)$ or $Tiles(e)$ as appropriate, and (2) if e is an AND-node, applying all the fault-actions in F to e.

Calculating $Tiles(e)$ has cost at most $|L(e)|$, since each formula in $L(e)$ contributes at most one element of $Tiles(e)$. Thus, the cost is $O(|spec|)$. Calculating $Blocks(e)$ involves constructing a tree, and applying α-β expansions until all the leaves contain only elementary formulae in their node labels. This has cost at most $O(|spec|^2 \sum_{f \in L(e)} |f|)$, since each α-β expansion discharges one connective in one formula in $L(e)$, and generates at most two new nodes, each of size $O(|spec|^2)$. Since each formula in $L(e)$ has length in $O(|spec|)$, and there are $O(|spec|)$ such formulae, the total cost of calculating $Blocks(e)$ is in $O(|spec|^4)$.

Each action $a \in F$ is a guarded command. We consider the size $|a|$ of a to be the length of the text of the guarded command.[14] Then, we define $|F| = \sum_{a \in F} |a|$. Applying all the fault-actions in F to an AND-node c involves "executing" each fault-action a in the propositional valuation given by node c, and generating a fault-successor OR-node if a's guard is true in c. This can be done with cost $|F|$.

Thus, the cost of expanding a single node is $O(|spec|^4 + |F|)$. There are at most $O(exp(4|spec|))$ nodes in T_0. Hence, the total cost of node expansion is $O((|spec|^4 + |F|)exp(4|spec|))$.

The remaining steps, namely applying the deletion rules, constructing the fragments from the fault-free full subdags, and constructing the model from the fragments, can all be done in time polynomial in the size of T_0, that is, in time $exp(O(|spec|))$. The proof is essentially the same as that for the CTL decision procedure. We refer the reader to Emerson [1981] for the details.

Thus, the overall time complexity is $|F|exp(O(|spec|))$, that is, single exponential in the size of the specification (= size of problem specification + size of problem-fault coupling specification), and linear in the size of the description of the fault actions. It is clear form the above discussion that the overall space complexity is also $|F|exp(O(|spec|))$.

All synthesis methods based on exhaustive state exploration will have a time complexity no better than single exponential in the specification size. Some methods [Kupferman et al. 2000; Pnueli and Rosner 1980a, 1989b; Wong-Toi and Dill 1990] have double exponential time and space complexity in the size of the specification.

8. DISCUSSION

8.1 The Scope of Our Synthesis Method

Our method is capable of dealing with any fault model in which faults can be represented as actions that perturb the system state. Our synthesis method guarantees correctness properties only once faults stop occurring, that is, along fault-free fullpaths. However, a single fault can have a permanent effect in that it can perturb the state of some system component in a way that permanently changes the behavior of that component. For example, in Section 2.3, an example is given of a stuck-at-low-voltage fault. A *single* occurrence of this fault in a wire causes the wire to *permanently* change its behavior so that it only outputs a low voltage, regardless of its input. Our method is able to deal with such faults because our use of auxiliary atomic propositions enables us to permanently record the occurrence of a fault, and so we can model the permanent change of system behavior that results. Another example of this is the mutual exclusion example shown in Section 6.1, where process P_1 may possibly stay fail-stopped forever after its fail-stop failure occurs.

Since our method can model faults that have permanent effects, as well as ones that have only transient effects, it has a broad scope of application. The

[14]We assume an underlying notation for guarded commands, for example, that given in Dijkstra [1976].

main limitation of our method is its underlying model of concurrent computation (Section 2.1): a process can read and update many shared variables in a single atomic transition. Extending our method to more realistic models of concurrency is thus a topic of future work, and is discussed further in Section 9.2 below. Another limitation is that our method cannot deal with faults that correspond to certain adversaries that can read any part of the system state. In particular, if the adversary can read the shared variables, then our method cannot model the associated faults. This is because the fault-actions are applied to the tableau T_0, whereas the shared variables are introduced only after the final model has been extracted, in order to distinguish propositionally identical states whose labels differ in some temporal formula (as in Emerson and Clarke [1982]). So, for example, our method cannot model an adversary that increments the shared variables.

