MaskD: A Tool for Measuring Masking Fault-Tolerance*

Luciano Putruele[1,3](✉), Ramiro Demasi[2,3], Pablo F. Castro[1,3], and Pedro R. D'Argenio[2,3,4]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina, {lputruele,pcastro}@dc.exa.unrc.edu.ar
[2] Universidad Nacional de Córdoba, FAMAF, Córdoba, Argentina, {rdemasi,pedro.dargenio}@unc.edu.ar
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
[4] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Abstract. We present MaskD, an automated tool designed to measure the level of fault-tolerance provided by software components. The tool focuses on measuring masking fault-tolerance, that is, the kind of fault-tolerance that allows systems to mask faults in such a way that they cannot be observed by the users. The tool takes as input a nominal model (which serves as a specification) and its fault-tolerant implementation, described by means of a guarded-command language, and automatically computes the masking distance between them. This value can be understood as the level of fault-tolerance provided by the implementation. The tool is based on a sound and complete framework we have introduced in previous work. We present the ideas behind the tool by means of a simple example and report experiments realized on more complex case studies.

1 Introduction

Fault-tolerance is an important characteristic of critical software. It can be defined as the capability of systems to deal with unexpected events, which may be caused by code bugs, interaction with an uncooperative environment, hardware malfunctions, etc. Examples of fault-tolerant systems can be found everywhere: communication protocols, hardware circuits, avionic systems, cryptocurrencies, etc. So, the increasing relevance of critical software in everyday life has led to a renewed interest in the automatic verification of fault-tolerant properties. However, one of the main difficulties when reasoning about these kinds of properties is given by their quantitative nature, which is true even for non-probabilistic systems. A simple example is given by the introduction of redundancy in critical systems. This is, by far, one of the most used techniques in fault-tolerance. In practice, it

* This work was supported by ANPCyT PICT-2017-3894 (RAFTSys), ANPCyT PICT 2019-03134, SeCyT-UNC 33620180100354CB (ARES), and EU Grant agreement ID: 101008233 (MISSION).

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 396–403, 2022.
https://doi.org/10.1007/978-3-030-99524-9_22

is well known that adding more redundancy to a system increases its reliability. Measuring this increment is a central issue for evaluating fault-tolerant software. On the other hand, there is no de-facto way to formally characterize fault-tolerant properties. Thus these properties are usually encoded using ad-hoc mechanisms as part of a general design.

The usual flow for the design and verification of fault-tolerant systems consists of defining a nominal model (i.e., the "fault-free" or "ideal" program) and afterwards extending it with faulty behaviors that deviate from the normal behavior prescribed by the nominal model. This extended model represents the way in which the system operates under the occurrence of faults. More specifically, a model extension enriches a transition system by adding new (faulty) states and transitions from and to those states, namely fault-tolerant implementation.

On the other hand, during the last decade, significant progress has been made towards defining suitable metrics or distances for diverse types of quantitative models including real-time systems [11], probabilistic models [7], and metrics for linear and branching systems [5,2,10,13,19]. Some authors have already pointed out that these metrics can be useful to reason about the robustness of a system, a notion related to fault-tolerance.

We present MaskD, an automated tool designed to measure the level of fault-tolerance among software components, described by means of a guarded-command language. The tool focuses on measuring masking fault-tolerant components, that is, programs that mask faults in such a way that they cannot be observed by the environment. It is often classified as the most beneficial kind of fault-tolerance and it is a highly desirable property for critical systems. The tool takes as input a nominal model and its fault-tolerant implementation and automatically computes the masking distance between them. It is based on a framework we have introduced in [4], and shown to be sound and complete. In Section 2 we give a brief introduction to this framework.

The tool is well suited to support engineers for the analysis and design of fault-tolerant systems. More precisely, it uses a computable masking distance function such that an engineer can measure the masking tolerance of a given fault-tolerant implementation, i.e., the number of faults that the implementation is able to mask in the worst case. Thereby, the engineers can measure and compare the masking fault-tolerance distance of alternative fault-tolerant implementations, and select one that best fits their preferences.

