Synthesis of Fault-Tolerant Concurrent Programs

(Extended Abstract)

Anish ARORA[1]
Department of Computer Science
The Ohio State University
anish@cis.ohio-state.edu

Paul C. ATTIE[2]
School of Computer Science
Florida International University
attie@fiu.edu

E. Allen EMERSON[3]
Department of Computer Sciences
The University of Texas at Austin
emerson@cs.utexas.edu

Abstract

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 [EC82, MW84, PR89, PR89b, AM94]. 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, i.e., 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.

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 [EC82] and Manna and Wolper [MW84]. 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 [MW84] 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 the 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. This approach suggests a two-stage synthesis method, where an intolerant program is synthesized in the first stage (based on the method of [EC82]) and then transformed in the second stage so that its behavior in the presence of faults is as desired. The first stage applies the method of [EC82] to obtain a model for the problem specification (from which an intolerant program could be synthesized). The second stage models the presence of faults by adding *fault-transitions* to this model. These fault-transitions result in new, previously unreachable, states being added. We synthesize *recovery-transitions* from these *perturbed states* by applying the [EC82] synthesis method in each such state to generate the recovery transitions from that state.

Our method has the same order of complexity as the [EC82] synthesis method, i.e., exponential time in the size of the problem description. We illustrate our method by synthesizing fault-tolerant solutions for the mutual exclusion and barrier synchronization problems. Our method can also establish *impossibility results*: this occurs when the required recovery is not attainable in the presence of the specified faults.

The paper is organized as follows. Section 2 defines the model of computation, the specification language, and the fault models and fault-tolerance properties that we consider. It also provides an overview of the [EC82] synthesis method. Section 3 formally defines the synthesis problem for fault-tolerant concurrent programs. Section 4 presents our synthesis method, along with correctness results. Section 5 presents two examples: mutual exclusion and barrier synchronization, along with an illustration of an impossibility result. Section 6 discusses related work. Section 7 discusses the complexity of our method and compares it to that of other temporal logic synthesis methods. Section 8 presents our conclusions and discusses extensions of this work. For reasons of space, all proofs are omitted, and are provided in the full paper.

[1]Supported in part by NSA Grant CDA-9514898, NSF Grant CCR-93-08640, and OSU Grant 221506.
[2]Supported in part by NSF Grant CCR-9702616 and AFOSR Grant F49620-96-1-0221.
[3]Supported in part by NSF Grant CCR-9415496 and by SRC Contract DP 95-DP-388.

2 Preliminaries

2.1 Model of Parallel Computation

We consider nonterminating concurrent programs of the form $P = P_1 \| \cdots \| P_K$ which consist of a finite number of fixed sequential processes $P_1, \ldots, P_K$ 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 [Dij76]). A *global state* is a tuple of the form $(s_1, \ldots, s_K, x_1, \ldots, x_m)$ 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 parallelism 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 the current state is $s = [s_1, \ldots, s_i, \ldots, s_K, x_1, \ldots, x_m]$ and that P_i contains an arc from node s_i to s_i' labeled by the action $B \rightarrow A$. If B is true in the current state then a transition can be made to the next state $s' = [s_1, \ldots, s_i', \ldots, s_K, x_1', \ldots, x_m']$ 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 path 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 trees of the entire program $P_1 \| \cdots \| P_K$ meet 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 $\mathcal{AP}_1, \ldots, \mathcal{AP}_K$. $\mathcal{AP}_i$ contains the atomic propositions local to 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 $M = (S_0, S, R, L)$ consisting of a countable set S of global states, a set $S_0 \subseteq S$ of initial states, a binary transition relation $R \subseteq S \times S$, giving the transitions of every process, and a labeling function $L : S \mapsto 2^{\mathcal{AP}}$ which labels each state with the set of atomic propositions true in the state. R is partitioned into relations $R_1, \ldots, R_K$, where R_i gives the transitions of process i. We require that R be total, i.e., $\forall x \in S, \exists y : (x, y) \in R$. A *fullpath* is an infinite sequence of states $(s_0, s_1, s_2 \ldots)$ such that $\forall i : (s_i, s_{i+1}) \in R$. 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. We define $\models$ inductively:

$M, s_0 \models p$ iff $p \in V(s_0)$
$M, s_0 \models \neg f$ iff $\mathrm{not}(s_0 \models f)$
$M, s_0 \models f \wedge g$ iff $s_0 \models f$ and $s_0 \models g$
$M, s_0 \models EX_j f$ iff for some state t, $(s_0, t) \in R_j$ and $t \models f$,
$M, s_0 \models A[fUg]$ iff for all fullpaths $(s_0, s_1, \ldots)$ in M, $\exists i[\ i \geq 0$ and $s_i \models g$ and $\forall j (0 \leq j \wedge j < i \Rightarrow s_j \models f)\ ]$
$M, s_0 \models E[fUg]$ iff for some fullpath $(s_0, s_1, \ldots)$ in M, $\exists i[\ i \geq 0$ and $s_i \models g$ and $\forall j (0 \leq j \wedge j < i \Rightarrow s_j \models f)\ ]$

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 $\mathsf{A}[\mathit{true} \mathsf{U} f]$, $\mathsf{EF} f$ for $\mathsf{E}[\mathit{true} \mathsf{U} f]$, $\mathsf{A}[f \mathsf{W} g]$ 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 $\mathsf{A}[\mathit{false} \mathsf{W} f]$, $\mathsf{EG} f$ for $\mathsf{E}[\mathit{false} \mathsf{W} f]$, $\mathsf{AX}_i f$ 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 $\mathsf{AX}_1 f \wedge \cdots \wedge \mathsf{AX}_k f$.