8.2 Extension of the Synthesis Method to Accommodate Multitolerance

Our synthesis method is not dependent on the particular way that labels are computed for perturbed states (Definition 2.1). It is sound and complete relative to any definition of the perturbed state labels. Thus, variants and extensions of the method can easily be generated by simply changing the way that labels of perturbed states are computed. For example, in Arora and Kulkarni [1998], the concept of *multitolerance* is presented. In multitolerance, the set of fault actions is partitioned into classes, and different fault classes may require different kinds of tolerance (masking, nonmasking, or fail-safe). Thus, one class of faults may require masking tolerance, while another class may require only failsafe tolerance.

It is straightforward to extend our method to deal with multitolerance. Simply allow the required tolerance to vary, depending on the particular fault action. Our method uses the function $Label_{TOL}(spec)$ ($TOL \in$ {nonmasking, fail-safe, masking}) to determine the labels of perturbed states. We replace this with $Label_a(spec)$, where a is the particular fault-action that has been executed. Thus, if nonmasking tolerance is required for fault action $a1$, and fail-safe tolerance for fault action $a2$, then we set $Label_{a1}(spec) = Label_{\text{nonmasking}}(spec)$ and $Label_{a2}(spec) = Label_{\text{fail-safe}}(spec)$. Also, Definition 5.1.1 must be changed to

$$c \xrightarrow{a,TOL} d \quad \text{if and only if} \quad \exists\varphi \subseteq \mathcal{AP} : c(a.guard) = true \text{ and } \{L(c){\uparrow}\mathcal{AP}\}\, a.body\, \{\varphi\} \text{ and } L(d) = \varphi \cup Label_a(spec).$$

These are the only needed changes. The new definition of $Label_a(spec)$ enables the label of the resulting OR-node to depend on the fault-action a. Thus, different fault-actions can be assigned different tolerances. The required recovery transitions are computed automatically by the method, exactly as before. Note that, as discussed in Section 5.2, the application of the various faults (e.g., $a1$, $a2$ above) is intertwined with the synthesis of normal and recovery transitions. We emphasize that the various faults are not applied in any particular order.

8.3 An Alternative Synthesis Method That Accommodates Fault-Prone Paths

Our soundness results (Theorem 7.1.9 and Corollary 7.1) are given in terms of fault-free paths only, that is, using the $\models_n$ satisfaction relation. That is, they give the formulae that are satisfied in the initial state, and in the perturbed states, but where satisfaction is determined by only considering fault-free paths (i.e., the A and E path quantifiers range only over fault-free paths). We consider an alternative method that also deals with fault-prone paths, that is, so that formulae can be asserted to be true in the initial state and in perturbed states under the $\models$ satisfaction relation, in which the A and E path quantifiers range over all paths, both fault-free and fault-prone.

A starting point for this alternative synthesis method could be the following:

(1) Use full subdags that include fault-transitions, that is, AND-nodes c in a full subdag need as successors both the non-fault-successors (given by $Tiles(c)$) and the fault-successors.
(2) The definition of fulfillment must take fault-prone paths into account, that is, use the "regular" definition of fulfillment (Definition 4.6) , rather than the fault-free one (Definition 5.1.3).

While this alternative method would accommodate stronger correctness statements, it may be inapplicable in many situations where our current method would work. For example, repeated occurrence of faults could violate some correctness property, causing the problem to have no model in this setting, whereas a model would exist in the setting where the correctness property holds once faults stop occurring. Thus our current method guarantees somewhat weaker correctness properties, but has a wider application, than the proposed alternative method. Working out the alternative method in full is a topic for future work.

