Synthesizing Robust Systems with RATSY[*]

Roderick Bloem Hans-Jürgen Gamauf Georg Hofferek Bettina Könighofer
Robert Könighofer
Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Austria

Specifications for reactive systems often consist of environment assumptions and system guarantees. An implementation should not only be correct, but also robust in the sense that it behaves reasonably even when assumptions are (temporarily) violated. We present an extension of the requirements analysis and synthesis tool RATSY, which is able to synthesize robust systems from GR(1) specifications. A finite number of safety assumption violations is guaranteed to induce only a finite number of safety guarantee violations. We show how the specification can be turned into a two-pair Streett game, and how a winning strategy corresponding to a correct and robust implementation can be computed. Finally, we provide some experimental results.

1 Introduction

Property synthesis allows to automatically create systems from formal specifications [6, 11, 2]. Synthesized systems are *correct-by-construction*. Recently, there has been a lot of progress in making property synthesis practicable [10, 4, 3]. One remaining problem is that synthesized systems often do not behave reasonably in unexpected situations, i.e., when environment assumptions are violated.

Many specifications consist of environment assumptions and system guarantees. Guarantees must be fulfilled only, if all assumptions are satisfied. If an assumption is violated, even only for one tick, the system is allowed to behave arbitrarily. Assumptions may be violated e.g. due to a buggy environment, or due to radiation-related bit-flips. The latter issue is becoming more serious, due to continuously decreasing feature sizes [12]. Clearly, if assumptions are violated, the system may not be able to fulfill all guarantees. However, it should try to recover, if the environment does. Unfortunately, synthesized systems sometimes stop performing any useful interaction once an assumption has been violated.

We present an extension of the requirements analysis and synthesis tool RATSY [2] which allows to synthesize robust systems from GR(1) specifications [10]. Our work is based on ideas from [5] and [1]. We define a system to be robust if finitely many environment failures induce only finitely many system failures [5]. A GR(1) specification can be turned into a one-pair Streett game such that a winning strategy corresponds to a correct implementation [1]. A second Streett pair can be added such that the strategy corresponds to a robust system. We show how to compute the strategy, based on the algorithm of [9].

Different notions of robustness have been studied in different settings. In [5], robustness for safety specifications is considered. Synthesis is done using one-pair Streett games. We use the same notion of robustness but consider GR(1) specifications. Robustness for liveness is addressed in [1]: for any number of violated assumptions, the number of violated guarantees must be as low as possible. We use their idea of transforming GR(1) into Streett games via a counting construction. In [8], robustness is not defined in terms of assumption and guarantee violations, but using metrics on the state of a system. Synthesis

[*]This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), and by the Austrian Science Fund (FWF) through the national research network RiSE (S11406-N23).

Submitted to:
SYNT 2012

is performed via special automata incorporating these metrics. Robustness of sequential circuits is also addressed in [7]. Inputs are divided into control and disturbance variables. A system is robust if a finite number of changes in disturbance inputs result in a bounded number of changes in the outputs. Synthesis is not addressed.

The rest of this paper is organized as follows. Section 2 presents an example to illustrate the problem. Section 3 explains our method to synthesize robust systems. Section 4 explains the computation of a winning strategy for two-pair Streett games in more detail. In section 5, our method is applied to an example. Section 6 presents experimental results and concludes.

2 Illustration of the Problem

Consider the specification of a simple arbiter for a resource shared between two clients. The input signals r_1 and r_2 are used by the clients to request access to the resource. The arbiter grants access via the output signals g_1 and g_2. The system must fulfill the following requirements. First, the system is never allowed to raise both grant signals at the same time. In LTL syntax, this can be written as $G_1 = \mathsf{G}\neg(g_1 \wedge g_2)$. Second, a request has to be followed immediately by a grant, which can be formalized by the guarantees $G_2 = \mathsf{G}(r_1 \rightarrow \mathsf{X} g_1)$ and $G_3 = \mathsf{G}(r_2 \rightarrow \mathsf{X} g_2)$. Finally, it is assumed that the environment never raises both request signals at the same time: $A = \mathsf{G}\neg(r_1 \wedge r_2)$. Combining the three guarantees and the assumption results in the specification $\varphi = A \rightarrow G_1 \wedge G_2 \wedge G_3$. It requires the arbiter to satisfy all three guarantees, if the assumption is fulfilled.

