# Specification-Centered Robustness

*(Invited Paper)*

Roderick Bloem*, Krishnendu Chatterjee†, Karin Greimel‡, Thomas A. Henzinger† and Barbara Jobstmann§
*IAIK, University of Technology Graz, Austria
†Institute of Science and Technology Austria (IST Austria)
‡NXP Semiconductors, Austria
§Verimag, CNRS and University of Grenoble, France

***Abstract*—In addition to being correct, a system should be robust, that is, it should behave reasonably even after receiving unexpected inputs. In this paper, we summarize two formal notions of robustness that we have introduced previously for reactive systems. One of the notions is based on assigning costs for failures on a user-provided notion of incorrect transitions in a specification. Here, we define a system to be robust if a finite number of incorrect inputs does not lead to an infinite number of incorrect outputs. We also give a more refined notion of robustness that aims to minimize the ratio of output failures to input failures. The second notion is aimed at liveness. In contrast to the previous notion, it has no concept of recovery from an error. Instead, it compares the ratio of the number of liveness constraints that the system violates to the number of liveness constraints that the environment violates.**

## I. Introduction

Classical software engineering distinguishes (cf. [21]) between *correctness* and *robustness*, the latter being the degree to which a system continues to function properly when confronted with invalid inputs, defects in connected software or hardware components, or unexpected operating conditions. The distinction recognizes that it is unrealistic to specify the proper reaction to each environment fault — this would be time consuming, error prone, and would clutter the specification. Note that robustness never precludes the specification of a proper reaction to an anticipated failure in the input, but only requires that behavior be "reasonable" in cases that are not completely specified.

Robustness is a very important property. For instance, hardware failure in one node of a large cloud should not bring down the entire network; a network daemon specified for up to 1000 requests per seconds could drop request 1001, but should not shut down when it comes; or the failure of one dining philosopher to put down a fork should not lead to starvation of all philosophers at the table.

The Wikipedia article for the related notion of fault tolerance states that a system exhibits graceful degradation under the following condition.

> If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naively-designed system in which even a small failure can cause total breakdown.

Note that this definition assumes a measurable and comparable severity of failure. Of course, the generality of the definition precludes a formalization of this notion.

In this paper, we describe two possible existing formalizations of the notion of robustness. We focus on failures in the input and output sequences of reactive systems, disregarding the possibility of a hardware failure *within* the system under consideration. We define a measure of degradation of quality and following the Wikipedia article in taking the proportion of the output degradation to the input degradation as the defining measure of robustness. In fact, we propose a simple way for the user to specify the degradation of quality in a property-based manner.

We assume that our specifications are of the form $A \to G$, where $A$ is a set of assumptions and $G$ is a set of guarantees. Both assumptions and guarantees are given as deterministic Büchi automata. This allows us to easily distinguish the (input) degradation due to the environment and the (output) degradation due to the system, and to assign a cost to each. We do this by adding extra, marked, edges to the automata, so that their language becomes universal. Every transition along a marked edge corresponds to a mistake and occurs a cost (cf. [7]). The quality of a given input/output trace is then the ratio of the system cost to the environment cost. This ratio is to be kept as low as possible. In particular, it should not be infinite, which happens if a system becomes fully irresponsive after a finite number of unexpected inputs.

This approach works fine for safety violations, but not for liveness violations. After all, a liveness violation cannot be detected within any finite amount of time. We therefore suggest a different approach for liveness violations by counting the number of individual requirements that are violated by the system and by the environment and again minimizing the ratio between these numbers.

Our definitions are easily extended to systems, most naturally by defining the robustness measure as the infimum of the measure across all possible traces. Other definitions are also possible, e.g., taking the average over all possible traces [34]. So, our notions naturally extend to verification and synthesis. Verification asks the question whether the robustness measure of a system is below a given threshold, and synthesis asks for the most robust system. We recapitulate the earlier solutions to these questions [9], [6] in this paper.

Note that robustness is particularly useful for synthesis. After all, synthesis tools differ from human designers in that they do not have a built in sense of a natural behavior of a system. Indeed, in our experience, we have seen synthesis tools produce several undesirable systems that have a specialized "sulk state" which is entered when the environment makes its first mistake, and is never left afterwards. We would like to note that the synthesis algorithm proposed in this paper will always yield the *most* robust system, which is something hard to specify in traditional formalisms. Furthermore, classical approaches to synthesizing robust, also called fault-tolerant, systems assume a given fault and a given recovery model, where the former describes the faults the environment can introduce and the later defined the possible counter actions the system can perform. We refer the reader to [22] for an interesting approach to making a system fault tolerant using discrete controller synthesis as well as an extensive study of related work. In our work we try to minimize the effort of specifying the fault and recovery model using quantitative constraints. Intuitively, we allow the environment and the system to behave arbitrarily but ask the system to satisfy a number of properties proportional to the number of properties the environment satisfies.