9. RELATED WORK AND CONCLUSIONS

9.1 Related Work

A fault model may be considered as a particular type of *adversarial environment*. Thus, by allowing the specification of a particular set of fault actions, our method in effect enables the synthesis of programs that are correct with respect to a *specified* set of possible environments, namely, those environments that generate exactly the faults given in the set F of fault-actions.

There has been considerable work on synthesis of programs that interact with an adversarial environment (usually called *reactive modules*). Pnueli and Rosner [1989a, 1989b] synthesized reactive modules that interact with an environment via an input variable x (that only the environment can modify) and an output variable y (that only the module can modify). In response to the environment modifying x, the module must modify y so that a given propositional linear time temporal logic formula $\phi(x, y)$ is satisfied (and so the execution of the module can be modeled as a two-player game). In the synthesis method of Anuchitanukul and Manna [1994], the environment and module take turns in selecting the next global state, according to some schedule. However, while this framework would accommodate a fault model, we argue as follows that, since it is based on propositional linear time temporal logic (PLTL), it is expressively inadequate for modeling faults. PLTL is suitable for expressing the correctness properties of fault-intolerant programs, where we are mainly interested in the properties that hold along *all* computation paths, but it cannot express the *possible* behavior of a process that has been affected by a fault. For instance, in the example of the mutual exclusion problem with fail-stop failures given in Section 6, the fault-coupling specification states that a fail-stopped process *may* stay down forever ($\mathsf{AG}(D_i \Rightarrow \mathsf{EG}D_i)$ in CTL). This simply cannot be written in PLTL. It therefore appears that the existential path quantifier of branching time temporal logic is essential in specifying the future behavior of failed processes. This objection to PLTL also applies to Pnueli and Rosner [1989a, 1989b], as well as to Wong-Toi and Dill [1990] who solved essentially the same problem as Pnueli and Rosner [1989a, 1989b] but in a somewhat more general setting.

Kupferman and Vardi [1997] presented a synthesis algorithm for reactive modules and CTL/CTL* specifications. Similarly to Pnueli and Rosner [1989a, 1989b], reactive modules communicate with the environment via input and output signals.[15] In the above approaches, the environment is "maximal" in the sense that its branching behavior includes all possibilities (i.e., all possible choices of inputs to present to the system) at each point. In Kupferman et al. [2000], the synthesis problem for "reactive environments" was addressed. A reactive environment can choose the set of possible inputs to present at each point based on the history of its interaction with the module up to that point. However, the method presented synthesizes a program that is correct with respect to all possible reactive environments. In general, this is too strong a criterion. Our method allows the set of possible environments to be specified, by means of specifying the set of fault-actions.

All of the above approaches have a time complexity at least exponential in the size of the temporal logic formula which gives the specification. Some, such as Kupferman et al. [2000], Pnueli and Rosner [1989a, 1989b], and Wong-Toi and Dill [1990], have a time complexity double exponential in formula size. Also, they specify the adversarial environment with a temporal logic formula, whereas our method uses nondeterministic actions to specify the adversarial (i.e., fault) behavior. We could have also used a CTL formula for this. However, in the worst case, the formula would be of a size at least linear in the number of the perturbed states generated by the fault-actions. This is because a fault-action can potentially affect all the atomic propositions. Hence, the result of applying a fault-action would have to be explicitly specified in the formula. The number of perturbed states in the worst case is exponential in the size of *spec*. Thus the formula size would be exponential in the size of *spec*. Including this formula in the labels of perturbed states and expanding these states would then

[15]There are also "unreadable inputs"—signals that the module cannot observe. Thus the setting is one of "incomplete information."

result in a fault-tolerant Kripke structure of a size double exponential in the size of *spec*. Thus, we feel that the use of actions rather than a temporal logic formula to specify the faults is more efficient.

