syntMaskFT: A Tool for Synthesizing Masking Fault-Tolerant Programs from Deontic Specifications

Ramiro Demasi[1*], Pablo F. Castro[3,4†], Nicolás Ricci[3,4†], Thomas S.E. Maibaum[2], and Nazareno Aguirre[3,4†]

[1] Fondazione Bruno Kessler, Trento, Italy, `demasi@fbk.eu`
[2] Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, `tom@maibaum.org`
[3] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina, `{pcastro,nricci,naguirre}@dc.exa.unrc.edu.ar`
[4] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract. In this paper we introduce syntMaskFT, a tool that synthesizes fault-tolerant programs from specifications written in a fragment of branching time logic with deontic operators, designed for specifying fault-tolerant systems. The tool focuses on producing masking tolerant programs, that is, programs that during a failure mask faults in such a way that they cannot be observed by the environment. It is based on an algorithm we have introduced in previous work, and shown to be sound and complete. syntMaskFT takes a specification and automatically determines whether a masking fault-tolerant component is realizable; in such a case, a description of the component is produced together with the maximal set of faults that can be supported for this level of tolerance. We present the ideas behind the tool by means of a simple example, and also report the result of experiments realized with more complex case studies.

Keywords: Fault-tolerance, Program synthesis, Temporal logics, Deontic logics

1 Introduction

Critical systems, i.e., systems that are involved in serious or vital activities such as medical procedures (e.g., software for medical devices) or the control of vehicles (e.g., software controllers in the automotive and the avionics industries) are

[*] My contribution to this paper was made during my PhD studies at McMaster University, supported by a Fellowship from the IBM Canada Centre for Advanced Studies, in support of the Automotive Partnership Canada funded project NECSIS.
[†] This work was partially supported 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).

subject to a variety of potential failures. In many cases, these failures are not the result of software defects; instead, these may be the result of environmental conditions, such as power outages, electronic noise, or the physical failure of devices, that are not straightforward to avoid. The seriousness of the activities in which critical systems are involved makes it necessary to mitigate the effect of such failures. Therefore, the problem of guaranteeing through verification a certain degree of *fault-tolerance*, ensuring that systems will not be corrupted or degraded below a certain level despite the occurrence of faults, has gained considerable attention in recent years. Moreover, given the complexity of these systems and their properties, *automated* verification techniques for fault-tolerant systems are becoming increasingly important. While verification is (usually) *a posteriori*, a related automated alternative is *synthesis*. Various automated system analysis techniques (e.g., SAT and automata based techniques) have been recently adapted for *system synthesis*, i.e., the task of automatically obtaining a correct-by-construction implementation from a system specification [1, 2].

Despite the growing research on system synthesis, the availability of tools for fault-tolerant system synthesis is still low. In this paper we present syntMaskFT, a tool for synthesizing masking fault-tolerant programs from deontic logic specifications. The theoretical foundations of the tool were put forward in [4, 5]. In this paper, we concentrate on *masking fault-tolerance* which intuitively corresponds to the case in which the system is able to completely mask faults, not allowing these to have any observable consequences for the users. Roughly speaking, our synthesis algorithm takes as input a component specification, and automatically determines whether a component with masking fault-tolerance is realizable or not. In case such a fault-tolerant component is feasible, its implementation, together with the maximal set of faults supported for this level of tolerance, are automatically computed. A distinguishing feature of the tool is the use of Deontic Logic. These logics enrich standard (temporal) modalities with operators such as *obligation* and *permission*, making it possible to distinguish between normal and abnormal system behavior. In our approach, the logical specification of the component is given in dCTL-, a fragment of a branching time temporal logic with deontic operators [3], especially designed for fault-tolerant component specification. Let us emphasize that in our approach faults are declaratively embedded in the logical specification, where these are understood as violations to the obligations prescribing the behavior of the system. Thereby, we can inject faults automatically from deontic formula violations. Regarding the engine of our tool, it is based on a tableau-based method for deriving a finite state model from a dCTL- specification, with simulation algorithms for calculating masking fault-tolerance. Finally, we have conducted a series of experiments to test the performance of syntMaskFT in practice.

2 dCTL

The logic dCTL is an extension of 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 path formula ψ. Most importantly, the deontic operators allow us to declaratively distinguish the normative (correct, without faults) part of the system from its non-normative (faulty) part; an example of its use is shown below. The tool deals with a fragment of dCTL (named $\mathsf{dCTL\text{-}}$), described in the following BNF style grammar:

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

The standard boolean operators and the CTL quantifiers A and E have the usual semantics. Deontic operators have the following meaning: $\mathbf{O}(\psi)$: *the path formula 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 the path formula ψ holds.* These operators allow one to capture the intended behavior of the system when no faults are present. We present a simple example to illustrate the use of this logic to specify systems. The semantics of the logic is given via colored Kripke structures. A *colored Kripke structure* is a 5-tuple $\langle S, I, R, L, \mathcal{N}\rangle$, where S is a finite set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, $L : S \rightarrow \wp(AP)$ 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 can be thought of as faulty transitions, or simply *faults* (see Fig.1).