2 The MaskD Tool

MaskD takes as input a nominal model and its fault-tolerant implementation, and produces as output the masking distance between them, which is a value in the interval $[0, 1]$. The input models are described using the guarded command language introduced in [3], a simple programming language common for describing fault-tolerant algorithms. More precisely, a program is a collection of processes, where each process is composed of a collection of labelled actions of the style: $[Label]\ Guard \rightarrow Command$, where $Guard$ is a Boolean condition over the actual

```
Process MEMORY {
 w : BOOL; // the last value written
 r : BOOL; // the value read from the
           // memory
 c0 : BOOL;
 Initial: w && c0 && r;
 [write1] true -> w=true, c0=true,
          r=true;
 [write0] true -> w=false, c0=false,
          r=false;
 [read0] !r -> r=r;
 [read1] r -> r=r;
}
```

```
Process MEMORY_FT {
 w : BOOL;
 r : BOOL;
 c0 : BOOL; // first bit
 c1 : BOOL; // second bit
 c2 : BOOL; // third bit
 Initial: w && c0 && c1 && c2 && r;
 [write1] true -> w=true, c0=true, c1=true,
          c3=true, r=true;
 [write0] true -> w=false, c0=false, c1=false,
          c3=false, r=false;
 [read0] !r -> r=r;
 [read1] r -> r=r;
 [fail1] faulty true -> c0=!c0, r =(!c0&&c1)||(c1&&c2)||
         (!c0&&c2);
 [fail2] faulty true -> c1=!c1, r =(c0&&!c1)||(!c1&&c2)||
         (c0&&c2);
 [fail3] faulty true -> c2=!c2, r =(c0&&c1)||(c1&&!c2)||
         (c0&&!c2);
}
```

Fig. 1. Processes for a memory cell example. On the left is the Nominal Model and on the right is the Fault-tolerant Model.

state of the program, *Command* is a collection of basic assignments, and *Label* is a name for the action. These syntactic constructions are called actions. The language also allows users to label an action as *internal* (i.e., silent actions). This is important for abstracting away internal parts of the system and building large models. Moreover, some actions can be labeled as *faulty* to indicate that they represent faults.

In order to compute the masking distance between two systems the tool uses notions coming from game theory. More precisely, a two-player game (played by the *Refuter* (R) and the *Verifier* (V)) is constructed using the two models. The intuition of this game is as follows. The Refuter chooses transitions of either the specification or the implementation to play, and the Verifier tries to match her choice. However, when the Refuter chooses a fault, the Verifier must match it with a masking transition. R wins if the game reaches the error state, denoted v_{err}. On the other hand, V wins when v_{err} is not reached during the game. Rewards are added to certain transitions in the game to reflect the fact that a fault was masked. Thus, given a play (a maximal path in the game graph) a function f_{mask} computes the value of the play: if it reaches the error state, the value is inversely proportional to the number of masking movements made by the Verifier; if the play is infinite, it receives a value of 0 indicating that the implementation was able to mask all the faults in the path. Summing up, the fault-tolerant implementation is masking fault-tolerant if the value of the game is 0. Furthermore, the bigger the number, the farther the masking distance between the fault-tolerant implementation and the specification.

As a running example, we consider a memory cell that stores a bit of information and supports reading and writing operations, presented in a state-based form in [6]. A state in this system maintains the current value of the memory cell ($m = i$, for $i = 0, 1$), writing allows one to change this value, and reading returns the stored value. In this system the result of a reading depends on the value stored in the cell. Thus, a property that one might associate with this model is

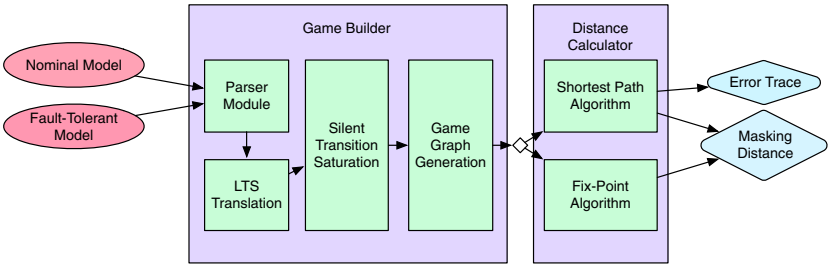

Fig. 2. Architecture of MaskD.

that the value read from the cell coincides with that of the last writing performed in the system.

A potential fault in this scenario occurs when a cell unexpectedly loses its charge, and its stored value turns into another one (e.g., it changes from 1 to 0 due to charge loss). A typical technique to deal with this situation is *redundancy*: in this case, three memory bits are used instead of only 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*. Figure 1 shows the processes representing the nominal and the fault-tolerant implementation of this example.