Furthermore, our method allows one to give a characterization of all the environments that the synthesized program must be able to deal with, namely, the set of environments each of which generates the particular faults that we specify (the environments in this set can differ from each other in aspects unrelated to faults). In other words, we can specify a particular set of environments. All extant work on open systems synthesis or controller synthesis deals with either a single, "maximal" environment (which can engage in all possible moves), or with the set of all possible environments. Thus, our paper handles a more general setting.

Finally, we remark that the use of the EX modality of CTL usually provides sufficient expressiveness to deal with reactive environments. For example, in the mutual exclusion specification, one of the conjuncts is $\mathsf{AG}(N_i \Rightarrow (\mathsf{AX}_i T_i \wedge \mathsf{EX}_i T_i))$, $i = 1, 2$. This is logically equivalent to $\mathsf{AG}(N_i \Rightarrow \mathsf{AX}_i T_i) \wedge \mathsf{AG}(N_i \Rightarrow \mathsf{EX}_i T_i)$. The latter formula means that, when process P_i is in its N_i state, it can *always* move to its T_i state. We can think of the transition from N_i to T_i as representing an input request from the environment, or more specifically, from a "user" process U_i, which sends requests to P_i, which we can regard as a "server" process. Then, when P_i enters C_i, we interpret that as an output to U_i, granting it access to the critical resource. See Lynch [1996] for a nice discussion of this way of modeling mutual exclusion with a user environment. More generally, the use of EX allows us to specify general *input enabling conditions* [Lynch and Tuttle 1989]. These are sufficient to model a very wide class of reactive environments and systems, a few of which are: resource allocation [Lynch 1996; Welch and Lynch 1993], distributed data services [Fekete et al. 1999], group communication services [Fekete et al. 2001], distributed shared memory [Lynch and Shvartsman 2002; Luchangco 2001], and reliable multicast [Livadas and Lynch 2002].

In Liu and Joseph [1992], a manual method of transforming a given fault-intolerant shared-memory concurrent program into a fault-tolerant one is presented. Specifications are given in UNITY [Chandy and Misra 1988], and faults are specified as fault-actions. The recovery-actions are designed manually, and the paper presents proof rules for establishing that the recovery-actions guarantee the required fault-tolerance. There is also a manual method for refining both the program and the fault-actions, for example, to a lower level where the faults operate on specific hardware components. This line of research is continued in Liu and Joseph [1999], where the specification language is the Temporal Logic of Actions [Lamport 1994], and real-time properties are also dealt with. The synthesis method of Kulkarni and Arora [2000] mechanically transforms a fault-intolerant program into a fault-tolerant one. The fault-tolerant program satisfies every specification (in the absence of faults) that the fault-intolerant program does. A program is given as a transition relation over states, that is, corresponding to Kripke structures in this paper. A separate transformation is given for each of failsafe, nonmasking, and masking tolerance.

9.2 Conclusions and Further Work

We have presented a method for the automatic synthesis of fault-tolerant concurrent programs from specifications expressed in the temporal logic CTL. The user of our method only has to construct a formal specification as described in Section 3. Our method then automatically generates a program that satisfies the specification, if such a program exists. The method has single-exponential time complexity in the size of the problem specification plus the size of the problem-fault coupling specification. This complexity is essentially that of generating the state-transition diagram of the program to be synthesized. We have dealt with different fault classes (fail-stops, general state failures) and different types of tolerance (masking, nonmasking, fail-safe). We have also shown how impossibility results may be mechanically generated using our method.

By using a real-time extension of CTL [Emerson et al. 1993], our method should be able to deal with real-time properties. This may also require an annotation of fault-actions with timing information.