Example 1. Consider a memory cell that stores a bit of information and supports reading and writing operations. A state in this system maintains the current value of the memory cell, writing allows one to change this value, and reading returns the stored value. A property that one might associate with this model is that the value read from the cell coincides with that of the last writing performed in the system. Moreover, a potential fault occurs when a cell unexpectedly loses its charge, and its stored value turns into another one. A typical technique to deal with this situation is *redundancy*: use three memory bits instead of one. Writing operations are performed simultaneously on the three bits. Reading, on the other hand, returns the value that is repeated at least twice in the memory bits; this is known as *voting*, and the value read is written back to the three bits.

We take the following approach to model this system: each state is described by variables r and w, which record the value stored in the system (taking *voting* into account) and the last writing operation performed, respectively. First, note that variable w is only used to enable the verification of properties of the model, thus this variable will not be present in any implementation of the memory. The state also maintains the values of the three bits that constitute the system, captured by boolean variables c_0, c_1 and c_2. Part of the specification together with the associated intuition, is shown below:

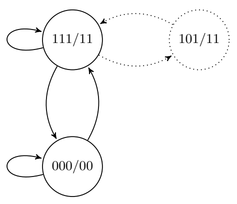
- $\mathbf{O}(r \leftrightarrow w)$, *the value read from the cell ought to coincide with the last writing performed.*
- $\mathbf{O}((c_0 \wedge c_1 \wedge c_2) \vee (\neg c_0 \wedge \neg c_1 \wedge \neg c_2))$, a safety property of the system: *the three bits should coincide*,

– $\mathsf{AG}(\neg r \leftrightarrow ((\neg c_0 \wedge \neg c_1) \vee (\neg c_0 \wedge \neg c_2) \vee (\neg c_1 \wedge \neg c_2)))$, *the reading of a* 0 *corresponds to the value read in the majority.*

We also note that we consider variables r, w as the *interface* of our memory cell, that is, the observable information of this specification. In particular, note the deontic formula given above; the first one states that we should read the same value that was written, and the second one says that, when no faults are present, the three bits of the cell coincide, otherwise a fault has occurred.

3 Masking Fault-Tolerance

Intuitively, a system is said to be *masking fault-tolerant* when the faulty behavior is masked in such a way that it cannot be observed by the user. In [6], Gärtner gives a more rigorous definition of masking fault-tolerance: a system is *masking tolerant* when it continues satisfying its specification even under the occurrence of faults. In [4], we propose to capture the notion of masking by means of simulation relations; here we introduce this idea by means of the memory example. Consider the colored Kripke structure in Figure 1 (where red states and arrows



$c_0 \wedge c_1 \wedge c_2 \wedge r \wedge w \rightarrow$
$\quad c_0, c_1, c_2, r, w := \neg c_0, \neg c_1, \neg c_2, \neg r, \neg w$
$c_0 \wedge c_1 \wedge c_2 \wedge r \wedge w \rightarrow \mathit{skip}$
$\neg c_0 \wedge \neg c_1 \wedge \neg c_2 \wedge \neg r \wedge \neg w \rightarrow$
$\quad c_0, c_1, c_2, r, w := \neg c_0, \neg c_1, \neg c_2, \neg r, \neg w$
$\neg c_0 \wedge \neg c_1 \wedge \neg c_2 \wedge \neg r \wedge \neg w \rightarrow \mathit{skip}$
$[c_0 \wedge c_1 \wedge c_2 \wedge r \wedge w \rightarrow c_1 := \neg c_1]$
$c_0 \wedge \neg c_1 \wedge c_2 \wedge r \wedge w \rightarrow c_1 := \neg c_1$

Fig. 1. Colored Structure and Guarded Program for Memory Cell

are depicted using dotted lines), this structure is a model of the system described in Example 1, where the circle labeled with 111/11 represents the state where all the bits are on, and r and w are set to *true*, and similarly for the other states. In this example a fault changing one bit is taken into account (the faulty state is drawn using dotted lines). Note that this model can also be described by using a simple guarded language; this is illustrated on the right in the same figure; note that in this case the faulty action is enclosed within brackets. We said that this structure is making fault-tolerant, since the faulty state is masked by the nonfaulty ones. Indeed, taking into account the variables in the interface (r, w in this case), one cannot observe any difference in comparison to the normal behavior of the program (the fault is masked by the redundancy of bits).