**Outline.** In Section II, we present the necessary background for our work. Section III gives an example and the main ideas to deal with safety specifications [9]. Section IV provides the same for liveness specifications [6]. In Section V we discuss related work, and we conclude with giving future prospects in Section VI.

## II. Preliminaries

**Alphabet, words, and languages.** An alphabet $A$ is a finite set of letters. A word $w$ is finite or infinite sequence of letters and we use $|w| \in \mathbb{N} \cup \{\infty\}$ to denote the length of $w$. The set of all finite (infinite) words over the alphabet $A$ is denoted by $A^*$ ($A^\omega$).

We consider systems with a set of input signals $I$ and a set of output signals $O$. We define $AP = I \cup O$. We use the signals as atomic propositions in the specifications defined below. Our input alphabet is thus $\Sigma_I = 2^I$, the output alphabet is $\Sigma_O = 2^O$, and we define $\Sigma = 2^{AP}$.

**Example 1.** *Consider a system with one input signal $a$ and an output signal $b$, then $AP = \{a, b\}$ and $\Sigma = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$, where $\{\}$ denotes that $a$ and $b$ are false, $\{a\}$ means that $a$ is true and $b$ is false, $\{b\}$ encodes the opposite evaluation ($a$ false and $b$ true), and $\{a, b\}$ means that both signal are true.*

**Moore machines.** We use Moore machines to represent systems. A *Moore machine* with input alphabet $\Sigma_I$ and output alphabet $\Sigma_O$ is a tuple $M = (Q, q_0, \delta, \lambda)$, where $Q$ is the set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma_I \to Q$ is the transition function, and $\lambda : Q \to \Sigma_O$ is the output function. In each state, the Moore machine outputs a letter in $\Sigma_O$, then reads a letters in $\Sigma_I$, and moves to the next state. The *run* of $M$ on a sequence $x = x_0 x_1 \ldots \in {\Sigma_I}^\omega$ is a sequence $\rho = \rho_0 \rho_1 \ldots \in Q^\omega$, where $\rho_0 = q_0$ and $\rho_{i+1} = \delta(\rho_i, x_i)$. The corresponding *word* is $\lambda(\rho) = w_0 w_1 \ldots \in \Sigma^\omega$, where $w_i = \lambda(\rho_i) \cup x_i$. The *language of* $M$, $L(M) \subseteq \Sigma^\omega$, consists of the words corresponding to the runs of $M$.

**Objective functions and acceptance conditions.** The specifications we use are automata and we synthesize a system that realizes a given specification using games. Both automata and games can be equipped with an objective function. Let $Q$ be a set of states, an *objective function* is a function $\mathrm{Acc} : Q^\omega \to \mathbb{R} \cup \{-\infty, \infty\}$ mapping infinite sequences of states[1] to the set of extended reals. An objective function that ranges only over 1 and 0 (representing accepting and not accepting, or winning and losing, respectively) is called *acceptance condition*.

The *Büchi acceptance condition* is $\mathrm{Acc}(\rho) = 1$ iff $\inf(\rho) \cap F \neq \emptyset$, where $F \subseteq Q$ is a set of *accepting states* and $\inf(\rho)$ is the set of elements that occur infinitely often in $\rho$. We abbreviate the Büchi condition with $\mathcal{B}(F)$. A *Generalized Reactivity (GR) acceptance condition* is a predicate $\bigwedge_{l=1}^{k} (\bigwedge_{i=1}^{m_l} \mathcal{B}(A_{l,i}) \to \bigwedge_{i=1}^{n_l} \mathcal{B}(G_{l,i}))$, where $A_{l,i} \subseteq Q$ are assumptions and $G_{l,i} \subseteq Q$ are guarantees. To simplify notation, we will assume that the $m_l$ are all equal to some constant $m$, and similarly for $n_l$ and $n$. The acceptance condition is a *GR(1) acceptance condition* if $k = 1$, it is a *generalized Büchi acceptance condition* if $k = 1$ and $m = 0$, it is a *Streett acceptance condition* with $k$ pairs if $m = n = 1$.

