Synthesizing Masking Fault-Tolerant Systems from Deontic Specifications

Ramiro Demasi[1], Pablo F. Castro[2,3], Thomas S.E. Maibaum[1], and Nazareno Aguirre[2,3]

[1] Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, demasira@mcmaster.ca,tom@maibaum.org
[2] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina, {pcastro,naguirre}@dc.exa.unrc.edu.ar
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract. In this paper, we study the problem of synthesizing fault-tolerant components from specifications, i.e., the problem of automatically constructing a fault-tolerant component implementation from a logical specification of the component, and the system's required level of fault-tolerance. We study a specific level of fault-tolerance: masking tolerance. A system exhibits masking tolerance when both the liveness and the safety properties of the behaviors of the system are preserved under the occurrence of faults. In our approach, the logical specification of components is given in dCTL, a branching time temporal logic with deontic operators, especially designed for fault-tolerant component specification. The synthesis algorithm takes the component specification, and automatically determines whether a component with masking fault-tolerance is realizable, and the maximal set of faults supported for this level of tolerance. Our technique for synthesis is based on capturing masking fault-tolerance via a simulation relation. Furthermore, a combination of an extension of a synthesis algorithm for CTL to cope with dCTL specifications, with simulation algorithms, is defined in order to synthesize masking fault-tolerant implementations.

Keywords: Formal specification, Fault-tolerance, Program synthesis, Temporal logics, Deontic logics, Correctness by construction

1 Introduction

The increasing demand for highly dependable and constantly available systems has focused attention on providing strong guarantees for software correctness, in particular, for safety critical systems. In this context, a problem that deserves attention is that of capturing *faults*, understood as unexpected events that affect a system, as well as expressing and reasoning about the properties of systems in the presence of faults. Indeed, various researchers have been concerned with formally expressing and reasoning about fault-tolerant behavior, and some formalisms and tools have been proposed for this task [13, 4]. Moreover, in formal approaches to fault-tolerance (and in general in formal approaches to software

development), it is generally recognized that powerful (semi-)automated analysis techniques are essential for a method to be effectively applicable in practice. Therefore, tools for automated or semi-automated reasoning, such as those based on model checking or automated theorem proving, have been central in many of the above cited works. In this direction, but with less emphasis, approaches for automatically *synthesizing* programs, in particular fault-tolerant ones, have also been studied [1, 12, 2, 5].

In this paper, we study the problem of automatically synthesizing fault-tolerant systems from logical specifications. This work concentrates on a particular kind of fault-tolerance, namely masking tolerance. As stated in [10], the fault tolerance that a system may exhibit can be classified using the liveness and safety properties that the designers want the system to preserve. In masking fault-tolerance, the system must preserve, in the presence of faults, both the safety and the liveness properties of the "fault free" system. More precisely, masking fault-tolerance is usually stated with respect to an observable part of the system or component, its so called *interface*. Essentially, a masking fault-tolerant system ensures that faults are not observable at the system's interface level, and both liveness and safety properties of the system are preserved even when subject to the occurrence of faults.

Our work is strongly related to the approach presented in [2]. The main difference between our approach and that introduced in [2] is that, to specify systems, we use dCTL-, a branching time temporal logic with deontic operators (see Section 2), as opposed to the well established branching time temporal logic CTL used in [2]. More precisely, the logic dCTL- features, besides the CTL temporal operators, deontic operators that allow us to declaratively distinguish the normative (correct, without faults) part of the system from its non-normative (faulty) part. In particular, in our approach faults are declaratively embedded in the logical specification. In our approach, faults are understood as violations to the deontic obligations on the behavior of the system, in contrast with the case of [2] and related works (e.g., [1, 12, 9]), where faults are given explicitly as part of the behavior model of the system. This leads to some differences in the way in which programs are synthesized. While in [2] a satisfiability algorithm for CTL based on tableau is employed, with faults corresponding to adding states in the tableau, we use instead the deontic specification to produce the faulty states combined with a characterization of masking fault tolerance by means of a simulation relation, in order to "cut out" inappropriate parts of the tableau. Our algorithm then combines a simulation algorithm with an adaptation of tableau based CTL satisfiability to deal with dCTL- specifications. The algorithm is presented in detail in Section 3.

There are some interesting properties that our proposed algorithm enjoys. If the algorithm is able to compute a masking fault tolerant implementation from the system specification, then the implementation produced is *maximal* in the sense that it "removes" the least number of states necessary to achieve the required tolerance. If the algorithm does not compute an implementation, then there are no feasible masking fault tolerant implementations for the specified

system. We show that our algorithm is sound, and that it holds the above mentioned kind of strong completeness (it returns a program that is maximal with respect to masking similarity, which as explained later on, implies completeness).

The remainder of the paper is structured as follows. In Section 2 we introduce some notions used throughout the paper. In Section 3, we describe our synthesis method. A practical case study is shown in Section 4. Section 5 reviews some related work. Finally, we discuss some conclusions and directions for further work.

2 Preliminaries

In this section we introduce some concepts that will be necessary throughout the paper. For the sake of brevity, we assume some basic knowledge of model checking; the interested reader may consult [3]. We model fault-tolerant systems by means of *colored Kripke structures*, as introduced in [6]. Given a set of propositional letters $AP = \{p, q, s, \dots\}$, a *colored Kripke structure* is a 5-tuple $\langle S, I, R, L, \mathcal{N}\rangle$, where S is a set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, $L : S \to \wp(AP)$ is a labeling function indicating which propositions are true in each state, and $\mathcal{N} \subseteq S$ is a set of *normal*, or "green" states. The complement of $\mathcal{N}$ is the set of "red", abnormal or faulty states. Arcs leading to abnormal states (i.e., states not in $\mathcal{N}$) can be thought of as faulty transitions, or simply *faults*. Then, normal executions are those transiting only through green states. The set of normal executions is denoted by $\mathcal{NT}$. We assume that in every colored Kripke structure, and for every normal state, there exists at least one successor state that is also normal, and that at least one initial state is green. This guarantees that every system has at least one normal execution, i.e., that $\mathcal{NT} \neq \emptyset$.

