Tolerange: Quantifying Fault Masking in Stochastic Systems

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 Tolerange, an open source tool tailored for measuring the masking fault-tolerance provided by stochastic systems. Tolerange takes as input a nominal model of a system together with the fault-tolerant version of it, both written in a Prism-like notation, and it computes the expected number of faults that the system's fault-tolerant version is able to mask. Our tool supports the analysis of randomized algorithms including the description of faults as probabilistic actions. It combines techniques coming from game theory, linear programming, and probabilistic transition systems. In this paper we describe the tool as well as its use to measure the masking fault-tolerance of some well-known examples.

1 Introduction

Measuring the fault-tolerance provided by systems is crucial for accurately assessing their dependability. An important kind of fault-tolerant systems are those that can mask faults in such a way that users cannot notice their occurrence. This type of fault-tolerance is often referred as *masking fault-tolerance* [7], and it is typically achieved by using some form of redundancy, e.g., replicating memory units, disks, or processes. However, in practice, systems can only mask faults during a finite period of time before exhibiting a failure. Hence, in most cases, designers are interested in quantifying the number of faults that systems are able to mask before failing. This can be thought of as the level of masking tolerance exhibited by systems. Even though critical systems are ubiquitous in modern life, few automated tools are available to perform such measures, which in practice are usually done using ad-hoc methods.

In this paper, we present Tolerange, a tool aimed at measuring the (expected) amount of masking fault-tolerance provided by stochastic systems. This encompasses both, the probabilities of faults, and the possible use of randomized algorithms. To the best of our knowledge, there are presently no other tools available for assessing the masking fault-tolerance of probabilistic systems.

Our tool is based on the notion of *probabilistic bisimulation relations* [11] between *probabilistic transition systems* [13]. The latter are a generalization of Markov chains supporting non-deterministic actions. Tolerange takes as input a specification (a description of the system without faults) and a fault-tolerant implementation of it (a system's version incorporating both faults and fault-tolerance mechanisms) and computes the expected number of *milestones* (events highlighted as important by the users) that the implementation guarantees to preserve under the presence of faults. A simple example of milestone could be the number of packages successfully transmitted by a sender in a communication protocol. The current tool extends the tool MaskD [12], which exclusively focuses on non-stochastic systems and does not support probabilistic models.

```
Process NOMINAL {
 v : INT;
 s : INT;          // 0 = normal,
                   // 1 = refresh
 Initial: v==0 && s==0;
 [write0] !(s==1) -> v=0,s=0;
 [write1] !(s==1) -> v=1,s=0;
 [read0] !(s==1) && v==0 -> v=v;
 [read1] !(s==1) && v==1 -> v=v;
 [tick] <1> s==0 -> 0.05 : s=1
                 ++ 0.95 : v=v;
 [refresh] s==1 && v==0 -> s=0, v=0;
 [refresh] s==1 && v==1 -> s=0, v=1;
}
```

Fig. 1. A Nominal Model for the Memory Example

Tolerange is meant to be used for measuring the masking fault-tolerance provided by stochastic models of critical software. For instance, it might help the designers to select between different fault-tolerant implementations. We illustrate this via a simple example. Consider the case of a RAM memory that uses redundancy to cope with faults, e.g., bits changing its value because of external noise. We add to the memory an additional fault-tolerant mechanism: a refreshing tick which is performed with certain frequency. In this setting, an important question is what is more convenient: adding redundancy at the hardware level (which can be more expensive) or refreshing with a higher frequency. Our tool helps to find an optimal balance between these two mechanisms. In Section 4, we analyse this example with Tolerange, as well as other well-known examples of fault-tolerance.

2 Running Example.

In this section we introduce in detail the example mentioned above. Consider a memory cell storing one bit of information that periodically refreshes its value. Fig. 1 shows the processes modeling the nominal and a fault-tolerant implementation of this example. Actions `read`i and `write`i (for $i = 0, 1$) represent the actions of reading and writing value i, respectively. The bit stored in the memory is saved in variable `v`. Action `tick` marks that one time unit has passed and, with probability `0.05`, it enables the refresh action (`refresh`). Variable `s` indicates whether the system is in write/read mode or producing a refresh.