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. 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.

We also define $\models_n$, a version of the $\models$ relation that is relativized to *fault-free fullpaths*.[4] $\models_n$ is defined with respect to a Kripke structure $M = (S_0, S, R, R_F, L)$, where S_0, S, R, L are as before, and $R_F \subseteq S \times S$ is a set of *fault-transitions*. A fault-free fullpath in M_F is a fullpath containing no transitions in $R_F - R$. 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."

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 synthesis method of [EC82] that this paper builds upon. We assume, in the sequel, that specifications are expressed in the form $\mathit{init{-}spec} \wedge \mathsf{AG}(\mathit{global{-}spec})$, where $\mathit{init{-}spec}$ contains only atomic propositions and boolean operators. $\mathit{init{-}spec}$ specifies the initial state, and $\mathit{global{-}spec}$ specifies correctness properties that are required to hold at all reachable states.

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

[4] The idea of relativized satisfaction comes from [EL85] where it is used to handle fairness in CTL model checking.

(0) Initial State (both processes are initially in their Noncritical region): $N_1 \wedge N_2$

(1) 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: $\mathsf{AG}(N_i \Rightarrow (\mathsf{AX}_i T_i \wedge \mathsf{EX}_i T_i))$

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

(3) 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: $\mathsf{AG}(C_i \Rightarrow (\mathsf{AX}_i N_i \wedge \mathsf{EX}_i N_i))$

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

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

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

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

(1–4) are *local structure* specifications; they describe the structure of each process.

2.3 Model of Faults

The faults that a concurrent program is subject to may be categorized in a variety of ways: (i) *Type,* e.g., the faults are stuck-at, fail-stop, crash, omission, timing, performance, or Byzantine. (ii) *Duration*, e.g., the faults are permanent, intermittent, or transient. (iii) *Observability*, e.g., the faults are detectable or not by the program. (iv) *Repair*, e.g. the faults are correctable or not by the program.

Towards 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. [AG93]). 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 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. 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 boolean variable $broken$ and the program action: $out \neq in \wedge \neg broken \rightarrow out := in$. If a fault occurs, the incorrect behavior of the wire 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: $\neg broken \rightarrow broken := true$.

We define the following classes of faults. One class is *fail-stop failures* [Sc84, Sc90]: A fault in this class stops a process from executing any actions, possibly forever. Thus, fail-stop failures effectively corrupt program processes in a detectable, uncorrectable, and potentially permanent manner. If some processes fail permanently, then the remaining processes can be thought of as a "subprogram" of the original program. Another class is *general state failures*: 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 failures are general in the sense that by a sequence of these faults the program may reach arbitrary global states [JV96]; thus, general state failures effectively corrupt global state in an undetectable, correctable, and transient manner.

2.4 Fault-Tolerance Properties

A tolerance property that a concurrent program satisfies in the presence of a class of faults may be categorized as *masking, nonmasking*, or *fail-safe* tolerance [AK98]. Intuitively, this classification is based upon how the problem specification is met in the presence of faults.

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

P **is** ***masking tolerant*** **to** F **for** $spec$ iff $\mathsf{AG}(global{-}spec)$ holds at all states reached in the presence of faults in F. That is, subsequent execution of P from these states satisfies the desired correctness properties of P.

P **is** ***nonmasking tolerant*** **to** F **for** $spec$ iff $\mathsf{AFAG}(global{-}spec)$ holds at all states reached in the presence of faults in F. 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** $spec$ iff $\mathsf{AG}(global{-}safety{-}spec)$ holds at all states reached in the presence of faults in F, where $global{-}safety{-}spec$ consists of all the safety properties in $global{-}spec$. 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.)

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 [AG93] 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.