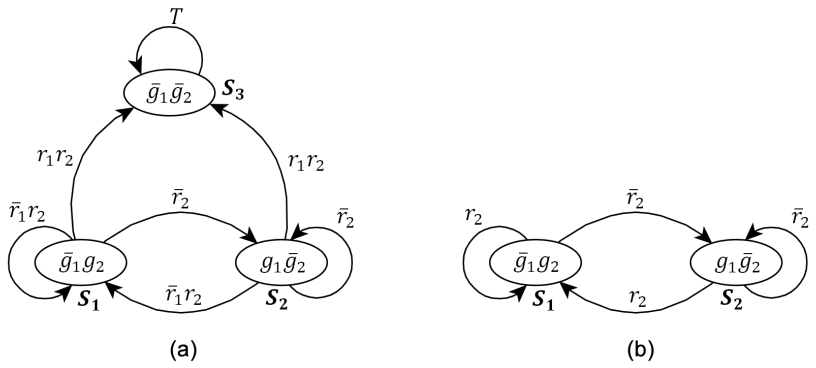

Figure 1: Synthesized Finite State Machines.

One possible implementation of φ (in form of a finite state machine) is shown in Figure 1(a). If the environment assumption is violated, i.e., r_1 and r_2 are raised at the same time, the machine enters state $S3$, and will remain there forever. Irrespective of future inputs, both grant signals stay low, therefore G_2 and G_3 will not be fulfilled anymore. This is not robust: a finite number of environment errors leads to an infinite number of system errors, i.e., the system does not recover. Our new synthesis algorithm guarantees that this cannot happen. Instead, our approach may lead to an implementation as shown in Figure 1(b), which does not exhibit the aforementioned weakness. If two requests occur simultaneously now, one will be discarded while the other one will be granted. Once the environment resumes correct behavior, the system will also fulfill all its guarantees again.

3 Robust Synthesis from GR(1) Specifications

A GR(1) specification consists of environment assumptions and system guarantees. There are two kinds of assumptions and guarantees. **Safety properties** encode conditions which have to hold in all time steps. They are of the form $\mathsf{G}\, p_i$ for some Boolean condition p_i. **Fairness properties** are conditions which have to hold infinitely often. They can be written as $\mathsf{G}\,\mathsf{F}\, p_j$. We define an implementation to be robust if a finite number of safety assumption violations induces only a finite number of safety guarantee violations.

Our robust synthesis method consists of two steps. First, the specification is transformed into a one-pair Streett game via a counting construction [1]. The safety properties are encoded directly into the transition relation of the Streett game. The fairness properties are expressed via the Streett pair. For m fairness assumptions $\mathsf{G}\,\mathsf{F} A_i$ (with $1 \leq i \leq m$) and n fairness guarantees $\mathsf{G}\,\mathsf{F} G_j$ (with $1 \leq j \leq n$), the state-space is extended with two counters $x \in \{0,\dots m\}$ and $y \in \{0,\dots n\}$, which can be encoded with $\lceil \log_2(m+1) \rceil + \lceil \log_2(n+1) \rceil$ additional bits. The counter x is incremented modulo $m+1$ whenever assumption A_x (corresponding to the current counter value) is satisfied; similarly for y, G_y, modulo $n+1$. If a counter has the special value 0, it is always incremented. The counter value $x = 0$ indicates that all A_i have been satisfied in a row; $y = 0$ indicates that same for all G_j. Hence, the condition $(\mathsf{G}\,\mathsf{F} x = 0) \rightarrow (\mathsf{G}\,\mathsf{F} y = 0)$, expressed by the Streett pair $\langle (x = 0), (y = 0) \rangle$, ensures that the liveness part of the specification is encoded properly in the game. A winning strategy for this game corresponds to a correct implementation.