A potential fault in this scenario occurs when a cell unexpectedly changes its value. In practice, the occurrence of such an error has a certain probability. A typical technique to deal with this situation is using three memory bits instead of one. In such a case, writing operations are performed simultaneously on the three bits,

while reading returns the value obtained by majority. Fig. 2 shows an augmented version of the nominal model with triple redundancy and the faults mentioned above. Therein, variable `v` counts the votes for value 1. A `tick` enables a refreshing as the original model, but also enables the occurrence of a fault with probability `0.1`. Now, variable `s` may get the value 2, representing a state in which a fault may occur.

3 The Tool

Tolerange takes as input a nominal model and a fault-tolerant version of it and produces as output the expected number of milestones that the implementation is able to guarantee under a fault model, which is a value in $\mathbb{R}^+$. To ensure that this value is well-defined we assume that the probability of a system's failure is 1. We also assume that the environment plays in a strongly fair manner (if a fault is infinitely often enabled, then it will occur infinitely often). We call these kinds of systems *almost-surely failing under fairness* [3]. The tool can automatically check whether the input is of this kind. It uses standard algorithms of game theory, together with linear programs for reasoning about probabilistic choices, the interested reader is referred to [4] for an in-depth description.

```
Process FAULTY {
 v : INT;
 s : INT;        // 0 = normal, 1 = refresh
                 // 2 = faulty
 Initial: v==0 && s==0;
 [write0] !(s==1) -> v=0,s=0;
 [write1] !(s==1) -> v=3,s=0;
 [read0] !(s==1) && v<=1 -> v=v;
 [read1] !(s==1) && v>1 -> v=v;
 [tick] <1> s==0 -> 0.05 : s=1
                 ++ 0.1 : s=2
                 ++ 0.85 : v=v;
 [tick] <1> s==2 -> 0.05 : s=1
                 ++ 0.95 : v=v;
 [refresh] s==1 && v<=1 -> s=0, v=0;
 [refresh] s==1 && v>1 -> s=0, v=3;
 [f] faulty s==2 && v<3 -> s=0, v=v+1;
 [f] faulty s==2 && v>=3 -> s=0, v=2;
 [f] faulty s==2 && v>0 -> s=0, v=v-1;
 [f] faulty s==2 && v<=0 -> s=0, v=1;
}
```

Fig. 2. Augmented Model for the Memory Example

The input models are written in a Prism-like language [10]. More precisely, a program is a collection of processes, where each process is composed of a collection of actions of the style: `[Label]<reward>Guard->[P]Command++[Q]Command`, where: `Guard` is a Boolean condition over the actual state of the program; `Command` is a collection of basic assignments; `Label` is a name for the action; the positive integer `reward` is optional, it states that the execution of this action counts as a "milestone" of value `reward`; and `++` is the probabilistic choice. Here `P` and `Q` are the probabilities corresponding to each branch of the choice. There can be many branches, as long as the sum of the probabilities is 1. The language also allows users to label actions as `faulty` to indicate that they model possible faults.

In order to compute the expected number of milestones guaranteed by the implementation, the tool defines a *two-player stochastic game* [6] using the inputs. The basis of the game is similar to a probabilistic bisimulation game [14], and it is played by two players, named for convenience the Refuter (R) and the Verifier (V). In this game, the Verifier intends to prove that the fault-tolerant version of the system is able to mask faults, while the Refuter intends to disprove that.