2.1 Architecture

MaskD is open source software written in Java. Documentation and installation instructions can be found at [1]. The architecture of the tool is shown in Fig. 2. We briefly describe below the key components of the tool:

Parser Module. It performs basic syntactic analysis over the input models, and produces data structures describing the inputs. Libraries Cup and JFlex were used to automatically generate the parser from the grammar describing the modeling language.

LTS Translation. The models obtained from the parser are translated into Labeled Transition Systems (LTSs), i.e., graphs whose vertices represent program states and whose transitions keep information about the actions in the models.

Silent Transition Saturation. The internal/silent transitions in the LTSs representing the input models are saturated using standard algorithms coming from process algebras [14]. As a result, saturated LTSs are generated, these are needed for verifying the masking relation when internal transitions are present.

Game Graph Generation. It uses the saturated LTSs to produce a game graph. Nodes in this graph encode the actual configuration of the game: the next player to play, the last played action, and references to the LTS states corresponding to the actual configuration of the game. Transitions in this graph correspond to the possible plays for the players, i.e., transitions in the original LTSs.

```
0. ERR_STATE
1. { <> , I_m1.read1 , <m1r,m1c0,m1c2> , V }
2. { <> , # , <m1r,m1c0,m1c2> , R }
3. { <> , I_m1.fail1 , <m1r,m1c0,m1c2> , V }
4. { <> , # , <m1c2> , R }
5. { <> , I_m1.fail3 , <m1c2> , V }
6. { <> , # , <> , R }
7. { <m1w,m1r,m1c0> , I_m1.write0 , <> , V }
8. { <m1w,m1r,m1c0> , # , <m1w,m1r,m1c0,m1c1,m1c2> , R }
```

Fig. 3. Error trace for the memory cell example.

Shortest Path Algorithm. If the input models are deterministic, Dial's shortest path algorithm is used to get the shortest path to the error state, from which the final value is calculated.

Fix-Point Algorithm. If the input models are non-deterministic, a bottom-up breadth-first search is used to compute the value of the game. This algorithm is based on well-known algorithms to solve reachability games that use *attractor* sets [15].

As explained above, an interesting point about our implementation is that, for deterministic systems, the masking distance between two systems can be computed by resorting to Dial's shortest path algorithm [17], which runs in linear time with respect to the size of the graphs used to represent the systems. In the case of non-deterministic systems, a fixpoint traversal approach based on breadth-first search is needed, making the algorithm less efficient. However, even in this case, the algorithm is polynomial.

2.2 Usage

The standard command to execute MaskD in a Unix operating system is:

```
./MaskD <options> <spec_path> <imp_path>
```

In this case the tool returns the masking distance between the specification and the implementation. Possible optional commands are: `-t: print error trace`, prints a trace to the error state; and `-s: start simulation`, starts a simulation from the initial state. A path to the error state is a useful feature for debugging program descriptions, which may be failing for unintended reasons. A trace for the memory cell example is shown in Fig. 3. States are denoted as `{spec_state, last_action_played, imp_state, player_turn}`. In this case, after two faults (bits being flipped), performing a read on the cell leads to the error state since on the nominal model the value is 0 while on the fault-tolerant model the value read by majority is 1. On the other hand, the simulation feature allows the user to manually select the available actions at each point of the masking game, which is also useful for verifying that the models behave as intended. By default, MaskD computes the masking distance for the given input using the algorithm for non-deterministic systems. The user can use option `-det` to switch to the deterministic masking distance algorithm.

Case Study	Redundancy	M. Distance	Time	Time(Det)
Redundant Memory Cell	3 bits	0.333	0.7s	0.6s
	5 bits	0.25	2.5s	1.9s
	7 bits	0.2	7.2s	5.7s
N-Modular Redundancy	3 modules	0.333	0.6s	0.5s
	5 modules	0.25	1.2s	0.7s
	7 modules	0.2	5.6s	3.8s
Dining Philosophers	2 philosophers	0.5	0.6s	0.6s
	3 philosophers	0.333	1.9s	0.9s
Byzantine Generals	3 generals	0.5	0.9s	–
	4 generals	0.333	17.1s	–
Raft LRCC (5)	1 follower	0	0.7s	0.8s
	2 followers	0	5.6s	3.6s
BRP (5)	1 retransm.	0.333	4.2s	–
	5 retransm.	0.143	4.8s	–
	10 retransm.	0.083	6.1s	–

Table 1. Some results of the masking distance for the case studies.