One potential difficulty with our method is the *state explosion problem*—the number of states in a structure usually increases exponentially with the number of processes, thereby restricting the applicability of synthesis methods based on state enumeration to small systems. In Attie and Emerson [1998] and Attie [1999], a method is proposed of overcoming the state explosion problem by considering the interaction of processes *pairwise*. The exponentially large global product of all the processes in the system is never constructed. Instead, using the CTL synthesis method of Emerson and Clarke [1982], small Kripke structures depicting the product of two processes are constructed and used as the basis for synthesis of concurrent programs consisting of arbitrarily many processes. Using the synthesis method given here instead of the Emerson and Clarke [1982] method, we can construct these "two-process structures," which will now incorporate fault-tolerant behavior. We then plan to extend the synthesis method of Attie and Emerson [1998] and Attie [1999] to take these structures as input and produce fault-tolerant programs of arbitrary size. We note that our method is efficient enough for small examples to be constructed by hand, as demonstrated in this paper. So, the application of our method to the synthesis of two-process structures should be feasible. This may not be true for other methods with higher time complexity (e.g., double exponential).

In Attie and Emerson [1996, 2001], we presented a method for synthesizing (fault-intolerant) programs for a model of concurrency in which every action is either an atomic read of a single variable, or an atomic write of a single variable. By integrating this with the synthesis method presented here, it will be possible to address the high atomicity limitation discussed in Section 8.1 above. A possible way of doing this is to first synthesize the fault-tolerant program, and then use the Attie and Emerson [1996, 2001] method to refine it to an atomic read/write program.

As we remarked in Attie [2000, 2002], the Byzantine consensus problem cannot be modeled in a shared memory model in which processes can access all of the shared variables, since a single Byzantine process can corrupt the entire memory, and therefore overwhelm any consensus algorithm. A shared memory model can be used, however, if the amount of memory that each process can access is limited, for example, by operating system level mechanisms such as access control matrices, access control lists, or capability lists [Tanenbaum 1987; Pfleeger 1989; Silberschatz and Galvin 1994]. The Attie and Emerson [1996, 2001] synthesis method can be adapted to enforce such access restrictions, since they are similar to the "single atomic write" restriction that limits a process to modifying only a single variable in any transition: the single atomic write restriction says "modify one variable and leave all the others unchanged," while the access control restriction says "leave the variables in this particular set unchanged." Combining our method with this adaptation of Attie and Emerson [1996, 2001] would enable the synthesis of low-atomicity concurrent programs that tolerate Byzantine faults.

Finally, combining both of the extensions discussed above would deal with state-explosion and grain of atomicity issues.