Roughly speaking, in each round of the game the Refuter may select a probabilistic transition in any of the models, and then the Verifier has to select a

probabilistic transition in the opposite model to match the Refuter's play. The match between the two probabilistic actions consists of a *probabilistic coupling* [9], which shows how the probabilities in one action are redistributed to another action. Furthermore, the Verifier may play a fault of the augmented model, in such a case the V is obliged to mask the fault and, from the user's point of view, it stays in the same state of the nominal model. This is modeled via a Dirac distribution. For instance, given a state s, the Dirac distribution (denoted $\varDelta_s$) states that, with probability 1, the next state is s. In addition, some game states may have associated a value to them, given by a function r, this indicates the reward given to the Verifier for some Refuter's action that she was able to match, for instance, when a fault is masked. The objective of the Verifier is to maximize $\sum_{i=0}^{\infty} \mathsf{r}(\rho_i)$, where ρ_i is the ith state of a play, while the objective of the Refuter is to minimize this function. The value of these games are obtained by means of computing the optimal strategies of both players using value iteration and Bellman equations [5], here linear programming is used for coping with the possible (uncountable) set of couplings between distributions. More precisely, a configuration of the game is a tuple: (s, a, s', μ, μ', P) where: s and s' are the current states in the nominal and augmented model, respectively. a is the last action played by the Refuter. μ and μ' are the probabilistic distribution played by some player, corresponding to the last action played, or to the matching step, a precise definition of the game graph is given in [4].

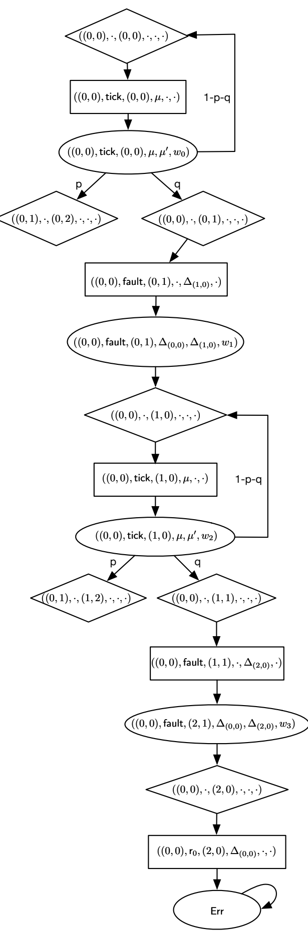

Fig. 3. A fragment of the Memory example game graph.

Consider the graph in Fig. 3. Therein, Verifier's nodes are depicted with boxes, Refuter's nodes with diamonds, and probabilistic nodes with circles. It represents a fragment of the masking game graph between `NOMINAL` and `FAULTY` of the running example. The vertices represent the variable values in the following order: $((\mathtt{v},\mathtt{s}), _, (\mathtt{v},\mathtt{s}), _, _, _, _)$. The distributions there are as follows:

$$\mu = \mathtt{p} \cdot (0,1) + (1{-}\mathtt{p}) \cdot (0,0)$$
$$\mu' = \mathtt{p} \cdot (0,2) + \mathtt{q} \cdot (0,1) + (1{-}\mathtt{p}{-}\mathtt{q}) \cdot (0,0)$$
$$\mu'' = \mathtt{p} \cdot (1,2) + \mathtt{q} \cdot (1,1) + (1{-}\mathtt{p}{-}\mathtt{q}) \cdot (1,0)$$
$$w_0 = \begin{cases} \mathtt{p} \cdot ((0,1),(0,2)) + \mathtt{q} \cdot ((0,0),(0,1) + \\ (1{-}\mathtt{p}{-}\mathtt{q}) \cdot ((0,0),(0,0)) \end{cases}$$
$$w_1 = \varDelta_{((0,0),(1,0))}$$

$$w_2 = \begin{cases} \mathtt{p} \cdot ((0,1),(1,2)) + \mathtt{q} \cdot ((0,0),(1,1)) + \\ (1{-}\mathtt{p}{-}\mathtt{q}) \cdot ((0,0),(1,0)) \end{cases}$$