2.5 Overview of the [EC82] Synthesis Method

The section summarizes the synthesis method of [EC82], upon which we build. This method constructs a *tableau*, which is a finite directed bipartite AND-OR graph. 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 set of CTL formulae, and no two AND-nodes (OR-nodes) have the same label. Intuitively, AND-nodes correspond to states in the final extracted model, and we will establish soundness of the method by showing that, in the final model, every state satisfies all the formulae in its label. OR-nodes correspond to "prestates." The OR-node successors of a given AND-node c (denoted $Tiles(c)$) are exactly the successors required to satisfy all the nexttime formulae in the label of c. In other words, if an AND-node c is labeled with n formulae of the form $\mathsf{AX}_i g$ and m formulae of the form $\mathsf{EX}_i h$, then c will have m successors associated with process i, each labeled with one of the EX_i-formulae and all of the AX_i-formulae. AND-nodes with no nexttime formulae in their label are given a single "dummy" successor labeled with the same formulae. If only EX-formulae are missing, then the formulae $\mathsf{EX}_1 \mathit{true}, \ldots, \mathsf{EX}_K \mathit{true}$ are added to the label, and the successors then computed as above. Each OR-node successor d is then "expanded" into a tree using the fixpoint characterization of the CTL modalities. 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 $\mathsf{AXAG}g$. The reader is referred to [Em81, EC82] for full details, where it is shown that this tree-construction process terminates. The final set of frontier nodes of the tree is taken to be the AND-node successors of d, and is denoted $Blocks(d)$. These AND-node successors 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 [EC82] method starts with a single OR-node labeled with the program specification, and repeatedly constructs successors of "frontier" nodes until there is no more change. It then applies a set of *deletion rules*, which, roughly speaking, remove all nodes that are propositionally inconsistent, or do not have enough successors, or are labeled with an eventuality formula which is not fulfilled. After the deletion rules are applied, and if the initial state is not deleted, a model can be extracted from the tableau by a process of "unraveling." Roughly speaking, for each AND-node c, 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 the tableau (i.e, $c' \rightarrow c''$ in FRAG[c] only if $c' \rightarrow d \rightarrow c''$ in the tableau 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]. Finally, the FRAG's are connected together (essentially, the frontier nodes of a FRAG are identified with root nodes of other FRAG's) to form a structure in which all eventualities are fulfilled. This structure is a model of the program specification. From the model, a program can be extracted by projecting onto the individual process indices.[5]

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 of the form $init{-}spec \wedge \mathsf{AG}(global{-}spec)$. 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 shared synchronization variables. We show how to remove this restriction in Appendix B below. 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***, which is a CTL formula $AG(coupling{-}spec)$. It relates the atomic propositions in the problem specification with those in the fault specification. It could, for instance, restrict the local states which a process can be in when an auxiliary proposition is true, or express that an auxiliary proposition cannot be written by any process. For example, a problem-fault coupling specification for the wire example of Section 2.3 with the stuck-at-low-voltage fault is $\mathsf{AG}(broken \Rightarrow \mathsf{AG}broken)$ $\wedge$ $\mathsf{AG}((broken \wedge \neg out) \Rightarrow \mathsf{AG}\neg out)$, i.e., once broken, always broken, and once broken and output is low, then output stays low forever.
4. A ***type of tolerance*** $TOL \in \{masking,\ fail{-}safe,\ nonmasking\}$, which specifies the desired tolerance property.

Required is to synthesize a concurrent program that (1) satisfies $init{-}spec \wedge \mathsf{AG}(global{-}spec)$ in the absence of faults, and (2) satisfies $\mathsf{AG}(coupling{-}spec)$ in the presence of faults, and (3) is TOL-tolerant to F for $init{-}spec \wedge \mathsf{AG}(global{-}spec)$.

4 The Synthesis Method

We first apply the tableau generation step of the [EC82] method outlined above to generate the tableau for $g = init{-}spec \wedge \mathsf{AG}(global{-}spec) \wedge \mathsf{AG}(coupling{-}spec)$. We then repeatedly apply the fault actions to the tableau, thereby generating perturbed states and fault-transitions. The perturbed states thus generated will have no successors in the tableau. For a given perturbed state d, we generate successors for d by considering d to be an OR-node and applying the [EC82] tableau-generation step outlined above, taking d to be the "root." If during this process a state is generated that has an identical label to an already occurring state, then the two states are identified. In effect, we use the tableau-generation step to construct, for each perturbed state, the necessary *recovery transitions.* This process must eventually terminate, as the number of possible perturbed states is finite. Finally, we apply a set of modified[6] deletion rules. From the resulting tableau, a model can be extracted by "unraveling," as in the [EC82] method, except that for every AND-node in a FRAG, we add all the perturbed states directly reached by executing a single fault-action. Finally, we extract the fault-tolerant program by projecting onto the individual process indices.

Figure 1 gives our overall method, and Figure 2 gives the procedure for expanding perturbed states.

[5] For example, Figure 4 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.

[6] To take fault-transitions into account.