**Automata.** A *(complete deterministic) automaton* $A$ over the alphabet $\Sigma$ is a tuple $A = (Q, q_0, \delta, \mathrm{Acc})$, where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, and Acc is the acceptance condition. A *run* of an automaton $A$ on a word $w = w_0 w_1 \ldots \in \Sigma^* \cup \Sigma^\omega$ is the sequence $\rho = \rho_0 \rho_1 \ldots \in Q^* \cup Q^\omega$ such that $\rho_0 = q_0$ and $\rho_{i+1} = \delta(\rho_i, w_i)$ forall $i$ with $0 \leq i \leq |w|$ if $w$ is finite, and $i \geq 0$ otherwise. An automaton accepts a word $w$ if its run $\rho$ is accepting ($\mathrm{Acc}(\rho) = 1$); its language $L(A)$ consists of the set of words it accepts. A *Büchi automaton* $A = (Q, q_0, \delta, F)$ is an automaton with a Büchi condition with accepting state set $F$. If there are no edges from non-accepting to accepting states, i.e., $\forall q \in Q \setminus F, \forall \sigma \in \Sigma : \delta(q, a) \in Q \setminus F$, then $A$ is called a *safety automaton*.

**Example 2.** *Figure 1(a) (on page 4) shows a safety automaton reading words over the alphabet $\Sigma = 2^{\{r_1, r_2\}}$. The automaton has two states $s_0$ and $s_1$ depicted as circles. A state depicted with two cycle (e.g., state $s_0$) is an accepting state. Transitions are summarized using edges labeled with Boolean expressions over $r_1$ and $r_2$; a horizontal alignment of two variables represents a conjunction and two vertically aligned variables are disjoint. We use an overline to denote negation and $\top$ to denote true. E.g., the label $\begin{matrix}\overline{r_1}\\ \overline{r_2}\end{matrix}$ of the self-loop on state $s_0$ represents the Boolean expression $\neg r_1 \vee \neg r_2$ satisfied by the three letters $\{\}$, $\{r_1\}$, and $\{r_2\}$. So, there are three transitions from $s_0$ to $s_0$, one for each letter satisfying the Boolean expression, i.e., $\delta(s_0, \{\}) = \delta(s_0, \{r_1\}) = \delta(s_0, \{r_2\}) = s_0$. The automaton accepts all words for which the corresponding run does not visit $s_1$, which are all words that do not use the letter $\{r_1, r_2\}$.*

The (synchronous) *product automaton* $A = A_1 \times A_2$ of

[1]Equivalently, an objective function can also map infinite sequences of pairs of states and letters to a value.

two automata $A_1$ and $A_2$ is defined as usual. The product of two safety automaton is again a safety automaton. Taking the product of two Büchi automata leads to an automaton with a generalized Büchi acceptance condition.

**Specifications.** We consider specifications consisting of two parts: *assumptions* and *guarantees*. They specify the interaction between an environment (controlling the input variables $\Sigma_I$) and a system (controlling the output variables $\Sigma_O$). We use automata to specify an assumption (or a guarantee). In our examples, we also show LTL formulas describing the discussed properties. Readers familiar with LTL [30] will find them useful, while they can be safely ignore by readers not familiar with LTL.

Given $m$ automata $A^a_1, \dots, A^a_m$ for the environment assumptions and $n$ automata $A^g_1, \dots, A^g_n$ for the system guarantees. The specification states that the system must fulfill all guarantees if the environment fulfills all assumptions, i.e.,

$$\varphi = L(A^a_1 \times \cdots \times A^a_m) \to L(A^g_1 \times \cdots \times A^g_n),$$

where $\to$ is defined for two languages $L, L'$ over alphabet $\Sigma$ in the usual sense, i.e., $L \to L' = (\Sigma^\omega \setminus L) \cup L'$. We say that a Moore machine $M$ *satisfies* the specification $\varphi$, if $L(M) \subseteq \varphi$. If all automata ($A_i^{a/g}$) are safety automata, then $\varphi$ is a safety specification. In the case of automata with Büchi conditions, $\varphi$ is called a GR(1) specification [29].