In order to obtain a system which is also robust, we extend the state-space of the Streett game by two additional Boolean variables ok_e and ok_s. Variable ok_e is set to false whenever the environment violates some safety assumption, ok_s is set to false iff the system violates a safety guarantee. Initially, both ok_e and ok_s are set to true. Our notion of robustness can now be formulated using the condition $(\mathsf{G}\,\mathsf{F} \neg ok_s) \rightarrow (\mathsf{G}\,\mathsf{F} \neg ok_e)$, which is expressed by the Streett pair $\langle (\neg ok_s), (\neg ok_e) \rangle$. An infinite number of system errors is only allowed if there is an infinite number of environment errors.

A winning strategy for the two-pair Streett game corresponds to a correct and robust implementation. We use a recursive fixpoint algorithm to compute the winning region [9]. Intermediate results of this computation can be used to obtain the winning strategy.

4 Computing a Winning Strategy for Streett(2)

Figure 2 shows the algorithm to compute the winning region of a Streett game [9]. The input `Set` is a set of Streett pairs $\langle a, b \rangle$. The function `pr(X)` returns the set of states from which the system can force the play into X in one step. `LFix` and `GFix` represent least and greatest fixpoint computations over sets of states. The operators `&`, `|` and `!` perform intersection, union, and complementation of sets.

The following discussion assumes `Set=`$\{\langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle\}$. Let Y_1 be the fixpoint in Y for the first Streett pair in the top-level call to `Str`. Y_2 is the result for the second pair. We denote the iterates of these fixpoint computations by $Y_{1,0} \dots Y_{1,C_1}$ and $Y_{2,0} \dots Y_{2,C_2}$. For both Streett pairs, the function `Str` is called recursively. The iterates of Y in the recursive call during the computation of $Y_{i,j}$ are denoted $Y_{i,j,0} \dots Y_{i,j,C_{i,j}}$ for $i \in \{1,2\}$ and $j \in \{0,\dots C_i\}$.

Figure 3 illustrates the intuitive meaning of the iterates. As long as a_1 and a_2 hold, it is possible to proceed to the next lower iterate of Y_i. Y_2 is reachable from $Y_{1,1}$ and Y_1 is reachable from $Y_{2,1}$. The resulting cycle allows to visit b_1 and b_2 infinitely often. If a_2 is not satisfied, the next lower iterate of Y_2 may not be reachable. Not reaching b_2 ever again is fine if a_2 is also never satisfied again. However,

```
Func main_Streett(Set)
 If (|Set|=0)
  Return mStr(true,false);
 Return Str(Set,true,false);
End -- Func main_Streett(Set)
```

```
Func mStr(sng,rt)
 GFix(X)
  X = rt | sng & pr(X);
 End -- GFix(X)
 Return X;
End -- mStr
```

```
Func Str(Set,sng,rt)
 GFix(Z)
  Foreach (<a,b> in Set)
   nSet = Set - <a,b>;
   p1 = rt | sng & b & pr(Z);
   LFix(Y)
    p2 = p1 | sng & pr(Y);
    If (|nSet|=0)
     Y = mStr(sng & !a,p2);
    Else
     Y = Str(nSet,sng&!a,p2);
   End -- LFix(Y)
   Z = Y;
  End -- Foreach (<a,b>)
 End -- GFix(Z)
 Return Z;
End -- Str
```

Figure 2: Algorithm to compute the winning strategy.

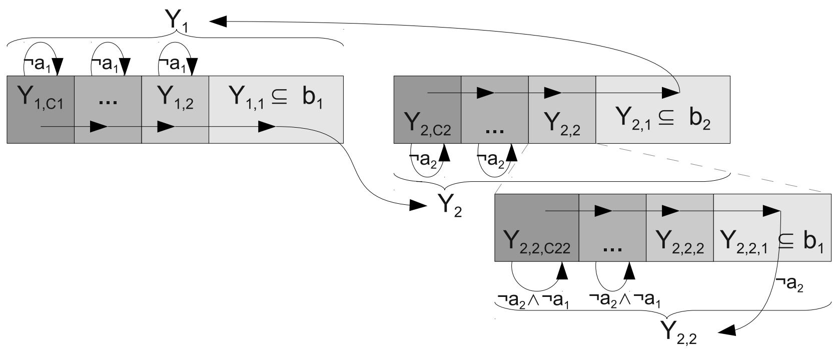

Figure 3: Illustration of the iterates of the fixpoint computation.

the other Streett pair still has to be handled. This is ensured through the iterates from the recursive step. Figure 3 shows them for $Y_{2,2}$ only. If a_1 holds, it is possible to proceed to the next lower iterate of $Y_{2,2}$ and from $Y_{2,2,1}$ back to $Y_{2,2}$. This cycle ensures that b_1 is visited infinitely often if a_1 holds infinitely often but a_2 does not. Analogously for all other iterates $Y_{i,j}$.