1. Apply the tableau-based CTL decision procedure of [EC82] to the CTL formula $g = init\text{–}spec \wedge \mathsf{AG}(global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$. Let $T_0 = (d_0, V_C^0, V_D^0, A_{CD}^0, A_{DC}^0, L^0)$ be the resulting tableau. Mark all nodes in V_C^0, V_D^0 as expanded.
2. Repeat until $T_i = T_{i+1}$, where $T_i = (d_0, V_C^i, V_D^i, A_{CD}^i, A_{DC}^i, L^i)$ is the tableau after i iterations:
   (a) $V_D^{i+1} := V_D^i \cup \mathit{FaultStates}(F, \mathit{TOL}, V_C^i)$; $A_{CD}^{i+1} := A_{CD}^i \cup \mathit{FaultTrans}(F, \mathit{TOL}, V_C^i)$.
   (b) For some unmarked OR-node $d \in V_D^{i+1} - V_D^0$, invoke EXPAND(d) (Figure 2).
3. Let $T_N = (d_0, V_C^N, V_D^N, A_{CD}^N, A_{DC}^N, L^N)$ be the final tableau from step 2. Apply the deletion rules in Figure 3 to T_N. If d_0 is deleted, then return an impossibility result and halt. Otherwise, let T_F be the result of applying the deletion rules to T_N and then removing all unreachable nodes. Extract a model M_F from T_F as follows.
   (a) For each AND-node c in T_F
      i. Construct FRAG[c] as in [EC82], i.e., ignore all fault-transitions.
      ii. $\text{FRAG}_F[c] := \text{FRAG}[c] \cup \mathit{FaultTrans}(F, \mathit{TOL}, V)$, where V is the set of AND-nodes in FRAG[c].
   (b) Construct $M_F = (s_0, S, R, R_F, L)$ as in [EC82] but use FRAG_F's instead of FRAG's.
   (c) Introduce shared variables into M_F to distinguish propositionally identical states (as in [EC82]).
4. Extract the fault-tolerant program from M_F by projecting onto the individual process indices.

Figure 1: The synthesis method.

Finally, we give the following technical definitions.

$c \xrightarrow{f,TOL} d$ iff $c(f.guard) = true$ and $L(d) = f.body(L(c)\!\uparrow\!\mathcal{AP}) \cup \mathit{Label}_{TOL}(spec)$ for fault action f.

Here $f.guard$ is the guard of fault-action f and $f.body$ is its body (recall that f is given as a guarded command). $L(d)$ is the label of d. Occurrence of fault f in AND-node (i.e., state) c leads to OR-node d. The propositional component of $L(d)$ results from applying $f.body$ to the propositions in $L(c)$, while the temporal component $\mathit{Label}_{TOL}(spec)$ of $L(d)$ is defined solely by the problem specification and the desired type of fault-tolerance:

- If TOL = masking, then $\mathit{Label}_{TOL}(spec) = \mathsf{AG}(global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$. For masking tolerance, the global specification must hold in all perturbed states.
- If TOL = non-masking, then $\mathit{Label}_{TOL}(spec) = \mathsf{AFAG}(global\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$. For non-masking tolerance, the global specification must eventually hold in all computations starting in perturbed states (provided that faults stop occurring).
- If TOL = fail-safe, then $\mathit{Label}_{TOL}(spec) = \mathsf{AG}(global\text{–}safety\text{–}spec) \wedge \mathsf{AG}(coupling\text{–}spec)$. For fail-safe tolerance, the safety component of the global specification must hold in all perturbed states. Note that recovery to states where *global–spec* holds is not required.

1. $T_d := \{d\}$. /* T_d is the tableau portion containing all recovery-transitions from d. */
2. Repeat until all frontier nodes of T_d are expanded
   (a) Select an unexpanded frontier node e.
   (b) If $\exists e' \in V_D^{i+1}(L(e) = L(e'))$ then merge e and e', else attach all $e' \in \mathit{Succ}(e)$ as successors of e and mark e as expanded. Update V_C^{i+1}, V_D^{i+1}, A_{CD}^{i+1}, A_{DC}^{i+1} appropriately.

Figure 2: Procedure EXPAND(d).

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 (including fault-successors) is already deleted.[7]

DeleteAU 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''.

DeleteEU 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', $g \in L(c'')$.

Figure 3: The deletion rules.

$\mathit{FaultStates}(F, \mathit{TOL}, V) = \{d \mid \exists f \in F, \exists c \in V : c \xrightarrow{F,TOL} d\}$. Returns the set of OR-nodes reached by executing a single fault action in some AND-node of V.

$\mathit{FaultTrans}(F, \mathit{TOL}, V) = \{(c, F, d) \mid \exists f \in F,\ \exists c \in V : c \xrightarrow{F,TOL} d\}$. Returns the fault-transitions generated by executing a single fault action in some AND-node of V.

If e is an OR-node, then define $\mathit{Succ}(e) = \mathit{Blocks}(e)$, and if e is an AND-node, then define $\mathit{Succ}(e) = \mathit{Tiles}(e) \cup \mathit{FaultStates}(F, \mathit{TOL}, \{e\})$.

Definition 1 (*Full Subdag*). *A* full subdag *D rooted at node e in T_N is a directed acyclic subgraph of T_N satisfying all of the following conditions*

1. *Node e is the unique node from which all other nodes are reachable,*

[7] This is where our deletion rules differ from those of [EC82].