$$w_3 = \Delta_{((0,0),(2,0))}$$

Notice that, in the majority of the vertices, many outgoing edges are omitted. In particular, the Verifier's vertex $((0,0),\mathtt{tick},(0,0),\mu,\cdot,\cdot,\mathrm{V})$ has infinitely many outgoing edges leading to probabilistic vertices of the form $((0,0),\mathtt{tick},(0,0),\mu,\mu',w,\mathsf{P})$, where w is a coupling for (μ,μ'). In the graph, we have chosen to distinguish coupling w_0 which is optimal for the Verifier (similarly later for w_2). We highlighted the path leading to error state v_{err}. Notice that this occurs as a consequence of the Refuter choosing to do a second `fault` in vertex $((0,0),\cdot,(1,1),\cdot,\cdot,\cdot,\mathrm{R})$ steering the game to the bottom part of the graph. Later, the Refuter chooses to read 0 in the `Nominal` model (at vertex $((0,0),\cdot,(2,0),\cdot,\cdot,\cdot,\mathrm{R})$) which the Verifier cannot match.

3.1 Architecture.

Tolerange is an open-source software written in Java and available at [2]. The architecture of the tool is shown in Fig. 4. The key components are:
Parser Module. It performs basic syntactic analysis over the input models, and produces data structures describing the inputs.
PTS Translation. The models obtained from the parser are translated into Probabilistic Transition Systems (PTSs), i.e., graphs whose vertices represent program states and probabilities associated with the transitions that keep information about the actions in the models.

Stochastic Masking Game Generation. A masking stochastic game is generated for the given PTSs where the weighting functions are symbolically captured by means of equation systems (linear programming).
Almost-Sure Failing Under Fairness Check. We provide an algorithm to check whether a game is almost-sure failing under fairness by computing predecessor sets in the symbolic game graph.
Value of the Game Calculation (Value Iteration). The value of the game is computed by solving a collection of functional equations via value iteration. We take such a value as the measure of fault-tolerance.

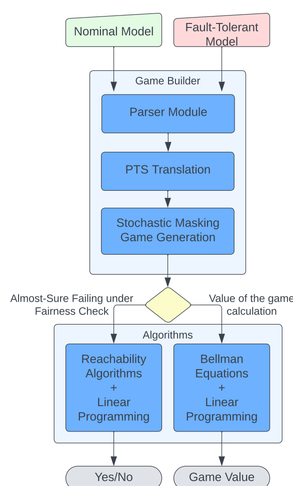

Fig. 4. Architecture of Tolerange.

3.2 Usage.

The standard command to execute Tolerange in a Unix operating system is: `./Tolerange <options> <spec_path> <imp_path>`. In this case the tool re-

turns the expected accumulated milestones achieved by the implementation. Optional commands are: `-f`, for checking if the game is almost-sure failing under fairness; `-gurobi`, which enables Gurobi [8] instead of SCC [1] for linear programming; and `b=N` to set an upper bound of N for the value of the game. By default, Tolerange computes the value of the game for the given input using SCC and an upper bound of the largest Java double.

4 Experiments

We have performed experiments on several examples. Due to space restrictions, here we only report the experiments performed over the memory example. Further case studies can be found in tool's repository. All the experiments were run on a MacBook Air with processor 1.3 GHz Intel Core i5 and a memory of 4 Gb. Table 1 reports the results obtained for our running example. The top subtable refers to counting the tick action as milestone, the middle subtable takes the refresh as milestone, and the bottom subtable reports the size of the models in number of states and transitions. M_t and M_r are the measurement results for the tick and refresh actions, considered as milestones, respectively.