To define a strategy, we introduce one bit m of memory. $m = 0$ means b_1 should be fulfilled next, $m = 1$ means b_2 should be fulfilled next. The strategy is composed of several parts, which we enumerate in the following table. They are prioritized from top to bottom. If a particular sub-strategy cannot be applied (because of violated assumptions), the next one is tried.

Nr.	present state in:	next state in:	informal description
1	$Y_{1,i} \setminus Y_{1,i-1}, \neg m$	$Y_{1,i-1}, \neg m$	step towards b_1
2	$Y_{2,i} \setminus Y_{2,i-1}, m$	$Y_{2,i-1}, m$	step towards b_2
3	$Y_{1,1}, \neg m$	Z, m	b_1 reached; switch towards b_2
4	$Y_{2,1}, m$	$Z, \neg m$	b_2 reached; switch towards b_1
5	$Y_{1,i,j} \setminus Y_{1,i,j-1}, \neg m$	$Y_{1,i,j-1}, \neg m$	$\neg a_1$; sub-game towards b_2
6	$Y_{2,i,j} \setminus Y_{2,i,j-1}, m$	$Y_{2,i,j-1}, m$	$\neg a_2$; sub-game towards b_1
7	$Y_{1,i,1}, \neg m$	$Y_{1,i}, \neg m$	b_2 reached in sub-game
8	$Y_{2,i,1}, m$	$Y_{2,i}, m$	b_1 reached in sub-game
9	$Y_{1,i,j} \setminus Y_{1,i,j-1}, \neg m$	$Y_{1,i,j}, \neg m$	$\neg a_1, \neg a_2$; stay
10	$Y_{2,i,j} \setminus Y_{2,i,j-1}, m$	$Y_{2,i,j}, m$	$\neg a_2, \neg a_1$; stay

5 Example of Robust Synthesis

To demonstrate our approach, this section gives an example. Consider the specification of a full-handshake protocol with a request input signal r and a grant output signal g. For the environment, the safety assumption $A_1 = \mathsf{G}((r \wedge \neg g \rightarrow \mathsf{X}\, r) \wedge (\neg r \wedge g \rightarrow \mathsf{X} \neg r))$ and the fairness assumption $A_2 = \mathsf{G}\,\mathsf{F}(\neg r \vee \neg g)$ are defined. The system has to satisfy the safety guarantee $G_1 = \mathsf{G}((\neg r \wedge \neg g \rightarrow \mathsf{X} \neg g) \wedge (r \wedge g \rightarrow \mathsf{X}\, g))$ and the fairness guarantee $G_2 = \mathsf{G}\,\mathsf{F}((r \wedge g) \vee (\neg r \wedge \neg g))$. Combining the assumptions and the guarantees results in the specification $\varphi = A_1 \wedge A_2 \rightarrow G_1 \wedge G_2$.

First, the specification is transformed into a **one-pair Streett game**. In this example there is no need for a **counting construction**, since there is only a single fairness assumption and guarantee. Figure 4(a) illustrates the encoding of the safety properties in the transition relation of the Streett game. The first bit of each state corresponds to the request signal r and the second bit to the grant signal g. For example, the transitions require that, if there is a request, r has to stay *high* until the request is granted.