2. *For every AND-node c in D, if c has any sons in D, then every non-fault successor of c in T_N is a son of c in D,*
3. *For every OR-node d, there exists precisely one AND-node c in T_N such that c is a son of d in D.*

4.1 Soundness and Completeness of the Method

Let $\models_n$ denote the $\models$ relation of CTL when path quantification is restricted to fault-free fullpaths. Due to lack of space, we only present the final soundness and completeness results. $M_F = (s_0, S, R, R_F, L)$ is the fault-tolerant model produced by our method.

Soundness means that all formulae in the label of a state hold in that state. Thus, by virtue of the way that state labels are computed (Sections 2.5 and 4) the resulting program is a solution of the synthesis problem.

Theorem 1 (*Soundness*).
For all $s \in S$, $g \in L(s)$, $M_F, s \models_n g$.

Definition 2 (*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 fulfilled for e' in D.*

A fault-subgraph rooted at node e is the fault-tolerant analogue of an infinite tree-like model rooted at e—it certifies that e's label $L(e)$ is satisfiable despite the occurrence of faults. Completeness then means that a node e is deleted iff $L(e)$ is unsatisfiable in the presence of faults.

Theorem 2 (*Completeness*). *Let e be a node that is deleted at some point in our method. Then there does not exist a fault-subgraph rooted at e.*

Fault-closure means that all the faults given in the fault specification are included in the final model.

Theorem 3 (*Fault-closure*). *For all $s \in S$, $f \in F$ such that $s(f.guard) = true$, there exists $t \in S$ such that $s \xrightarrow{f} t \in R_F$.*

5 Examples

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 P_i, the auxiliary proposition D_i (denoting P_i is "down") and two fault actions: one that truthifies D_i and falsifies all other propositions of P_i (denoting the fail-stop of P_i), and the other that truthifies exactly one of N_i, T_i, or C_i and falsifies all other propositions of P_i (denoting the repair of P_i).[8] 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: $\mathsf{AG}(D_i \Rightarrow \neg(N_i \vee T_i \vee C_i))$

(2) A fail-stopped process may stay down forever:
$\mathsf{AG}(D_i \Rightarrow \mathsf{EG} D_i)$

Finally, the type of fault-tolerance we require is masking.

Figure 4 shows a Kripke structure for the above problem and the process P_1 extracted from it (the second process, P_2, is omitted for space reasons and can be obtained from P_1 by interchanging the process indices 1 and 2). The portion of the Kripke structure above the dark horizontal line is the model for the mutual exclusion specification that is produced by the CTL decision procedure of [EC82] (and from which a fault-intolerant program could be extracted). The entire Kripke structure is 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 (in P_1, a similar convention is used). 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.

Our second example is the barrier synchronization problem subject to general state failures. The barrier synchronization specification is given in appendix A. 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 non-masking.

Figure 5 gives the model generated by our synthesis method for this problem. (The model for the barrier synchronization specification that is produced by the synthesis method of [EC82] is obtained by removing the four perturbed states and all incident transitions.) For clarity, the fault-transitions are omitted. Also shown is the process P_1 extracted from the model (again P_2 is omitted and can be obtained from P_1 by interchanging process indices 1 and 2). Note that the tolerance of the extracted program is a special case of non-masking, namely self-stabilizing [AG93].

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 <u>two</u> 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 <u>not</u> permit the fault-tolerant program to generate any new states or transitions under normal (fault-free) operation.

---

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

5.1 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 $[s1_1e1_2]$. 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.

6 Related Work

A fault model may be considered as a particular type of *adversarial environment*. There has been considerable work on synthesis of programs that interact with an adversarial environment (usually called *reactive modules*). [PR89, PR89b] synthesize 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 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). The environment however, cannot modify the internal state of the module, and so this framework cannot model faults. The synthesis method of [AM94] overcomes this limitation; 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 5, 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 [PR89, PR89b], as well as [WD90] which solves essentially the same problem as [PR89, PR89b] but in a somewhat more general setting.

Finally, [KV97] presents a synthesis algorithm for reactive modules and CTL/CTL* specifications. Similarly to [PR89, PR89b], reactive modules communicate with the environment via input and output signals.[9] Since the environment cannot modify the internal state of the module, [KV97] is not directly applicable to the synthesis problem addressed in this paper. Nevertheless, since it does deal with branching time temporal logic, it might be possible to adapt or generalize the approach to deal with faults.

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

7 Complexity of the Method

Since the number of possible nodes in T_N is at most exponential in the size of the specification, it is easy to verify that our method has single exponential time and space complexity in the size of the specification. This compares favorably to the methods of [PR89, PR89b, WD90], all of which have double exponential time and space complexity in the size of the specification. The method of [KV97] also has single exponential time and space complexity.

8 Conclusions and Further Work

We have presented a method for the synthesis of fault-tolerant programs from specifications expressed in temporal logic. The method has single-exponential time complexity in the size of the problem specification plus the size of the problem-fault coupling specification. 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.