Fault Prob.	Refr. Prob.	3 bits redundancy		5 bits redundancy		7 bits redundancy	
		M_t	Time	M_t	Time	M_t	Time
0.5	0.5	6	30s	14	1m43s	30	4m48s
	0.1	4.44	13s	7.28	32s	10.74	1m4s
	0.05	4.2	11s	6.62	26s	9.28	43s
0.1	0.5	70	4m21s	430	29m24s	2590.1	162m27s
	0.1	30	1m22s	70	4m23s	150	13m58s
	0.05	25	1m3s	47.5	2m41s	81.25	6m43s
0.05	0.5	240.02	12m10s	2660.1	128m37s	29281	68m49s
	0.1	80	30m31s	260.01	13m29s	800.03	89m20s
	0.05	60	2m39s	140	9m29s	300	23m26s

Fault Prob.	Refr. Prob.	3 bits redundancy		5 bits redundancy		7 bits redundancy	
		M_r	Time	M_r	Time	M_r	Time
0.5	0.5	3	35s	7	2m13s	15	4m54s
	0.1	0.44	14s	0.72	35s	1.07	1m8s
	0.05	0.21	12s	0.33	27s	0.46	58s
0.1	0.5	35	4m19s	215.02	37m55s	1295.05	199m34s
	0.1	3	1m33s	7	5m58s	15	14m24s
	0.05	1.25	1m19s	2.38	3m15s	4.06	8m6s
0.05	0.5	120.01	15m11s	1330.06	163m3s	14640.5	713m25s
	0.1	8	3m40s	26.01	16m54s	80.03	63m2s
	0.05	3	2m58s	7	8m58s	15	$25m$

size (states/ transitions)		3 bits	5 bits	7 bits
	spec.	4/12	4/12	4/12
	impl.	12/56	18/84	24/112
	game	505/2304	757/3688	1009/5012

Table 1. Results on the redundant memory cell with time and size information.

From the results in the table we can conclude that either, increasing the redundancy or augmenting the frequency of refreshing, has positive effects on the measures. In practice, these values can be taken into account when designing a fault-tolerant component that provides an optimal balance between efficiency and hardware costs. For example, assuming a fault probability of 0.05, one might prefer 3 bits and more frequent refreshing, over 5 bits with a less often refreshing, despite the software overhead.

As a final remark, we observe that small probabilities have a great impact in running times (in many cases even more than the model sizes), this is mainly due to the linear programming procedures that are used to solve the stochastic games. We plan to improve the tool by explicitly using the solutions of the linear programs (vertices of a polytope) instead of embedding the linear programs into Bellman equations.

References

1. SCC tool, https://www.ssclab.org/
2. Tolerange tool, https://github.com/cl-unrc-lab/Tolerange
3. Castro, P.F., D'Argenio, P.R., Demasi, R., Putruele, L.: Playing against fair adversaries in stochastic games with total rewards. In: CAV 2022, Haifa, Israel (2022)
4. Castro, P.F., D'Argenio, P.R., Demasi, R., Putruele, L.: Quantifying masking fault-tolerance via fair stochastic games. In: Proceedings of the Combined 30th International Workshop on Expressiveness in Concurrency and 20th Workshop on Structural Operational Semantics (2022)
5. Chatterjee, K., Henzinger, T.A.: Value iteration. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking - History, Achievements, Perspectives. Lecture Notes in Computer Science, vol. 5000, pp. 107–138. Springer (2008)
6. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer New York, NY, 1st edn. (1997)
7. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Computing Surveys **31** (1999)
8. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2022), https://www.gurobi.com
9. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991. pp. 266–277. IEEE Computer Society (1991)
10. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011)
11. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Information and Computation **94**(1), 1–28 (1991)
12. Putruele, L., Demasi, R., Castro, P.F., D'Argenio, P.R.: MaskD: A tool for measuring masking fault-tolerance. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Munich, Germany. Lecture Notes in Computer Science, vol. 13243, pp. 396–403. Springer (2022)
13. Segala, R.: Modelling and Verification of Randomized Distributed Real Time Systems. Ph.D. thesis, MIT (1995)
14. Stirling, C.: The joys of bisimulation. In: Brim, L., Gruska, J., Zlatuska, J. (eds.) Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic, August 24-28, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1450, pp. 142–151. Springer (1998)