As is usual in the definition of temporal operators, we employ the notion of trace. Given a colored Kripke structure $M = \langle S, I, R, L, \mathcal{N}\rangle$, a *trace* is a maximal sequence of states, whose consecutive pairs of states are adjacent with respect to R. When a trace of M starts in an initial state, it is called an *execution* of M, with *partial* executions corresponding to non-maximal sequences of adjacent states starting in an initial state. Given a trace $\sigma = s_0, s_1, s_2, s_3, \dots$, the ith state of σ is denoted by $\sigma[i]$, and the final segment of σ starting in position i is denoted by $\sigma[i..]$. Moreover, we distinguish among the different kinds of outgoing transitions from a state. We denote by $\dashrightarrow$ the restriction of R to faulty transitions, and $\to$ the restriction of R to non-faulty transitions. We define $Post_N(s) = \{s \in S |\ s \to s'\}$ as the set of (immediate) successors of s reachable via non-faulty (or good) transitions; similarly, $Post_F(s) = \{s \in S |\ s \dashrightarrow s'\}$ represents the set of successors of s reachable via faulty arcs. Analogously, we define $Pre_N(s')$ and $Pre_F(s')$ as the set of (immediate) predecessors of s' via normal and faulty transitions, respectively. Moreover, $Post^*(s)$ denotes the states which are reachable from s. Without loss of generality, we assume that every state has a successor [3]. We denote by $\Rightarrow^*$ the transitive closure of $\dashrightarrow \cup \to$.

In order to state properties of systems, we use a fragment of dCTL [6], a branching time temporal logic with deontic operators designed for fault-tolerant system verification. Formulas in this fragment, that we call dCTL-, refer to properties of behaviors of colored Kripke structures, in which a distinction between *normal* and *abnormal* states (and therefore also a distinction between normal and abnormal traces) is made. The logic dCTL is defined over the Computation Tree Logic CTL, with its novel part being the deontic operators $\mathbf{O}(\psi)$ (obligation) and $\mathbf{P}(\psi)$ (permission), which are applied to a certain kind of path formula ψ. The intention of these operators is to capture the notion of *obligation* and *permission* over traces. Intuitively, these operators have the following meaning:

- $\mathbf{O}(\psi)$: property ψ is obliged in every future state, reachable via non-faulty transitions.
- $\mathbf{P}(\psi)$: there exists a normal execution, i.e., not involving faults, starting from the current state and along which ψ holds.

Obligation and permission will enable us to express intended properties which should hold in *all* normal behaviors and *some* normal behaviors, respectively. These deontic operators have an implicit *temporal* character, since ψ is a path formula. Let us present the syntax of dCTL-. Let $AP = \{p_0, p_1, \dots\}$ be a set of atomic propositions. The sets Φ and Ψ of *state formulas* and *path formulas*, respectively, are mutually recursively defined as follows:

$$\Phi ::= p_i \mid \neg\Phi \mid \Phi \rightarrow \Phi \mid \mathsf{A}(\Psi) \mid \mathsf{E}(\Psi) \mid \mathbf{O}(\Psi) \mid \mathbf{P}(\Psi)$$
$$\Psi ::= \mathsf{X}\Phi \mid \Phi\, \mathcal{U}\, \Phi \mid \Phi\, \mathcal{W}\, \Phi$$

Other boolean connectives (here, state operators), such as $\wedge$, $\vee$, etc., are defined as usual. Also, traditional temporal operators G and F can be expressed as $\mathsf{G}(\phi) \equiv \phi\ \mathcal{W}\ \bot$, and $\mathsf{F}(\phi) \equiv \top\ \mathcal{U}\ \phi$. The standard boolean operators and the CTL quantifiers A and E have the usual semantics. Now, we formally state the semantics of the logic. We start by defining the relation $\vDash$, formalizing the satisfaction of dCTL- state formulas in colored Kripke structures. For the deontic operators, the definition of $\vDash$ is as follows:

- $M, s \vDash \mathbf{O}(\psi) \Leftrightarrow$ for every $\sigma \in \mathcal{NT}$ such that $\sigma[0] = s$, we have that, for every $i \geq 0$, $M, \sigma[i..] \vDash \psi$.
- $M, s \vDash \mathbf{P}(\psi) \Leftrightarrow$ for some $\sigma \in \mathcal{NT}$ such that $\sigma[0] = s$, we have that, for every $i \geq 0$, $M, \sigma[i..] \vDash \psi$.

For the standard CTL operators, the definition of $\vDash$ is as usual (cf. [3]). We denote by $M \vDash \varphi$ the fact that $M, s \vDash \varphi$ holds for every state s of M, and by $\vDash \varphi$ the fact that $M \vDash \varphi$ for every colored Kripke structure M. Furthermore, the α and β classification of formulas given in [2] for tableau can be extended to our setting. For CTL operators, this is done as in [2]. For the deontic operators we proceed as follows:

- $\mathbf{O}(\varphi\, \mathcal{U}\, \psi)$ is classified as a β formula. In this case: $\beta_1 = O_\psi$ and $\beta_2 = O_\varphi \wedge \mathsf{AX}\mathbf{O}(\varphi\, \mathcal{U}\, \psi)$, where O_φ is obtained by substituting in φ any propositional variable p by a fresh variable O_p, and similarly for O_ψ.