APPENDIX: COMPLETE PSEUDOCODE FOR THE SYNTHESIS ALGORITHM

```
(1) /* Construct the tableau $T_0 = (d_0, V_C^0, V_D^0, A_{CD}^0, A_{DC}^0, L^0)$ */
    Let $d_0$ be an OR-node with label $\{spec\}$;
    $T_0 := d_0$;
    **repeat** until $frontier(T_0) = \emptyset$
    (a) Select a node $e \in frontier(T_0)$;
    (b) **if** $\exists e' \in V_D^0 : L(e) = L(e')$ **then**
            merge $e$ and $e'$
        **else**
            attach all $e' \in Succ(e)$ as successors of $e$ and mark $e$ as expanded
        **endif**;
        Update $V_C^0, V_D^0, A_{CD}^0, A_{DC}^0$ appropriately.
    where the successors $Succ(e)$ of a node of either type are defined as follows:
    if $e$ is an OR-node, then $Succ(e) = Blocks(d)$, and if $e$ is an AND-node, then
    $Succ(e) = Tiles(e) \cup FaultStates(F, TOL, \{e\})$.
(2) /* Apply the deletion rules to $T_0$, resulting in $T_F$ */
    Repeatedly apply the deletion rules in Figure 2 to $T_0$ until no deletion rule
    is applicable. If $d_0$ is deleted, then return an impossibility result and halt.
    Otherwise, let $T_F$ be the tableau induced by the nodes that are still reach-
    able (via normal, fault, and recovery transitions) from $d_0$.
(3) /* Construct the fragment FFRAG[$c$] for every AND-node $c$, using the fault-
    free full subdags in $T_F$ */
    (a) Let FFRAG$_1$ be a copy of FDAG[$c, g_1$]. To obtain FFRAG$_{j+1}$ from
        FFRAG$_j$, do
        i. identify any two nodes on the frontier of FFRAG$_j$ that have the same
           label;
       ii. **forall** $s' \in frontier($FFRAG$_j)$ **do**
               /* let $c'$ be the AND-node in $T_F$ that $s'$ is a copy of */
               **if** $g_{j+1} \in L(s')$ **then**
                   attach a copy of FDAG[$c', g_{i+1}$] to FFRAG$_j$ at $s'$
        /* call the resulting directed acyclic graph FFRAG$_{j+1}$ */
    (b) Obtain FFRAG$'[c]$ from FFRAG$_m$ by identifying any two nodes in
        $frontier$(FFRAG$_m$) with the same label
    (c) To obtain FFRAG$[c]$ from FFRAG$'[c]$, do:
        i. **forall** AND-nodes $c'$ in FFRAG$'[c]$ and $a \in F$:
              **forall** $d$ such that $c' \xrightarrow{a,TOL} d$
                 attach a copy of at least one node $c'' \in Blocks(d)$ as
                 successor of $c'$;
                 label the transition from $c'$ to $c''$ as a fault-transition
(4) /* Construct the model $M_F$, using the fragments FFRAG$[c]$, for every AND-
    node $c$ */
    (a) Choose $c_0 \in Blocks(d_0)$ arbitrarily (recall that $d_0$ is the root of $T$);
        Let $M_1 =$ FFRAG$[c_0]$;
    (b) To obtain $M_{i+1}$ from $M_i$, do
        i. **forall** $s \in frontier(M_i)$ **do**
              /* let $c$ be the AND-node in $T_F$ that $s$ is a copy of */
              **if** there exists $s' \in interior(M_i)$ such that $s'$ is also a copy of $c$, and
           a copy of
                FFRAG$[c]$ is directly embedded in $M_i$ with root $s'$, **then**
                 identify $s$ and $s'$
              **else**
                 replace $s$ by a copy of FFRAG$[c]$
              **endif**
        /* call the resulting graph $M_{i+1}$ */
    (c) The construction halts with $i = N$ when $frontier(M_N)$ is empty. Let
        $M = M_N$. We write $M = (c_0, S, A, A_F, L)$, where $c_0$ is given in Step 4a
        (we write $c_0$ instead of $\{c_0\}$), $L$ is given by the labels of each node, $S$ is
        the set of all nodes in $M_N$, $A$ is the set of all transitions in $M_N$ that are
        labeled with a process index (i.e., normal or recovery transitions), and
        $A_F$ is the set of all transitions in $M_N$ that have label $(a, TOL)$ for some
        $a \in F$ (i.e., the fault transitions).
        Let $M_F = (c_0, S, A, A_F, L_0)$ where $L_0$ is $L$ restricted to the propositions
        occurring in $spec$. $M_F$ is a model of $spec$.
(5) /* Extract the fault-tolerant program from $M_F$ */
    (a) **for** every maximal set $\{s_1, \ldots, s_n\}$ of states in $M_F$ such that $L_0(s_1) =$
        $L_0(s_2) = \cdots = L_0(s_n)$
          i. introduce a new shared variable $x$
         ii. add the proposition $x = k$ to the label of $s_k$, $k \in [1:n]$
        iii. label each transition of $M_F$ that enters $s_k$ with the assignment $x := k$,
             for $k \in [1:n]$
    (b) **forall** transitions $s \xrightarrow{i,A} t$ in $M_F$
           add an arc to $P_i$ going from $s{\uparrow}i$ to $t{\uparrow}i$, and with the label $\wedge(L(s){\downarrow}i) \rightarrow A$
```

REFERENCES

ANUCHITANUKUL, A. AND MANNA, Z. 1994. Realizability and synthesis of reactive modules. In *Proceedings of the 6th International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 818. Springer-Verlag, Berlin, Germany, 156–169.