**Single and Double Cost Automata.** A *single (double) cost automaton* over the alphabet $\Sigma$ is a tuple $C = (Q, q_0, \delta, c)$ consisting of a complete deterministic automaton $(Q, q_0, \delta)$ and a cost function $c : Q \times \Sigma \to \mathbb{N}$ ($c : Q \times \Sigma \to \mathbb{N} \times \mathbb{N}$, respectively) that associates to each transition a value in $\mathbb{N}$ ($\mathbb{N} \times \mathbb{N}$, resp.) called *cost*. In a double cost automaton, we use $c_s$ and $c_e$ to refer to the cost function of the first and the second component, respectively. The *maximal cost* is the smallest $W \in \mathbb{N}$ s.t. $\forall q \in Q, \forall \sigma \in \Sigma : c(q, \sigma) \leq W$ ($c_e(q, \sigma), c_s(q, \sigma) \leq W$). The cost of a word $w = w_0 \cdots \in \Sigma^* \cup \Sigma^\omega$, denoted by $C(w)$, is the sum $\sum_{i=0}^{|w|-1} c(\rho_i, w_i)$, where $\rho = \rho_0 \dots$ is the run of $C$ on $w$. For double cost automata, we use $C_e(w)$ and $C_s(w)$ to refer to the first and second component, respectively, of the cost of the word $w$. Note that $C(w)$, $C_e(w)$, and $C_s(w)$ are objective functions assigning to each word over $\Sigma$ a value.

Given a double cost automaton $C = (Q, q_0, \delta, c)$ over $\Sigma$, the *ratio objective* [9] maps every word $w = w_0 w_1 \cdots \in \Sigma^\omega$ to the ratio of the costs accumulated along the run $\rho = \rho_0 \rho_1 \dots$ of $C$ on $w$, i.e.,

$$R(w) = \lim_{n \to \infty} \limsup_{m \to \infty} \frac{\sum_{i=n}^{m} c_s(\rho_i, w_i)}{1 + \sum_{i=n}^{m} c_e(\rho_i, w_i)}. \qquad (1)$$

The limits in Eq. 1 reflect that fact that $w$ has infinite length. The meaning of the formula is as follow: the two sums are the accumulated rewards of $c_s$ and $c_e$ encountered up to position $m$ of the word $w$. The $+1$ in the denominator avoids division by 0 if the accumulated costs are 0. It has no effect in all other cases. The inner limit ($m \to \infty$) tells us to sum the rewards along the entire infinite word. The outer limit ($n \to \infty$) allows us to ignore a finite prefix of the word. In particular, if the sum in the numerator is finite, then this limit ensures that $R(w) = 0$.

The *sum of two cost automata* $A_1 = (Q_1, q_{01}, \delta_1, c_1)$ and $A_2 = (Q_2, q_{02}, \delta_2, c_2)$ is the cost automaton $A = A_1 + A_2 = (Q, q_0, \delta, c)$, where $A$ is the product of the automata $A_1$ and $A_2$ with costs $c = c_1 + c_2$, i.e., $c((q_1, q_2), \sigma) = c_1(q_1, \sigma) + c_2(q_2, \sigma)$. The *product of two single cost automata* $A_1 = (Q_1, q_{01}, \delta_1, c_1)$ and $A_2 = (Q_2, q_{02}, \delta_2, c_2)$ is a double cost automaton $A = A_1 \times A_2 = (Q, q_0, \delta, c)$, where $A$ is the product of the automata $A_1$ and $A_2$ with costs $c = (c_1, c_2)$, i.e., $c((q_1, q_2), \sigma) = (c_1(q_1, \sigma), c_2(q_2, \sigma))$.

**Games.** Two-player games are a generalization of automata, in which the set of states is partitioned into two sets: the set of *Player-1 states* and the set of *Player-2 states*. A game is played by moving a pebble over the state graph in rounds: initially, the pebble is put on some starting state. Then, in every round the owner (Player 1 or 2) of the state holding the pebble moves it along one of the outgoing edges to a successor state, and the next round starts. The sequence of states visited in this way is called a *play*. A function telling Player 1 (or Player 2) how to move the pebble possibly depending on the *history* of the play (i.e., the states visited previously in this play) is called *strategy*. A strategy that depends only on the current state of the game, i.e., it is independent of the history, is called *positional*. We use objective functions to assign values to plays. We consider complementary objectives for the two players: Player 1 tries to minimize the value of a play and Player 2 tries to maximize it (or vice versa). The *Player-1 (optimal) value* of a state is the minimal (maximal) value Player-1 can achieve for any play starting in this state, i.e., for any strategy Player-2 might follow. The *Player-2 optimal value* is defined analogously. A strategy achieving the optimal value is called *optimal strategy*.

For more details and a formal definition of games, we refer the reader to [33]. In this paper, we focus on the definition and application of robustness and abstract from the underlying techniques, so familiarity with game theory is not required.