- $\mathbf{P}(\varphi\ \mathcal{U}\ \psi)$ is classified as a β formula. In this case: $\beta_1 = O_\psi$ and $\beta_2 = O_\varphi \wedge \mathsf{EXO}(\varphi\ \mathcal{U}\ \psi)$, where O_φ and O_ψ are defined as before.
- $\mathbf{O}(\varphi\ \mathcal{W}\ \psi)$ is classified as a β formula. In this case: $\beta_1 = O_\psi$ and $\beta_2 = O_\varphi \wedge \mathsf{AXO}(\varphi\ \mathcal{W}\ \psi)$, where O_φ and O_ψ are defined as before.
- $\mathbf{P}(\varphi\ \mathcal{W}\ \psi)$ is classified as a β formula. In this case: $\beta_1 = O_\psi$ and $\beta_2 = O_\varphi \wedge \mathsf{EXO}(\varphi\ \mathcal{W}\ \psi)$, where O_φ and O_ψ are defined as before.

This classification of formulas will be essential for the tableau proofs and synthesis algorithm, presented later on in this paper.

Fault-tolerance in characterized in our work via simulation relations. Various detailed notions of fault-tolerance, namely masking, nonmasking and failsafe tolerances, are all defined via appropriate simulation relations, relating a specification of the system (i.e., its fault-free expected behavior) with the fault-tolerant implementation [8]. In this paper, we concentrate on masking fault-tolerance, although synthesis mechanisms for other kinds of fault-tolerance, definable via appropriate simulation relations, are relatively direct. In order to make the paper self contained, let us reproduce here the definition of masking simulation.

Definition 1. *(Masking fault-tolerance) Given two colored Kripke structures* $M = \langle S, I, R, L, \mathcal{N}\rangle$ *and* $M' = \langle S', I', R', L', \mathcal{N}'\rangle$*, we say that a relationship* $\prec_{Mask} \subseteq S \times S'$ *is* masking fault-tolerant *for sublabelings* $L_0 \subseteq L$ *and* $L'_0 \subseteq L'$ *iff:*

(A) $\forall s_1 \in I : (\exists s_2 \in I' : s_1 \prec_{Mask} s_2)$ *and* $\forall s_2 \in I' : (\exists s_1 \in I : s_1 \prec_{Mask} s_2)$.
(B) for all $s_1 \prec_{Mask} s_2$ *the following holds:*
  (1) $L_0(s_1) = L'_0(s_2)$.
  (2) if $s'_1 \in Post_N(s_1)$*, then there exists* $s'_2 \in Post(s_2)$ *with* $s'_1 \prec_{Mask} s'_2$.
  (3) if $s'_2 \in Post_N(s_2)$*, then there exists* $s'_1 \in Post_N(s_1)$ *with* $s'_1 \prec_{Mask} s'_2$.
  (4) if $s'_2 \in Post_F(s_2)$*, then either there exists* $s'_1 \in Post_N(s_1)$ *with* $s'_1 \prec_{Mask} s'_2$ *or* $s_1 \prec_{Mask} s'_2$.

Notice that Definition 1 makes use of a sublabeling $L_0 \subseteq L$, whose intention is to capture the *observable* part of the state, that visible from the component's interface. Our approach is in this sense *state based*, as opposed to *event based* approaches where the interface is captured via observable actions/events. Masking fault-tolerance corresponds to the kind of fault-tolerance that completely "masks" faults, not allowing them or their consequences to be observable. Masking fault-tolerance must then preserve both safety and liveness properties of the "fault free" system. For further details, we refer the interested reader to [8].

In the next section, we use the fault-tolerance simulation relation in combination with a CTL synthesis algorithm (extended to cope with dCTL-) in order to automatically construct, from a logical system description, a masking fault-tolerant system. The resulting system is maximal with respect to masking tolerance, in the sense that it "cuts out" the minimal part of the system augmented with faults, to make the resulting program masking tolerant. The synthesized program may not support all original faults, or support faults only when they occur in certain situations, but it is in a sense the most general solution: the "removed" transitions are those that would lead to nonmasking system conditions.

3 The Synthesis Approach

Given a dCTL- specification of a component and a desired level of fault-tolerance (in this case, masking), our goal is to automatically obtain a fault-tolerant component, with the required level of fault-tolerance. Masking fault-tolerance requires the system augmented with faults to preserve the observable behavior of the fault-free system, in what concerns both safety and liveness properties. The interface, in our case, is captured by a subset of the state variables (i.e., a state sublabeling L_0). The dCTL- system specification involves the use of CTL to describe the system declaratively (including safety and liveness properties of the system), while the deontic operators of dCTL- allow us to capture *obligations*, and to indirectly characterize faults as events violating these obligations. Notice that the deontic specification states what the expected behavior of the system is, and, indirectly, what the possible faults are. In other words, the possible faults are not explicitly given, as in other approaches, but stated at the specification level. We compare our approach with related work in Section 5.

The synthesis process is based on the extraction of a finite behavior model from a dCTL- specification. This is achieved by constructing a behavior model that captures the system augmented with faults, and then combining a synthesis algorithm for dCTL- with a simulation relation that captures masking tolerance, in order to remove from this model those states and faults that lie outside the required level of tolerance, i.e., that cannot be masked. The synthesis algorithm aims at detecting the *maximal* set of faults that can be tolerated (for the required level of fault-tolerance), and returning a (maximal) program that provides recovery from these faults. Of course, if the resulting system can only deal with the empty set of faults, then no masking fault-tolerant program is possible, from the provided specification.