Arora, A. and Gouda, M. 1993. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Trans. Softw. Eng. 19*, 11, 1015–1027.

Arora, A. and Kulkarni, S. 1998. Component based design of multitolerant systems. *IEEE Trans. Softw. Eng. 24*, 1, 63–78.

Attie, P. 2000. Wait-free Byzantine agreement. Tech. Rep. NU-CCS-00-02. College of Computer Science, Northeastern University, Boston, MA. Available on-line at http://www.ccs.neu.edu/home/attie/pubs.html.

Attie, P. 2002. Wait-free byzantine consensus. *Inf. Process. Lett. 83*, 4 (Aug.), 221–227.

Attie, P. C. 1999. Synthesis of large concurrent programs via pairwise composition. In *CONCUR'99: 10th International Conference on Concurrency Theory* (Aalborg, Denmark). Lacture Notes in Computer Science, vol. 1664. Springer-Verlag, Berlin, Germany.

Attie, P. C. and Emerson, E. A. 1996. Synthesis of concurrent systems for an atomic read/atomic write model of computation (extended abstract). In *Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, PA). ACM Press, New York, NY, 111–120.

Attie, P. C. and Emerson, E. A. 1998. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst. 20*, 1 (Jan.), 51–115.

Attie, P. C. and Emerson, E. A. 2001. Synthesis of concurrent systems for an atomic read/write model of computation. *ACM Trans. Program. Lang. Syst. 23*, 2 (Mar.), 187–242. Extended abstract appears in *Proceedings of the ACM Symposium on Principles of Distributed Computing* (PODC), 1996.

Chandra, T. and Toueg, S. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (Mar.), 225–267.

Chandy, K. M. and Misra, J. 1988. *Parallel Program Design*. Addison-Wesley, Reading, MA.

Dijkstra, E. W. 1976. *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs, NJ.

Emerson, E. 1981. Branching time temporal logic and the design of correct concurrent programs. Ph.D. dessertation. Division of Applied Sciences, Harvard University, Cambridge, MA.

Emerson, E. A. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, J. V. Leeuwen, Ed. Vol. B, *Formal Models and Semantics*. The MIT Press/Elsevier, Cambridge, MA.

Emerson, E. A. and Clarke, E. M. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program. 2*, 241–266.

Emerson, E. A. and Lei, C. 1985. Modalities for model checking: Branching time logic strikes back. In *12'th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, LA). ACM Press, New York, NY, 84–96.

Emerson, E. A., Mok, A. K., Sistla, A. P., and Srinivasan, J. 1993. Quantitative temporal reasoning. *Real Time Syst. J. 2*, 331–352.

Emerson, E. A., Sadler, T. H., and Srinivasan, J. 1992. Efficient temporal satisfiability. *J. Logic Comput. 2*, 2, 173–210. Extended abstract appears in *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, 1989, pp. 166–178.

Fekete, A., Gupta, D., Luchangco, V., Lynch, N., and Shvartsman, A. 1999. Eventually-serializable data service. *Theor. Comput. Sci. 220*, 1 (June), 113–156. Special Issue on Distributed Algorithms.

Fekete, A., Lynch, N., and Shvartsman, A. 2001. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst. 19*, 2 (May), 171–216.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (April), 374–382.

Jayaram, M. and Varghese, G. 1996. Crash failures can drive protocols to arbitrary states. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing* (Philadelphia, PA). ACM Press, New York, NY.

Kulkarni, S. and Arora, A. 2000. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRT FT 2000, Pune, India, September 20–22, 2000*. Lecture Notes in Computer Science, vol. 1926. Springer-Verlag, Berlin, Germany.