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 [AE98], 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 [EC82], small Kripke structures depicting the product of two processes are constructed and used as the basis for synthesis of programs with arbitrarily many processes. Using the synthesis method given here instead of the [EC82] method, we can construct these "two-process structures," which will now incorporate fault-tolerant behavior. We then plan to extend the synthesis method of [AE98] to take these structures as input and produce fault tolerant programs of arbitrary size.

We plan to integrate our method with the synthesis method of [AE96] for synthesizing atomic read/write programs. A possible approach is to first synthesize the fault-tolerant program, and then use the [AE96] method to refine it to an atomic read/write program. We are also currently extending our method to deal with byzantine failure.

References

[AE98] P.C. Attie and E.A. Emerson, "Synthesis of concurrent systems with many similar processes," *ACM TOPLAS*, 20(1): 51–115 (1998).

[AE96] P.C. Attie and E.A. Emerson, "Synthesis of concurrent systems for an atomic read / atomic write model of computation (extended abstract)," *Proc. 15'th ACM PODC*, Philadelphia, May 1996, 111–120.

[AG93] A. Arora and M. G. Gouda, "Closure and convergence: a foundation of fault-tolerant computing," *IEEE TSE*, 19(11): 1015–1027 (1993).

[AG94] A. Arora and M. G. Gouda, "Distributed reset," *IEEE TOC*, 43(9): 1026–1038 (1994).

[AK98] A. Arora and S. Kulkarni, "Component based design of multitolerance," *IEEE TSE* 24(1): 63–78.

[AM94] A. Anuchitanukul and Z. Manna, "Realizability and synthesis of reactive modules," *CAV 94*, Springer LNCS 818, 156–169.

[Dij76] E.W. Dijkstra, **A discipline of programming**, Prentice-Hall Inc., 1976.

[Em81] E.A. Emerson, "Branching time temporal logic and the design of correct concurrent programs," Ph. D. dissertation, Division of Applied Sciences, Harvard University, 1981.

[EC82] E.A. Emerson and E.M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Science of Computer Programming* 2 (1982) 241–266.

[EL85] E.A. Emerson and C.L.Lei, "Modalities for model checking: branching time strikes back," *12'th ACM POPL.*

[EMSS93] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan, "Quantitative temporal reasoning," *Real Time Systems Journal* 2: 331–352 (January 1993).

[JV96] M. Jayaram and G. Varghese, "Crash failures can drive protocols to arbitrary states," *Proc. 15'th ACM PODC.*

[KV97] O. Kupferman and M. Vardi, "Synthesis with incomplete information," *ICTL 97.*

[MW84] Z. Manna and P. Wolper, "Synthesis of communicating processes from temporal logic specifications," *ACM TOPLAS*, 6 (1984) 68–93.

[Pe81] G.L. Peterson, "Myths about the mutual exclusion problem," *IPL*, 12 (1981) 115–116.

[PR89] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," *Proc. 16'th ACM POPL*, (1989).

[PR89b] A. Pnueli and R. Rosner, "On the synthesis of asynchronous reactive modules," *Proc. 16th ICALP*, Springer LNCS 372, (1989), 652–671.

[Sc84] F.B. Schneider, "Byzantine generals in action: implementing fail-stop processors," *ACM TOCS* 2 (2): 145–154.

[Sc90] F.B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, 22 (4): 299–319.

[WD90] H. Wong-Toi and D.L. Dill, "Synthesizing processes and schedulers from temporal specifications," *CAV 90*, Springer LNCS 531, 272–281.

A The Barrier Synchronization Specification

Each process consists of a cyclic sequence of two terminating phases, phase 1 and phase 2. Process $i, i = 1, 2$, is in exactly one of 4 local states, $s1_i$, $e1_i$, $s2_i$, $e2_i$, corresponding to the start of phase 1, the end of phase 1, the start of phase 2, and the end of phase 2, respectively:

(0) Initial State (both processes are initially at the start of phase 1): $s1_1 \wedge s1_2$

(1) The start of phase 1 is always followed by the end of phase 1: $\mathsf{AG}(s1_i \Rightarrow \mathsf{AX}_i e1_i)$

(2) The end of phase 1 is always followed by the start of phase 2: $\mathsf{AG}(e1_i \Rightarrow \mathsf{AX}_i s2_i)$

(3) The start of phase 2 is always followed by the end of phase 2: $\mathsf{AG}(s2_i \Rightarrow \mathsf{AX}_i e2_i)$

(4) The end of phase 2 is always followed by the start of phase 1: $\mathsf{AG}(e2_i \Rightarrow \mathsf{AX}_i s1_i)$