In this paper, we are concerned with the synthesis of a single component. The approach can be extended to extract several concurrent components from a specification, by using indexes as done in [2]. We leave this as further work. More precisely, the problem of synthesis of a fault-tolerant component has as input a problem specification, a dCTL- formula `problem-spec` of the form `init-spec` $\wedge$ `normal-spec`, where `init-spec` and `normal-spec` can be any dCTL- formula. From this description, we want to automatically obtain a system that satisfies `init-spec` $\wedge$ `normal-spec`, while being masking tolerant with respect to the maximal set of faults obtained from violations of the system obligations.

3.1 The Synthesis Algorithm for Masking Tolerance

Our synthesis algorithm has three phases. The pseudocode of the algorithm is shown in Figures 1, 2, 3, and 4. It starts by building a tableau (Figure 1), following the tableau based algorithm for CTL satisfiability. We employ the rules α and β both for CTL and deontic formulas. That is, we construct a graph $T_N = (d, V_C, V_D, A_{CD}, V_{DC}, L)$, where V_C are called *And*-nodes and V_D are called *Or*-nodes. The rules used for building the graph (involving also deontic formulas) allow us to obtain the sub-formulas of the original specification. We stop when all

Algorithm 1 Construction of tableau $T_N = (d, V_C, V_D, A_{CD}, V_{DC}, L)$

Require: deontic specification *dSpec*: `init-spec` and `normal-spec`
Ensure: Tableau T_N

```
 1: Let d be an Or-node with label {dSpec}
 2: T_N := d
 3: repeat
 4:    Select a node e ∈ frontier(T_N)
 5:    if ∃ e' ∈ V_D with L(e) = L(e') then
 6:       merge e and e'
 7:    else
 8:       for all e' ∈ Succ(e) being an And-node do
 9:          if e' is non-faulty then
10:             Norm := Norm ∪ {e'}
11:          else
12:             if ∃e'' ∈ Succ(e) faulty such that NForm(e') = NForm(e'') then
13:                delete(e")
14:             end if
15:          end if
16:       end for
17:       attach all e' ∈ Succ(e) as successors of e and mark e as expanded
18:    end if
19:    update V_C, V_D, A_CD, V_DC appropriately
20: until frontier(T_N) = ∅
21: Apply the deletion rules to T_N
22: Apply Algorithm 2 to check nodes in Norm
23: Apply Algorithm 3 to remove and create faulty nodes
24: Apply Algorithm 4 to check the relation of masking and remove nodes.
25: return T_N
```

the *frontier* nodes are generated (nodes where no rules can be applied). We then start applying the deletion rules explained in [2], in order to remove inconsistent nodes and nodes containing eventuality formulas that cannot be satisfied. When this process finishes, we obtain a graph similar to that obtained by the tableau method for CTL satisfiability. Regarding deontic operations, we modified the CTL algorithm to cope with these (recall our classification of deontic operators as α and β). *Or*-nodes are expanded following the traditional rules. On the other hand, when a new *And*-node (say x) is created, we check if there is some violation, i.e., if either $O_p \in x$ and $p \notin x$, or $O_{\neg p} \in x$ and $p \in x$, belong to the node. If this is the case, the node is considered faulty (proposition O_p is undestood as: *p should be true*, and when p is false we get a state in which the normal or desirable behavior is not fulfilled). Otherwise, the node is added to $Norm$, the set of normal (non-faulty) states (line 10 of Alg. 1). If there is a faulty node (say e') such that it has the same CTL formulas as a non-faulty node (say e''), e' is deleted, since it is masked by e'' (line 13).

Secondly, the algorithm enters a phase where nodes originating from the specification of the system, that cannot fulfil deontic eventualities, are searched

Algorithm 2 Alg. for computing faulty states

Require: Tableau generated by alg. 1
Ensure: All faulty states are identified and removed from $Norm$

1: **repeat**
2: **if** $\exists x \in V_c$ s.t. $\mathbf{O}(p\,\mathcal{U}\,q)$ and $\exists v : v \in Norm \wedge q \notin v \wedge x \rightarrow^* v$ **then**
3: $Norm := Norm \backslash \{x\}$
4: **end if**
5: **if** $\exists x \in V_c$ s.t. $\mathbf{P}(p\,\mathcal{U}\,q)$ and $\forall v : v \in Norm \wedge q \notin v \wedge x \rightarrow^* v$ **then**
6: $Norm := Norm \backslash \{x\}$
7: **end if**
8: **until** $Norm$ does not change

for. These nodes are marked as faulty, and their treatment is shown in Alg. 2. Thirdly, we inject *faults*. We take each non-faulty state (i.e., each *And*-node) and produce a copy of it which is an *Or*-node. But in order to distinguish it from the other kinds of nodes, we call these *FOr*-nodes (faulty *Or*-nodes). The process for dealing with these states is in Alg. 3. This algorithm consists of an adaptation of the backward simulation algorithm shown in [3]. We use it in order to only generate the nodes that can be masked, and cut out the remaining ones. The generation is performed by applying the indicated operations. If a faulty node that is not masked by any normal state is created, then we move "upwards" in the graph, to appropriately prune the graph to get rid of this unmasked state. After that, since all the faulty nodes that can be masked were generated, we check condition $B.2$ of the masking simulation relation. This step may also lead to cutting out further faulty nodes, namely those which exhibit normal behavior, but that are not part of the correct behavior of the system. Finally, the synthesized program is extracted from the generated tableau. The extraction is as follows. First, a Kripke structure M is obtained from the tableau by unfolding the tableau as explained in [2]. Then, we delete the non propositional formulas from the nodes, and we add new propositional variables to distinguish nodes that have the same formulas, to avoid erroneously collapsing nodes (these extra variables can be seen as variables that indicate different phases of the algorithm; they play the same same role as the shared variables of the algorithm for synthesis introduced in [2]). Then, each transition $s \rightarrow t$ is labeled with the command $A \rightarrow B$ iff A is the conjunction of all variables occurring in s and the negation of those variables that do not appear in s (we assume that the finite alphabet of propositional variables is given). We add $b := \neg b$ if b changes its value from s to t. An example of this is shown in the next section.