Kupferman, O., Madhusudan, P., Thiagarajan, P., and Vardi, M. 2000. Open systems in reactive environments: Control and synthesis. In *Proceedings 11th International Conference on Concurrency Theory* (College Park, PA). Lecture Notes in Computer Science, vol. 1877. Springer-Verlag, Berlin, Germany, 92–107.

KUPFERMAN, O. AND VARDI, M. 1997. Synthesis with incomplete information. In *2nd International Conference on Temporal Logic*. Kluwer Academic Publishers, Manchester, U.K., 91–106.

LAMPORT, L. 1994. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May), 872–923.

LIU, Z. AND JOSEPH, M. 1992. Transformation of programs for fault-tolerance. *Form. Aspects Comput. 4*, 5, 442–469.

LIU, Z. AND JOSEPH, M. 1999. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Trans. Program. Lang. Syst. 21*, 1 (Jan.), 46–89.

LIVADAS, C. AND LYNCH, N. A. 2002. A formal venture into reliable multicast territory. In *Formal Techniques for Networked and Distributed Systems—FORTE 2002 (Proceedings of the 22nd IFIP WG 6.1 International Conference* (Houston, TX)), M. Y. V. Doron Peled, Ed. Lecture Notes in Computer Science, vol. 2529. Springer, Berlin, Germany, 146–161. Also, full version in Tech. Memo MIT-LCS-TR-868, MIT Laboratory for Computer Science, Cambridge, MA, November 2002.

LUCHANGCO, V. 2001. Memory consistency models for high performance distributed computing. Ph.D. dessertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

LYNCH, N. AND SHVARTSMAN, A. 2002. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Distributed Computing (Proceedings of the 16th International Symposium on DIStributed Computing (DISC)* (Toulouse, France)), D. Malkhi, Ed. Lecture Notes in Computer Science, vol. 2508. Springer-Verlag, Berlin, Germany, 173–190. Also, Tech. Rep. MIT-LCS-TR-856, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

LYNCH, N. A. 1996. *Distributed Algorithms*. Morgan-Kaufmann, San Francisco, CA.

LYNCH, N. A. AND TUTTLE, M. R. 1989. An introduction to input/output automata. *CWI-Quart. 2*, 3 (Sept.), 219–246. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Tech. Memo MIT/LCS/TM-373. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, November 1988. Also: Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 1987). ACM Press, New York, NY, 137–151.

MANNA, Z. AND WOLPER, P. 1984. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst. 6*, 1 (Jan.), 68–93. Also appears in *Proceedings of the Workshop on Logics of Programs, Yorktown-Heights, N.Y.* Lecture Notes in Computer Science vol. 131. Springer-Verlag, Berlin, Germany (1981).

MANOLIOS, P. AND TREFLER, R. 2001. Safety and liveness in branching time. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, CA. (Boston, MA).

PFLEEGER, C. 1989. *Security in Computing*. Prentice-Hall, Englewood Cliffs, NJ. USA.

PNUELI, A. AND ROSNER, R. 1989a. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 179–190.

PNUELI, A. AND ROSNER, R. 1989b. On the synthesis of asynchronous reactive modules. In *Proceedings of the 16th ICALP*. Lecture Notes in Computer Science, vol. 372. Springer-Verlag, Berlin, Germany, 652–671.

SCHNEIDER, F. 1984. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst. 2*, 2 (May), 145–154.

SCHNEIDER, F. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec.), 299–319.

SILBERSCHATZ, A. AND GALVIN, P. 1994. *Operating System Concepts*. Addison-Wesley, Reading, MA.

TANENBAUM, A. S. 1987. *Operating Systems, Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ.

WELCH, J. AND LYNCH, N. 1993. A modular Drinking Philosophers algorithm. *Distrib. Comput. 6*, 4 (July), 233–244.

WONG-TOI, H. AND DILL, D. 1990. Synthesizing processes and schedulers from temporal specifications. In *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 531. Springer-Verlag, Berlin, Germany, 272–281.

Received December 2001; revised November 2002; accepted April 2003