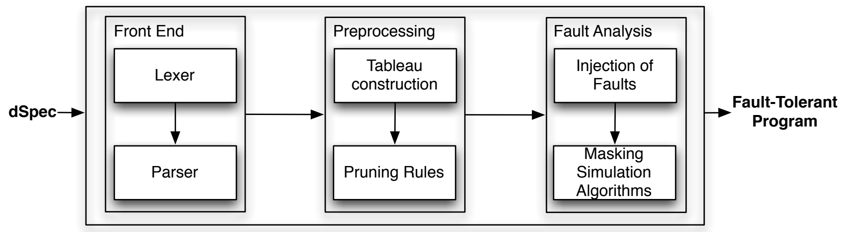


Fig. 2. The Architecture of syntMaskFT.

4 The Tool **syntMaskFT**

The main goal of syntMaskFT is, given a specification, to return the description of a system that masks a maximum number of faults. The description of the system can be given in two ways: a colored Kripke structure, or a simple description using a guarded command language in the style shown in the figure above. To this end, the tool uses a SAT method for dCTL- together with a simulation relation to prune the state space. The architecture of syntMaskFT is illustrated in Figure 2. The input of syntMaskFT is a deontic specification *dSpec*, composed of an $interface$, an *init-spec*, and a *normal-spec*. *interface* is described by a subset of the state variables, which, intuitively, form the visible part of the system; *init-spec* and *normal-spec* are dCTL- formulas, where the former specifies the initial states of the system, and the latter specifies properties that are required to hold in all states that are reachable from the initial states. Initially, syntMaskFT reads a deontic specification *dSpec* as an input file, which is then tokenized (Lexer) and parsed to obtain abstract syntax trees according to the dCTL- expression grammar (Parser). The abstract syntax trees are stored as elements of a set of dCTL- formulas. The preprocessing component constructs an initial tableau T_N for the input *dSpec* based on a dCTL- SAT procedure. Pruning rules are applied to the the tableau T_N in order to remove all nodes that are either propositionally inconsistent, do not have enough successors, or are labeled with a CTL or deontic eventuality formula which is not fulfilled. This process returns as a result *true*, if *dSpec* is satisfiable, or $false$, in the case *dSpec* is unsatisfiable. If *dSpec* is satisfiable, it has a finite model that is embedded in the tableau T_N. Assuming a positive result from the dCTL- decision procedure for *dSpec*, the next step is to perform a fault analysis. In this phase, faults are injected into the tableau in the first place, where faults are understood as (all possible) violations to the deontic obligations imposed in the description of the correct behavior of the system. Subsequently, a masking simulation algorithm (taking into account the input interface) is executed in order to remove those nodes from the tableau that cannot be masked. Finally, the tableau T_F is unravelled into a masking fault-tolerant program implementing *dSpec*.

Table 1. Experimental results.

Name	Faults Injected	faults unmasked/removed	Time in sec
Byzantine Agreement	7	4	0.20
Token Ring	220	150	111.85
N-Modular-Redundancy	410	260	535.91
Memory Cell	100	70	10.13

5 Implementation and Evaluation

The syntMaskFT tool is implemented in Java. All experiments have been conducted on a computer with a 2.9 Ghz Intel Core i5 with 4 GB of memory.

We have performed experiments to test the performance of our tool in practice. A well-known case study in the fault-tolerant community is the *Byzantine agreement problem*, formalized in [7]. We have specified this example in dCTL- and synthesized a solution for one general and three lieutenants. Another experiment that we have performed is *N-Modular-Redundancy* (NMR), a form of modular redundancy in which N systems perform a process whose results are processed by a majority-voting system to produce a single output. An NMR system can tolerate up to n module failures, where $n = (N-1)/2$. For this case study, we have evaluated 5-modular-redundancy using our tool. Our third experiment involves an adaptation of a case study from [2], a token ring for solving distributed mutual exclusion, where processes $0 \ldots N$ are organized in a ring with the token being circulated along the ring in a fixed direction. We have synthesized a token ring for four processes and an identical result to that reported in [2]. Finally, our last experiment is the memory cell presented in Example 1. Table 1 summarizes the experimental results on these models, reporting the number of faults injected and removed to achieve masking tolerance, and running times.

syntMaskFT is free software. Documentation and installation instructions can be found at `https://code.google.com/p/synt-mask-ft/`.

References

1. P.C. Attie, A. Arora, and E. A. Emerson, *Synthesis of fault-tolerant concurrent programs*, ACM Trans. Program. Lang. Syst. 26(1), 2004.
2. B. Bonakdarpour, S. Kulkarni and F. Abujarad, *Symbolic synthesis of masking fault-tolerant distributed programs*, Distributed Computing 25(1), 2012.
3. 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.
4. 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.
5. R. Demasi, P.F. Castro, T.S.E. Maibaum and N. Aguirre, *Synthesizing Fault-Tolerant Systems from Deontic Specifications*, in Proc. of ATVA, 2013.
6. F. Gärtner, *Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments*, ACM Comput. Surv. 31(1), 1999.
7. L. Lamport and S. Merz, *Specifying and Verifying Fault-Tolerant Systems*, in Proc. of FTRTFT, 1994.