Let us state two important properties of the synthesis algorithm, whose proofs are sketched following the proofs of correctness given for the algorithms for CTL satisfiability based on tableau [7, 2], and for checking (bi)simulations [3, 11].

Theorem 1. *Given a specification S over a set AP of propositional letters, if we obtain a program P by applying the synthesis algorithm over the sublabeling obtained from $AP' \subseteq AP$, then P is a masking tolerant implementation of S, i.e., $P \prec_{Mask} P$ (with respect to AP') and $P \vDash S$.*

Algorithm 3 Computes relations satisfying $B.3$ and $B.4$ of Def. 1

Require: Tableau Generated by SAT
Ensure: $Masks$ and $RemoveL$ satisfy conditions $B.3$ and $B.4$.

for all s_2 **do**
  $Masks(s_2) := \{s_1 \in Norm \mid L_0(s_1) = L_0(s_2)\}$
  $RemoveL(s_2) := Norm \backslash Pre_N(Masks(s_2))$ {Note that all the nodes in $Norm$ are already generated}
end for
while $\exists\ s_2' \in S \backslash Norm$ with $RemoveL(s_2') \neq \emptyset$ or there is a unexpanded v Or-node **do**
  select s_2' such that $RemoveL(s_2') \neq \emptyset$ or s_2' in $faultySucc(v)$
  for all $s_1 \in RemoveL(s_2')$ **do**
    for all $s_2 \in Pre_N(s_2')$ **do**
      if $s_1 \in Masks(s_2)$ **then**
        $Masks(s_2) := Masks(s_2) \backslash \{s_1\}$
        for all $s \in Pre_N(s_1)$ with $Post_N(s) \cap Masks(s_2) = \emptyset \wedge s \notin Masks(s_2)$ **do**
          $RemoveL(s_2) := RemoveL(s_2) \cup \{s\}$
        end for
      end if
    end for(* this takes care of the faulty transitions*)
    for all $s_2 \in Pre_F(s_2')$ **do**
      if $s_1 \in Masks(s_2) \wedge s_1 \notin Masks(s_2')$ **then**
        $Masks(s_2) := Masks(s_2) \backslash \{s_1\}$
        if $Masks(s_2) = \emptyset$ **then**
          delete $DAG[s_2]$
          $removeL(s_2) := \emptyset$
        else
          for all $s \in Pre_N(s_1)$ with $Post_N(s) \cap Masks(s_2) = \emptyset$ **do**
            $RemoveL(s_2) := RemoveL(s_2) \cup \{s\}$
          end for
        end if
      end if
    end for
  end for
  $RemoveL(s_2') := \emptyset$ and all the FOr-nodes are expanded
end while

Sketch of Proof. First, we prove that Alg. 3 ensures conditions $B.3$ and $B.4$ of Def. 1. Then we prove that Alg. 4 ensures condition $B.2$. Notice that, when Alg. 2 starts, all the normative nodes ($Norm$) have been computed. Then, we have the following invariant of Alg. 3: *(i)* $RemoveL(s_2) = Norm \backslash Pre_N(Masks(s_2))$; *(ii)* for any relation $\prec_{Mask}$: $\{s_1 \in Norm \mid s_1 \prec_{Mask} s_2\} \subseteq Masks(s_2) \subseteq \{s_2 \in Norm \mid L(s_1) = L(s_2)\}$; and *(iii)* $\forall s_2 \in Masks(s_1)$, either:

- $\exists s_1' \in Post(s_1)$ with $Post_N(s_2) \cap Masks(s_1') = \emptyset \wedge s_1' \notin Masks(s_1)$ and $s_2 \in RemoveL(s_1)$,
- $\forall s_1' \in Post(s_1) : Post_N(s_2) \cap Masks(s_1') = \emptyset$

From the last item we obtain that, when $RemoveL(s'_1) = \emptyset$ for every s'_1, then: $\forall s_1 \in S : \forall s_2 \in Masks(s_1) : \forall s'_1 \in Post(s_1) : Post_N(s_2) \cap Masks(s'_1) \neq \emptyset \vee s_2 \in Masks(s'_1)$. That is, the relation defined as $s_1 \prec_{Mask} s_2$ holds item $B.3$ and $B.4$ of Def. 1.

On the other hand, notice that before executing Alg. 3, all the faulty nodes have been calculated. For this algorithm we have the following invariant: *(i)* $RemoveR(s_2) = S \backslash Pre(Masked(s_2))$; *(ii)* for any relation $\prec_{Mask}$: $\{s_2 \in V_C \mid s_1 \prec_{Mask} s_2\} \subseteq Masked(s_1) \subseteq \{s_2 \in V_c \mid L(s_1) = L(s_2)\}$; and *(iii)* $\forall s_2 \in Masked(s_1)$, either:

- $\exists s'_1 \in Post(s_1)$ with $Post(s_2) \cap Masked(s'_1) = \emptyset$ and $s_2 \in RemoveL(s_2)$,
- $\forall s'_1 \in Post_N(s_1) : Post(s_2) \cap Masked(s'_1) = \emptyset$

That is, when $RemoveR(s_1) = \emptyset$, then we have $\forall s_1 \in S : \forall s_2 \in Masked(s_1) : \forall s'_1 \in Post_N(s_1) : Post(s_2) \cap Masked(s'_1) \neq \emptyset$. Thus, the relation defined as: $s_1 \prec_{Mask} s_2$ iff $s_1 \in Masks(s_2) \wedge s_2 \in Masked(s_1)$ satisfies condition $B.2$ of Def. 1. Since it also satisfies $B.3$ and $B.4$, it is a masking relation. The proof that the obtained structure satisfies the specification can be obtained, for CTL operators, following the proof given in [2]. For the deontic operators notice that all the nodes that do not satisfy the deontic operators are marked as faulty, ensuring that the safety deontic formulas are preserved. We treat deontic eventualities by marking as faulty all the nodes that have unfulfilled deontic eventualities. Thus, both CTL and deontic formulas are satisfied. Termination can be proved by resorting to the approach for proving termination of simulation algorithms (cf. [3]). The only point to note is that the injection of faults finishes at some point since states start repeating.