## III. Robustness wrt Safety Specifications

In this section, we consider specifications $A \to G$, where $A$ and $G$ are safety specifications (or conjunctions thereof). We first present our notion of robustness with respect to these safety specifications. Then, we give an example. Finally, we summarize how to check if a given system is robust and how to construct a robust systems.

### A. Defining Robustness

Our notion of robustness is based on *error functions*, which are used to define a distance between a specification and a given behavior $w$. Intuitively, an error function maps every possible behavior to a value indicating how "close" the behavior is to a correct behavior. In particular, if $w$ satisfies the specification the value is 0 (distance 0). A higher value indicates a higher distance and so a more serious violation of the specification.

**Example 3.** *Consider the specification* $\varphi = \mathsf{always}(a)$ *that requires that the atomic proposition* $a$ *is always true and* *an error function that assigns to each word over the alphabet $\{\{\},\{a\}\}$ the number of times $a$ is false. Then, e.g., the word $w = \{a\}\{a\}\{a\}$ is assigned to $0$ ($w$ satisfies $\varphi$), the word $v = \{a\}\{\}\{\}\{a\}$ has value $2$ ($v$ has two faults), and the word $u = \{a\}\{\}\{\}\{a\}\{\}$ has value $3$.*

Formally, we require the following from an error function.

**Definition 4.** *An* error function *is a function $d : \Sigma^* \cup \Sigma^\omega \to \mathbb{N} \cup \{\infty\}$ that monotonically increases in the sense that if $w'$ is a prefix of $w$ then $d(w') \leq d(w)$. An* error specification *is a pair of error functions $(d_e, d_s)$.*

Error specifications are the specifications we use in the sequel. They provide a measure of "badness" for both the environment behavior (using $d_e$) and the system behavior (using $d_s$). The severeness of a fault depends on the applications, so does the error specification. We assume that error specifications are provided by the user. In the following we discuss how error specifications relate to classical specifications.

**Definition 5.** *A Moore machine $M$* realizes *an error specification $(d_e, d_s)$ if $\forall w \in L(M) : d_e(w) = 0$ implies $d_s(w) = 0$.*

Thus, an error specification induces a classical specification $A \to G$, where $A = \{w \in \Sigma^\omega \mid d_e(w) = 0\}$ and $G = \{w \in \Sigma^\omega \mid d_s(w) = 0\}$ are sets of infinite words. In [9], we also introduced an alternative notion to realizability, called *strictly realizable*, that forbids the system to make mistakes before the environment does. We do not discuss it here, since the problem of strict realizability can be reduced to realizability by adding a simple preprocessing step.

**Definition 6.** *A Moore machine $M$ is* robust *with respect to an error specification $(d_e, d_s)$ if $\forall w \in L(M) : d_e(w) \neq \infty$ implies $d_s(w) \neq \infty$.*

This means that a robust system can recover from a finite environment error. Note that a system can be robust with respect to a specification that it does not realize if it contains a word with a finite system error but no environment error. Error specifications can forbid words by assigning infinite system costs. (In particular, this is possible when such specifications are given by double cost automata, as below.)

In order to calculate the quality of a robust system we want to calculate the largest system error for every environment error.

**Definition 7.** *A Moore machine $M$ is $k$-robust with respect to an error specification $(d_e, d_s)$ if $\exists d \in \mathbb{N} : \forall w \in L^*(M) : d_s(w) \leq k \cdot d_e(w) + d$.*

Obviously, every $k$-robust system is robust, regardless of $k$. Also, every robust system is $k$-robust for some finite $k$, see Theorem 11, i.e., for every finite Moore machine, the growth of the system error is either linear with respect to the environment error or unbounded. This motivates our choice of the robustness measure as a linear function. The definition of $k$-robustness allows us to rank Moore machines with respect to error specifications: A smaller $k$ is better, it means that the system error increases slowly with the environment error. The constant $d$ allows the system finitely many system failures independent of the environment error. In this paper, we focus on the infinite behavior of a machine, and note that $d$ can be bounded by the product of the size of the Moore machine and the maximal weight. We leave minimization of $d$ to future work.

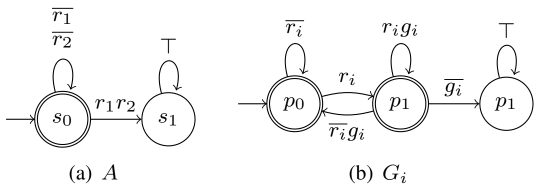

Fig. 1. Automata $A$ for $\mathsf{always}(\neg(r_1 \wedge r_2))$ and $G_i$ for $\mathsf{always}(r_i \to \mathsf{next}\, g_i)$.