(5) P_i is always in exactly one of the states $s1_i$, $e1_i$, $s2_i$, $e2_i$: $\mathsf{AG}(s1_i \equiv \neg(e1_i \vee s2_i \vee e2_i)) \wedge \mathsf{AG}(e1_i \equiv \neg(s1_i \vee s2_i \vee e2_i)) \wedge \mathsf{AG}(s2_i \equiv \neg(s1_i \vee e1_i \vee e2_i)) \wedge \mathsf{AG}(e1_i \equiv \neg(s1_i \vee s2_i \vee e2_i))$

(6) The processes are never simultaneously at the start of different phases: $\mathsf{AG}\neg(s1_1 \wedge s2_2) \wedge \mathsf{AG}\neg(s1_2 \wedge s2_1)$

(7) The processes are never simultaneously at the end of different phases: $\mathsf{AG}\neg(e1_1 \wedge e2_2) \wedge \mathsf{AG}\neg(e1_2 \wedge e2_1)$

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

B Allowing Faults to Corrupt Shared Synchronization Variables

The [EC82] method introduces shared variables to disambiguate all propositionally identical states with different labels. If this is not done, the extracted program will exhibit the same future behavior from all such states, even though they have different labels, thus allowing pending eventualities to remain unfulfilled.[10] We have assumed up to now that fault actions cannot reference these 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 $[1 : n]$.[11] We note that, by construction of the [EC82] 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.[12] Bearing this in mind, suppose the occurence 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 $[1 : n]$, e.g., as 1.[13] 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.

[10] For example, not introducing a shared variable in the mutual exclusion example in Figure 4 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 $\mathsf{AG}(T_2 \Rightarrow \mathsf{AF}C_2)$.

[11] We use the notation $[m : n]$ to denote the integers from m up to n inclusive.

[12] In fact, the [EC82] method does not even record the value of x in states outside $\bar{t}$.