The definition of masking similarity ensures that the safety and liveness properties of the normal behavior of P are preserved in the presence of faults. If the synthesized program P contains no faults, we conclude that is not possible to synthesize a masking tolerant program supporting faults, from the specification. Moreover, we can prove that the synthesized program is the most general.

Theorem 2. *Given a specification S, if a structure M is obtained by the synthesis algorithm, then for any other structure $M' \vDash S$ such that it is masking and the non-faulty part of M' coincides with that of M, then we have $M' \prec M$, where $\prec$ is the usual notion of simulation with respect to L_0.*

Sketch of Proof. The simulation relation is defined as: *(i)* if $s \in Norm$, $s' \prec s$ iff $s' \prec_{Mask} s$; *(ii)* if $s \notin Norm$, $s' \prec s$ iff $Masked(s') \subseteq Masked(s)$, i.e., the M' nodes masked for s' are a subset of those masked by s in M. In order to prove that this relation is a simulation, assume $s \prec t$. If $s \rightarrow s'$ and $s' \in Post_N(s)$, by condition $B.3$ of Def. 1 we obtain that there is a $t \rightarrow t'$ such that $s' \prec t'$. Otherwise, if $s \rightarrow s'$ and $s' \in Post_F(S)$ and s is normative, then the transition matches some part of the specification. Thus, a similar transition is in M and therefore we have $t \rightarrow t'$. Now if s' masks any node, the same node has to be masked by t' (otherwise M' would not be masking). Thus, $s' \prec t'$. A similar reasoning can be used when s is faulty.

Algorithm 4 Computes relations that satisfy condition $B.2$ of Def. 1

Require: Colored Kripke structure M
Ensure: Relations $Masked$ and $RemoveR$ satisfy condition $B.2$ of Def. 1

for all $s_2 \in \mathcal{F}$ **do**
  $Masked(s_2) := \{s_1 \mid s_2 \in Masks(s_1)\}$
  $RemoveR(s_2) := S \backslash Pre(Masked(s_2))$ {Note that all the faulty and normal states are already generated}
end for
while $\exists\, s_2' \in \mathcal{F}$ with $RemoveR(s_2') \neq \emptyset$ **do**
  select s_2' such that $RemoveR(s_2') \neq \emptyset$
  for all $s_1 \in RemoveR(s_2')$ **do**
    for all $s_2 \in Pre(s_2')$ **do**
      if $s_1 \in Masked(s_2)$ **then**
        $Masked(s_2) := Masked(s_2) \backslash \{s_1\}$
        if $Masks(s_1) \backslash \{s_2\} = \emptyset$ **then**
          delete $DAGG(s_2)$
        else
          for all $s \in Pre(s_1)$ with $Post(s) \cap Masked(s_2) = \emptyset$ **do**
            $RemoveR(s_2) := RemoveR(s_2) \cup \{s\}$
          end for
        end if
      end if
    end for
  end for
  $RemoveR(s_2') := \emptyset$
end while

Since the CTL algorithm is complete, if some structure that satisfies the CTL specification exists, then the algorithm produces it, and by Theorem 2 we obtain a program that is masking, and preserves as many faulty states as possible. That is, as a corollary of Theorem 2, the synthesis algorithm is *complete*.

4 An Example: A Memory Cell

Let us consider a memory cell that stores a bit of information and supports reading and writing. A state in this system maintains the current value of the cell ($m = i$, for $i = 0, 1$), writing allows one to change this value, and reading returns the stored value. Evidently, in this system the result of a read operation depends on the value stored in the cell. Some potential faults occur when a bit's value (say 1) unexpectedly loses its charge and it turns into another value (say 0). *Redundancy* can be employed to deal with this situation, using for instance three memory bits instead of one. Also, a variable v, that indicates the value that the user wants to write (i.e., $v = 0$, $v = 1$ or $v = \perp$, the latter being the case in which the system is "idle" with respect to writing) is added to the model.

Writing operations are performed simultaneously on the three bits, whereas a reading returns the value that is repeated at least twice in the memory bits.

Each state in the model is described by variables r_i and w_i which record the last writing operation performed and the actual reading in the state. Each state also has three bits, described by boolean variables c_0, c_1 and c_2. The requirements on this system ($\texttt{init-spec} \wedge \texttt{normal-spec}$) can be specified in dCTL-, as follows:

(1) $c_0 \leftrightarrow c_1 \wedge c_0 \leftrightarrow c_2$. In the initial state the three bits contain the same value.
(2) $\mathbf{O}((c_0 \wedge c_1 \wedge c_2) \vee (\neg c_0 \wedge \neg c_1 \wedge \neg c_2))$. A safety property of the system: the three bits should coincide.
(3) $\mathbf{O}((r_0 \rightarrow w_0) \wedge (r_1 \rightarrow w_1))$. The value read from the cell ought to coincide with the last writing performed.
(4) $\mathsf{AG}(w_0 \equiv \neg w_1)$. If a zero has been written, then w_1 is false and vice versa.
(5) $\mathsf{AG}(w_0 \,\mathcal{U}\, w_1) \wedge (w_1 \,\mathcal{U}\, w_0)$. Variable w_1 only changes when w_0 becomes true, and vice versa.
(6) $\mathsf{AG}(r_0 \equiv (\neg c_0 \wedge \neg c_1) \vee (\neg c_0 \wedge \neg c_2) \vee (\neg c_1 \wedge \neg c_2))$. The reading of a 0 corresponds to the value read in the majority.
(7) $\mathsf{AG}(r_1 \equiv (c_0 \wedge c_1) \vee (c_0 \wedge c_2) \vee (c_1 \wedge c_2))$. The reading of a 1 corresponds to the value read in the majority.
(8) $\mathsf{AG}(v = 1 \rightarrow \mathsf{AX}(w_1 \wedge v = \bot \wedge c_0 \wedge c_1 \wedge c_2))$. If the user wants to write 1, then in the next step the memory will be setup to one.
(9) $\mathsf{AG}(v = 0 \rightarrow \mathsf{AX}(w_0 \wedge v = \bot \wedge \neg c_0 \wedge \neg c_1 \wedge \neg c_2))$. Similar to the previous, but for 0.
(10) $\mathsf{AG}(v = \bot \rightarrow \mathsf{AX}(v = 1 \vee v = 0 \vee v = \bot))$. At any moment the user may decide to write a value.

Besides these formulas, one may add additional constraints, e.g., indicating that atomic steps (including faults) change bits by one. These constraints are straightforward to capture in CTL.

Let us now illustrate how our synthesis approach works on this example. Fig. 1 shows the partial tableau generated by Alg. 1 for this problem. *And*-nodes and *Or*-nodes are shown as rectangles and rounded corner rectangles, respectively. For the sake of brevity, we put only the relevant information inside each box. Initially, a tableau is built using Alg. 1, employing the rules α and β for CTL and dCTL- formulas until every node in the tableau has at least one successor. The tableau contains a fault injection part, generated from the *And*-node in the second level of the tableau. This *FOr*-node is labeled identically as its *And*-node predecessor. From this *FOr*-node we generate all possible faults from deontic formula violations. Particularly, this node has O_{c_0}, O_{c_1}, and O_{c_2}, deontic propositional variables, expressing that c_0, c_1, and c_2 should be true there, which is the case in this node. Now, we start to consider those cases in which an obligation might be violated. Following Alg. 2, we negate one-by-one these deontic propositional variables and check on-the-fly whether it is possible to mask these faulty states using Alg. 3. We generated three faulty *And*-nodes (for the sake of brevity, just two of them are drawn) from the *FOr*-node with similar information to it except for the new negated propositional variable. The first *FAnd*-node successor introduces $\neg c_0$ violating O_{c_0}. The second and third *FAnd*-nodes introduce $\neg c_1$ and $\neg c_2$ violating O_{c_1} and O_{c_2}, respectively. Every time a new faulty *FAnd*-node is created, we check whether it can be masked. For the

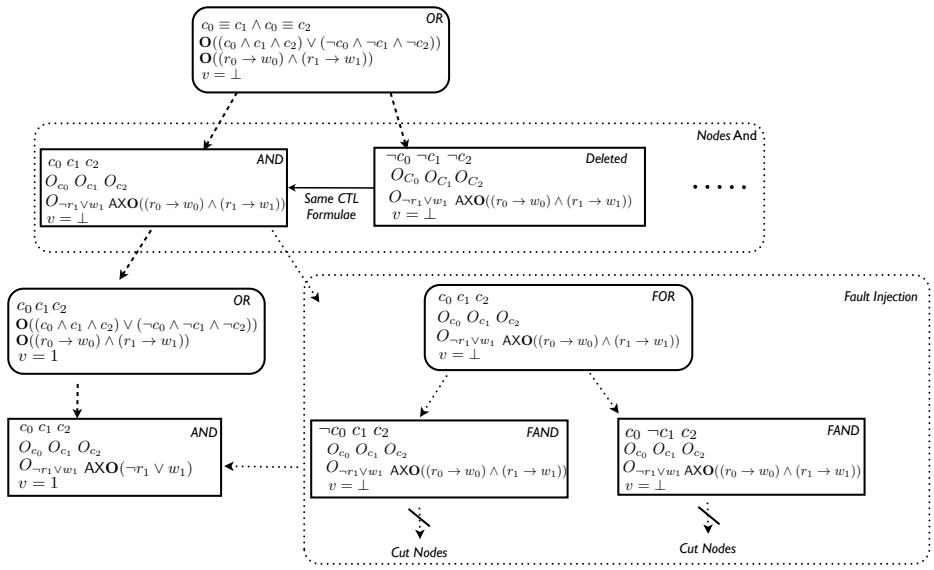

Fig. 1. Partial tableau for a Memory Cell.

case of the $FAnd$-node which contains $\neg c_0$ (say f_0), Alg. 3 checks whether this $FAnd$-node can be masked. Similarly for the other $FAnd$-nodes. We continue the process of negating deontic propositional variables from these faulty And-nodes. As a successor of f_0, we obtain the same information of f_0 with $\neg c_1$. Thus, we have that O_{c_0} and O_{c_1} are violated. Our algorithm cuts out these nodes because they cannot be masked. Similar results are obtained for the other combinations. Moreover, for each masked $FAnd$-node f, a (recovery) transition is added from it to each successor of $Masks(f)$ in case that we can reach a normal successor using the rules of the tableau. Notice that faults introduced change a bit and keep the bits unchanged during the recovery process. After that, since all the faulty nodes that can be masked were generated, we check condition $B.2$ of the simulation relation by using Alg. 4. This process may also cut out other faulty nodes: those which exhibit normal behavior which is not the behavior of the correct part of the system. Finally, we are ready to extract the fault-tolerant program from the tableau using the unfolding process (see Section 3). Fig. 2 shows the transition diagram of the program extracted from the structure in Fig. 1. For the sake of simplicity, the program does not include all the masked faults (these are similar to those shown in the program).