3 Experiments

We report on Table 1 some results of the masking distance for multiple instances of several case studies. These are: a Redundant Cell Memory (our running example), N-Modular Redundancy (a standard example of fault-tolerant system [18]), a variation of the Dining Philosophers problem [8], the Byzantine Generals problem introduced by Lamport et al. [12], the Log Replication consistency check of Raft [16], and the Bounded Retransmission Protocol (a well-known example of fault-tolerant protocol [9]) where we have modeled using silent actions and evaluating it with the weak masking distance. All case studies have been evaluated using the algorithms for both deterministic and non-deterministic games, with the exception of the non-deterministic models (i.e., the Byzantine Generals problem and the Bounded Retransmission Protocol). It is worth noting that most of the computational complexity arises from building the game graph rather than the actual masking distance calculation. For space reasons, we omit details of each case study and its complete experimental evaluation (delegated to the tool documentation).

Some words are useful to interpret the results of our running example. For the case of a 3 bit memory the masking distance is 0.333; the main reason for this is that the faulty model (in the worst case) is only able to mask 2 faults (in this example, a fault is an unexpected change of a bit) before failing to replicate the nominal behaviour (i.e., reading the majority value). Thus, the result comes from the definition of masking distance and taking into account the occurrence of two faults. The situation is similar for the other instances of this problem with more redundancy.

We have run our experiments on a MacBook Air with a 1.3 GHz Intel Core i5 processor and 4 GB of memory. The case studies for reproducing the results are available in the tool repository.

References

1. MaskD: Masking Distance Tool. https://doi.org/10.5281/zenodo.5815693
2. de Alfaro, L., Faella, M., Stoelinga, M.: Linear and branching system metrics. IEEE Trans. Software Eng. **35**(2), 258–273 (2009)
3. Arora, A., Gouda, M.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering **19**(11) (1993)
4. Castro, P.F., D'Argenio, P.R., Demasi, R., Putruele, L.: Measuring masking fault-tolerance. In: TACAS 2019, Prague, Czech Republic (2019)
5. Cerný, P., Henzinger, T.A., Radhakrishna, A.: Simulation distances. Theor. Comput. Sci. **413**(1), 21–35 (2012)
6. Demasi, R., Castro, P.F., Maibaum, T.S.E., Aguirre, N.: Simulation relations for fault-tolerance. Formal Asp. Comput. **29**(6), 1013–1050 (2017)
7. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labelled Markov processes. Theor. Comput. Sci. **318**(3), 323–354 (2004)
8. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Informatica **1**(2), 115–138 (1971)
9. Groote, J.F., van de Pol, J.: A bounded retransmission protocol for large data packets. In: Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96, Munich, Germany, July 1-5, 1996, Proceedings. pp. 536–550 (1996)
10. Henzinger, T.A.: Quantitative reactive modeling and verification. Computer Science - R&D **28**(4), 331–344 (2013)
11. Henzinger, T.A., Majumdar, R., Prabhu, V.S.: Quantifying similarities between timed systems. In: Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings. pp. 226–241 (2005)
12. Lamport, L., Shostak, R.E., Pease, M.C.: The Byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
13. Larsen, K.G., Fahrenberg, U., Thrane, C.R.: Metrics for weighted transition systems: Axiomatization and complexity. Theor. Comput. Sci. **412**(28), 3358–3369 (2011)
14. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
15. nski, M.J.: Algorithms for solving parity games. In: Apt, K.R., Grädel, E. (eds.) Lectures in Game Theory for Computer Scientist, chap. 3, pp. 74–95. Cambridge University Press, New York, NY, USA (2011)
16. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference. pp. 305–319. USENIX Association (2014)
17. R.B.Dial: Algorithm 360: shortest-path forest with topological ordering. Communications of ACM **12** (1969)
18. Shooman, M.L.: Reliability of Computer Systems and Networks: Fault Tolerance,Analysis,and Design. John Wiley & Sons, Inc (2002)
19. Thrane, C.R., Fahrenberg, U., Larsen, K.G.: Quantitative analysis of weighted transition systems. J. Log. Algebr. Program. **79**(7), 689–703 (2010)