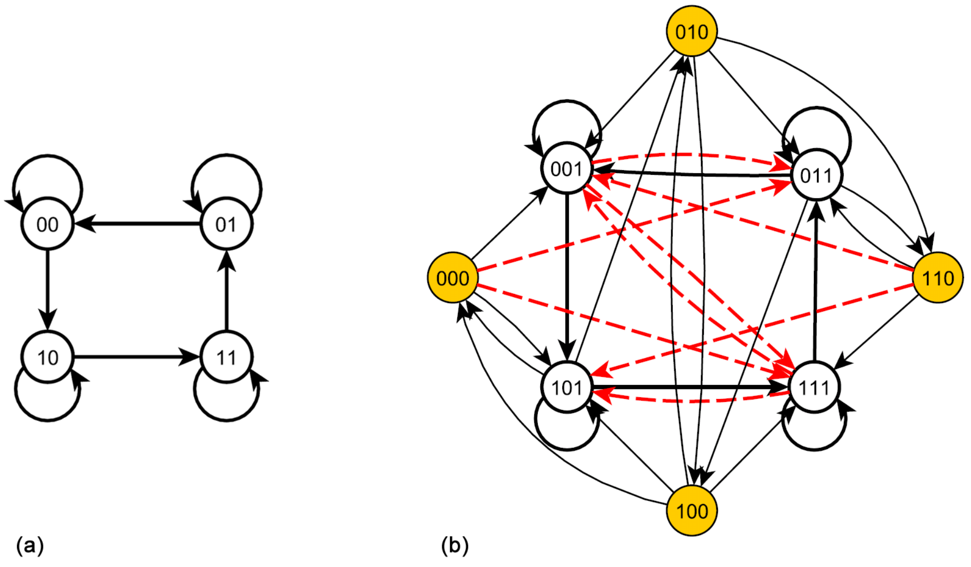

Figure 4: arbiter example (a) Encoding of the safety properties in the transition relation. (b) Extension of the state space.

The following step is to **extend the state space** with the variables ok_e and ok_s, as shown in Figure 4(b). The third bit of each state corresponds to the signal ok_e, which encodes an error caused by the environment. If this bit is *true*, no error occurred. Black solid lines indicate that there is no system error ($ok_s = 1$) and red dashed-lines indicate that there is one ($ok_s = 0$). Colored states represent states where an environment error has occurred. E.g., assume we start in state 101. In this state, a request occurred which has not been granted yet, and no environment error occurred. The safety assumption prohibits the environment from lowering the request. If it does anyway, depending on the choice of the system, either the state "010" or "000" is entered, which are both colored states.

Next, the **winning region and the strategy** are computed. Figure 5 illustrates the iterates of the fixpoint computation. We have $a_1 = \neg(r \wedge g), b_1 = (r \wedge g) \vee (\neg r \wedge \neg g), a_2 = \neg ok_s, b_2 = \neg ok_e$. To illustrate strategy computation, we consider the following scenario. Assume that $m = 1$ and the arbiter is in a state out of $Y_{2,2} \backslash Y_{2,1}$. The value of $m = 1$ dictates to visit a state out of $Y_{2,1}$ next, if possible. $Y_{2,1}$ contains all states with an environment error. If we assume that the environment always behaves correctly, the set $Y_{2,1}$ becomes unreachable. In order to win the game anyway, the system is not allowed to make a mistake either, so the arbiter stays in $Y_{2,2}$. This way the second Streett pair $\langle(\neg ok_s), (\neg ok_e)\rangle$ is fulfilled, because both sets are only visited finitely often. To win the game, the first Streett pair also has to be fulfilled. Therefore the subgame is entered, trying to reach states in b_1 while staying in $Y_{2,2}$. Through the loop in $Y_{2,2}$, it is possible to visit these states infinitely often, fulfilling the first Streett pair as well.

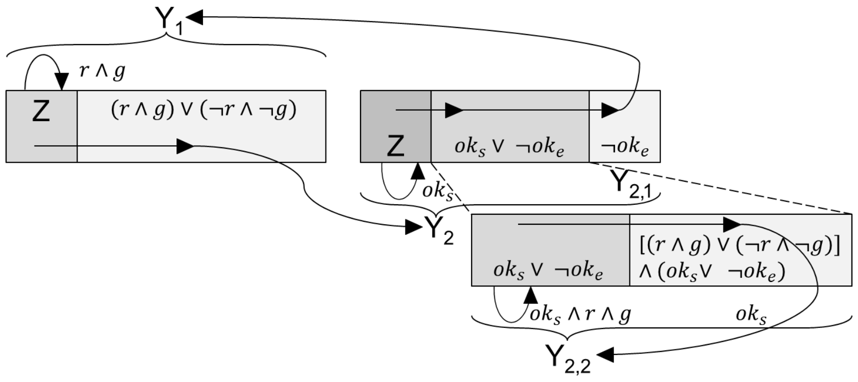

Figure 5: Illustration of the iterates of the fixpoint computation.