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

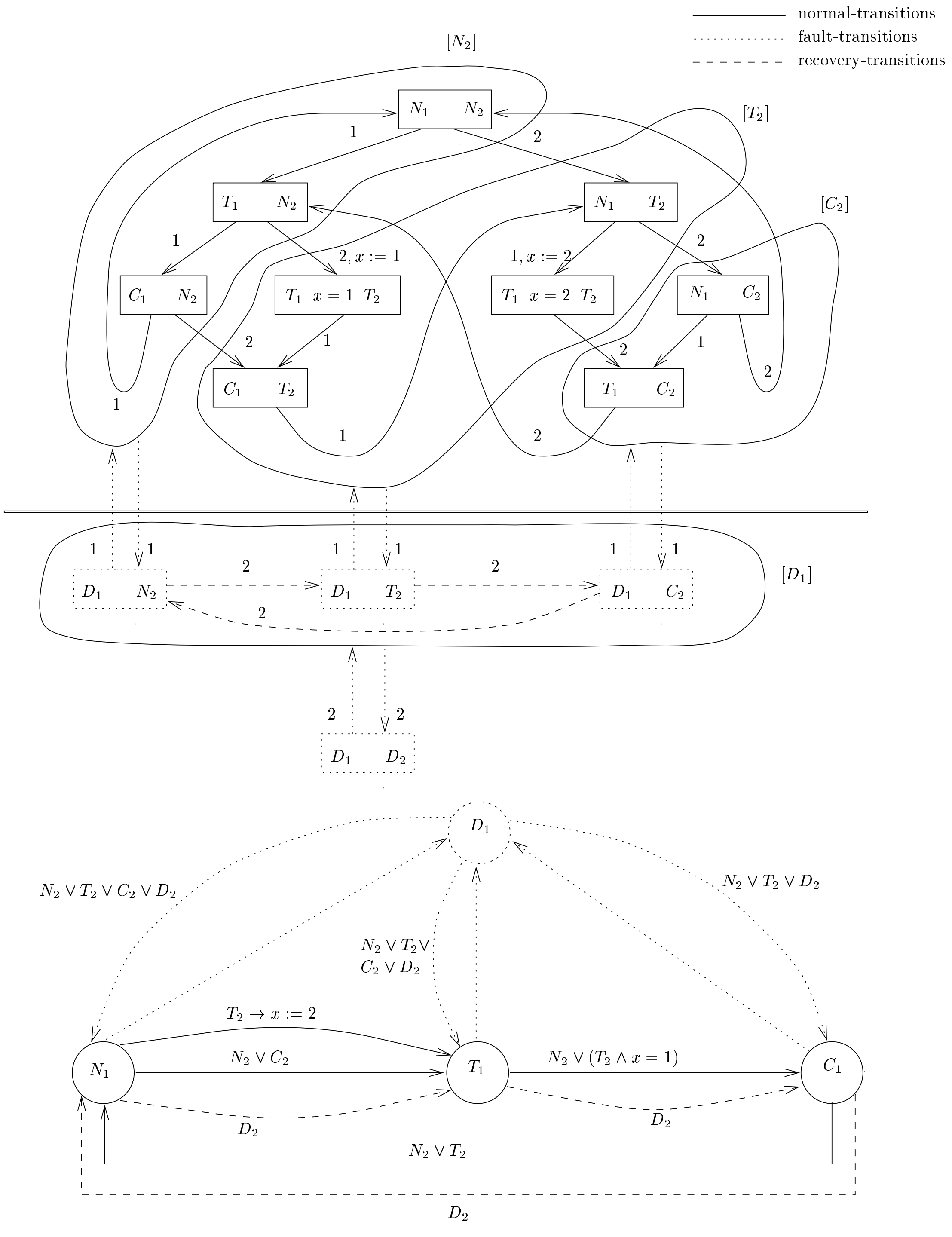
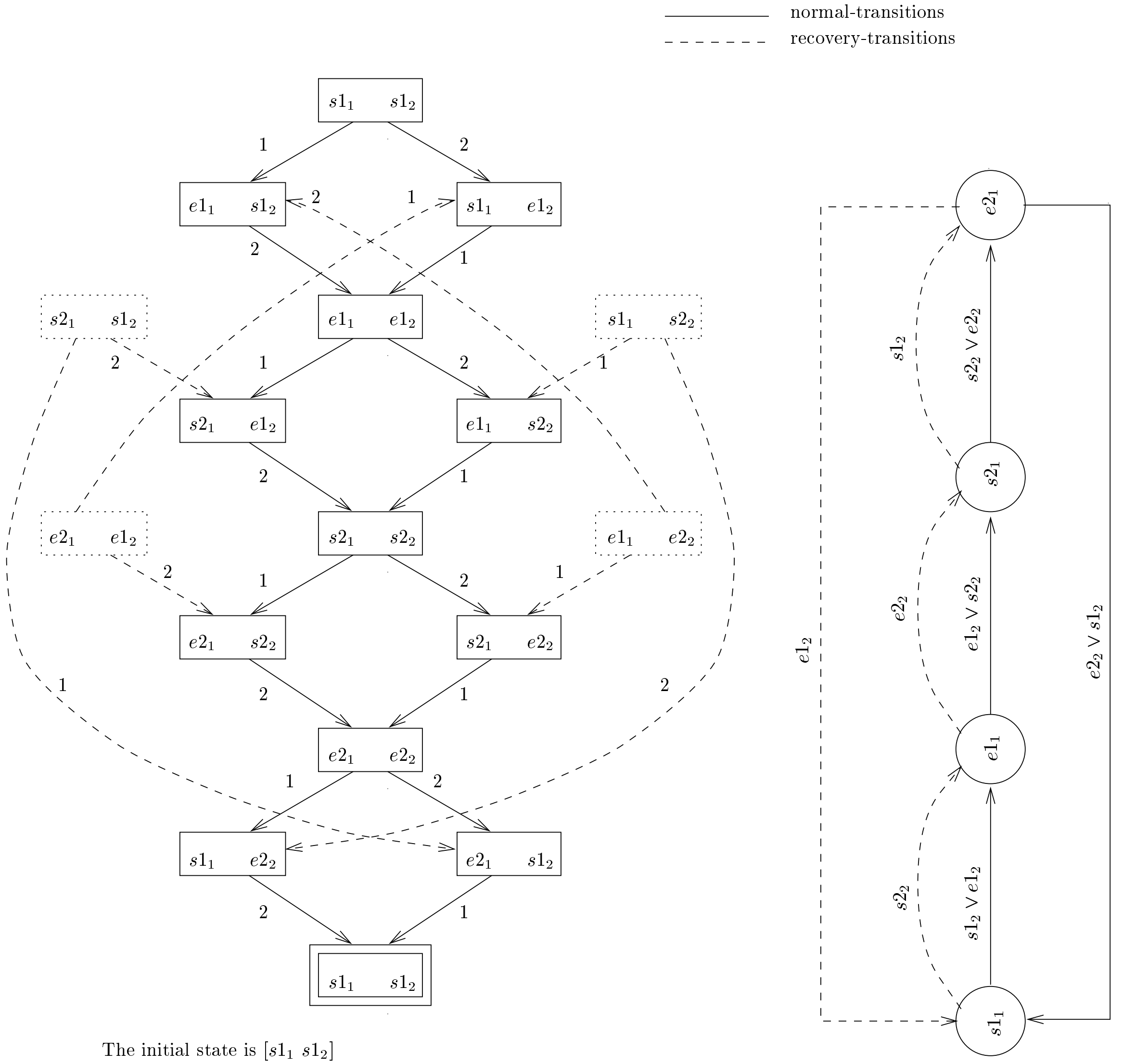


Figure 4: Two-process mutual exclusion structure for the fail-stop failures model (above), and process P_1 extracted from the structure (below).



The initial state is $[s1_1\ s1_2]$
The fault-states are $[s2_1\ s1_2]$, $[s1_1\ s2_2]$, $[e2_1\ e1_2]$, $[e1_1\ e2_2]$
The double boxed state at the bottom is the initial state, it is repeated for clarity of the figure

Figure 5: Barrier synchronization structure for the general state failures model (above), and process P_1 extracted from the structure (below).