This program was generated considering that faults are computed from deontic operators automatically, only considering some basic operations on the data structures of the states (in this case bits). Other approaches [1, 12, 2] require faults to be given as input of the synthesis, e.g., as special actions specified as guarded commands. Our synthesis method can be straightforwardly adapted to

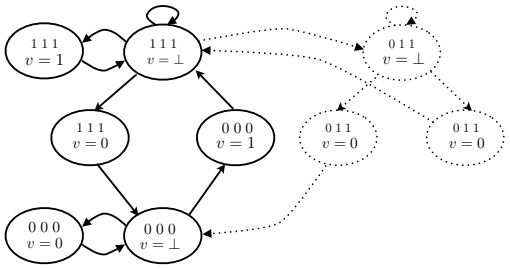

Fig. 2. Part of the fault-tolerant program extracted from the structure in figure 1.

consider fault specifications given by the user, capturing these as CTL formulas. For example, we can add the following formula in the memory cell example:

(11) $\mathsf{AG}(c_i \wedge v = \bot \rightarrow \mathsf{AX}(v = \bot \wedge \neg c_i))$, for $i = \{0, 1, 2\}$, at some point a bit may lose its charge.

Notice that sentence (11) is covered in our synthesis process.

5 Related Work

Various approaches have been proposed for synthesis of reactive systems from temporal logic specifications. The initial work was presented by Emerson and Clarke [7]. Their synthesis method was based on a decision procedure for checking the satisfiability of a CTL temporal logic specification. With respect to automated synthesis of fault-tolerant systems, Attie, Arora, and Emerson [2] presented an algorithm for synthesizing fault-tolerant programs from CTL specifications, based on a tableau method defined by Emerson and Clarke in [7]. One main difference with our work is that we use deontic operators to distinguish between good and bad system's behavior, while in [2] the abnormal behavior is captured by means of faulty actions. Another difference with our work is that in [2] safety properties only need to hold after faults or through fail-free paths, which implies that the semantics of CTL has to be adapted to cope with this condition. Another important stream of work is presented in [5]. Therein, Unity style programs are developed, the Unity logic is used to specify programs and to state fault-tolerant properties. Moreover, only a finite number of faults are allowed. It is important to notice that a main difference between that work and our approach is that our synthesized programs preserve *all* safety and liveness properties of the non-faulty part of the obtained program, while both [5] and [2] preserve only the properties explicitly stated in the specification.

6 Conclusions

We have presented an approach to synthesizing fault-tolerant components from dCTL- specifications. dCTL- is a branching time temporal logic equipped with

deontic operators, which is especially designed for fault-tolerant component specification. We believe this logic is better suited for fault-tolerance specification, and therefore synthesizing fault-tolerant implementations from dCTL- specifications is relevant. In order to capture fault-tolerance, we use an approach based on defining appropriate (bi)simulation relations, describing the relationship that must hold between a system specification and its fault-tolerant implementation. Our mechanism for synthesis is then based on combining decision procedures for the satisfiability of dCTL- temporal formulas, with (bi)simulation algorithms for checking a user required level of fault-tolerance. Here, we have dealt with masking tolerance, but our approach can be extended to other kinds of fault-tolerance, if these are captured via simulation relations.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by a Fellowship from IBM Canada, in support of the Automotive Partnership Canada funded project NECSIS; by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grants PICT PAE 2007 No. 2772, PICT 2010 No. 1690 and PICT 2010 No. 2611; and by the MEALS project (EU FP7 programme, grant agreement No. 295261).

References

1. A. Arora and S. Kulkarni, *Automating the Addition of Fault-Tolerance*, in Proc. of FTRTFT, 2000.
2. P.C. Attie, A. Arora, and E. A. Emerson, *Synthesis of fault-tolerant concurrent programs*, ACM Trans. Program. Lang. Syst. 26(1), 2004.
3. C. Baier and J.-P. Katoen, *Principles of Model Checking*, The MIT Press, 2008.
4. C. Bernardeschi, A. Fantechi and S. Gnesi, *Model checking fault tolerant systems*, Softw. Test., Verif. Reliab. 12(4), 2002.
5. B. Bonakdarpour, S. Kulkarni and F. Abujarad, *Symbolic synthesis of masking fault-tolerant distributed programs*, Distributed Computing 25(1), 2012.
6. P.F. Castro, C. Kilmurray, A. Acosta, and N. Aguirre, *dCTL: A Branching Time Temporal Logic for Fault-Tolerant System Verification*, in Proc. of SEFM, 2011.
7. E. M. Clarke and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching-time temporal logic*, in Proc. of Logic of Programs, 1981.
8. R. Demasi, P.F. Castro, T.S.E. Maibaum and N. Aguirre, *Characterizing Fault-Tolerant Systems by Means of Simulation Relations*, in Proc. of IFM, 2013.
9. A. Ebnenasir, S. Kulkarni and A. Arora *FTSyn: a framework for automatic synthesis of fault-tolerance*, STTT 10(5), 2008.
10. F. Gärtner, *Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments*, ACM Comput. Surv. 31(1), 1999.
11. M.R. Henzinger, T.A. Henzinger, and P.W. Kopke, *Computing Simulations on Finite and Infinite Graphs*, in Proc. of FOCS, 1995.
12. S. Kulkarni and A. Ebnenasir, *Automated Synthesis of Multitolerance*, in Proc. of DSN, 2004.
13. L. Lamport and S. Merz, *Specifying and Verifying Fault-Tolerant Systems*, in Proc. of FTRTFT, 1994.