6 Results and Conclusions

We tested our implementation in RATSY with an arbiter, with N request and acknowledge lines (cf. Section 2). Table 6 compares the synthesis time (seconds) and the implementation size (lines of Verilog), with and without robustness. As expected, the robust approach takes more time and creates larger circuits than RATSY's original synthesis algorithm. This is due to the higher complexity of the new method. Simulating the synthesized systems shows that the number of system errors needed to recover after one environment error is really small. In most practical cases only one or even no system errors are needed.

The original synthesis algorithm of RATSY gave no formal guarantees for robustness. The extension presented in this paper guarantees that synthesized systems are *correct-and-robust-by-construction*. This comes at the cost of larger circuits and longer synthesis times, due to the increased computational

Table 1: Performance results

N	size w/o robustness	size with robustness	time w/o robustness	time with robustness
2	85	501	0.04	0.15
3	145	1,234	0.08	1.07
4	230	2,829	0.14	3.37
5	324	5,614	0.18	11.13
10	1,072	90,215	0.81	3,485
15	2,215	$6.2 \cdot 10^6$	3.30	26,172

complexity. Experimental results show that synthesized robust systems are able to recover with just very few system errors. In many practical cases, the ratio between system errors and environment errors is less than one. Since, in practice, one has to be prepared for environment errors, guaranteed robustness is an important property enhancing the quality of a system.

References

[1] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger & Barbara Jobstmann (2010): *Robustness in the Presence of Liveness*. In: *CAV*, pp. 410–424. Available at `http://dx.doi.org/10.1007/978-3-642-14295-6_36`.

[2] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan & Richard Seeber (2010): *RATSY - A New Requirements Analysis Tool with Synthesis*. In: *CAV*, pp. 425–429. Available at `http://dx.doi.org/10.1007/978-3-642-14295-6_37`.

[3] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Martin Weiglhofer (2007): *Interactive presentation: Automatic hardware synthesis from specifications: a case study*. In: *DATE*, pp. 1188–1193. Available at `http://doi.acm.org/10.1145/1266366.1266622`.

[4] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Martin Weiglhofer (2007): *Specify, Compile, Run: Hardware from PSL*. 190, pp. 3–16. Available at `http://dx.doi.org/10.1016/j.entcs.2007.09.004`.

[5] Roderick Bloem, Karin Greimel, Thomas A. Henzinger & Barbara Jobstmann (2009): *Synthesizing robust systems*. In: *FMCAD*, pp. 85–92. Available at `http://dx.doi.org/10.1109/FMCAD.2009.5351139`.

[6] A. Church (1962): *Logic, Arithmetic and Automata*. In: Proceedings International Mathematical Congress.

[7] Laurent Doyen, Thomas A. Henzinger, Axel Legay & Dejan Nickovic (2010): *Robustness of Sequential Circuits*. In: *ACSD*, pp. 77–84. Available at `http://doi.ieeecomputersociety.org/10.1109/ACSD.2010.26`.

[8] Rupak Majumdar, Elaine Render & Paulo Tabuada (2011): *Robust discrete synthesis against unspecified disturbances*. In: *HSCC*, pp. 211–220. Available at `http://doi.acm.org/10.1145/1967701.1967732`.

[9] Nir Piterman & Amir Pnueli (2006): *Faster Solutions of Rabin and Streett Games*. In: *LICS*, pp. 275–284. Available at `http://doi.ieeecomputersociety.org/10.1109/LICS.2006.23`.

[10] Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2006): *Synthesis of Reactive(1) Designs*. In: *VMCAI*, pp. 364–380. Available at `http://dx.doi.org/10.1007/11609773_24`.

[11] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, pp. 179–190. Available at `http://doi.acm.org/10.1145/75277.75293`.

[12] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger & Lorenzo Alvisi (2002): *Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic*. In: *DSN*, pp. 389–398. Available at `http://doi.ieeecomputersociety.org/10.1109/DSN.2002.1028924`.