**Definition 8.** *A Moore machine ($k$-)*robustly realizes *an error specification if it realizes the specification and it is ($k$-)robust with respect to the specification.*

In the remainder, we use double cost automata to define error specifications. The environment (system) error function associated with $C$ maps each $w \in \Sigma^* \cup \Sigma^\omega$ to its cost $C_e(w)$ ($C_s(w)$, respectively). Note that a double cost automaton can be seen as the product of two single cost automata. We can construct an error specification from a set of cost automata $C_{A_i}$ for the system and $C_{G_i}$ for the environment. The error specification (a double cost automaton) is the product of the sum of all $C_{A_i}$ and the sum of all $C_{G_i}$.

**Example 9.** *Consider the resource controller for two clients. It has two request signals $r_1$ and $r_2$ as inputs and two grant signals $g_1$ and $g_2$ as outputs. We want the system to respond to each request with a grant in the next step. Formally, we require that the system satisfies $G_i = \mathsf{always}(r_i \to \mathsf{next}\, g_i)$ for $i \in \{1, 2\}$. The system should also guarantee that grants are mutually exclusive, i.e., $G_3 = \mathsf{always}\, \neg(g_1 \wedge g_2)$. To avoid a contradicting specification, we assume that requests are also mutually exclusive, i.e., $A = \mathsf{always}\, \neg(r_1 \wedge r_2)$. Figure 1 shows two safety automata, one for $A$ and one for $G_i$. The automaton for $G_3$ is obtained from $A$ by renaming $r_1$ and $r_2$ to $g_1$ and $g_2$, respectively.*

*Starting from the specification $A \to (G_1 \wedge G_2 \wedge G_3)$, we can define what it means for the system and the environment to fail. In particular, the environment violates assumption $A$ if it raises $r_1$ and $r_2$ at the same time. This corresponds to taking the edge from $s_0$ to $s_1$. In Figure 2(a), we show a cost automaton that counts every violation of the environment. Note that once the environment "pays" for taking the edge $r_1 r_2$, we go back to the initial state, resetting the specification. Similarly, if the system violates Guarantee $G_i$ by choosing to go from $p_1$ to $p_2$, it also incurs cost $1$ as shown in Figure 2(b).*

*Note that it is up to the user to define the cost of a violation and the state in which to continue after the specification is violated. A reset or a skip are two natural alternatives. A reset corresponds to an edge to the initial state. For a skip, we*

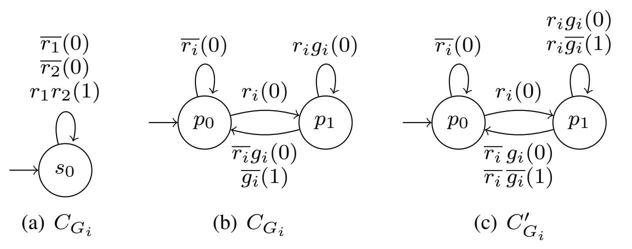

Fig. 2. Cost automata counting violations of $A$ and $G_i$, respectively.

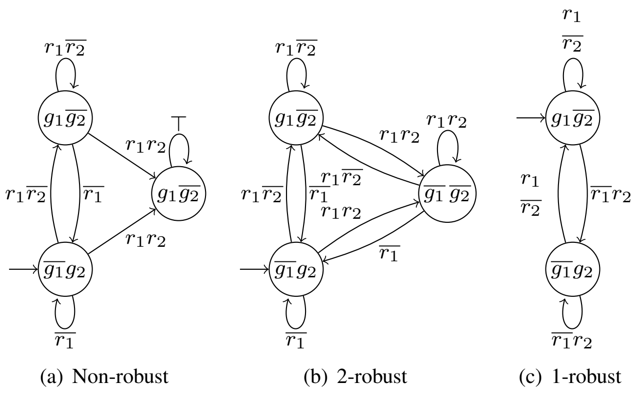

Fig. 3. A non-robust, 2-robust, and a 1-robust system.

*simply add a self-loop. In Figure 2(c) we show an alternative cost automaton for $G_i$ with $i \in \{1, 2\}$, which uses a mixture of reset and skip. For the cost automaton $C_{G_i}$, the word $\{r_1\}\{r_1\}\{\}^\omega$ has cost 1 whereas it has cost 2 for the cost automaton $C'_{G_i}$. (Recall that $\{\}$ denotes the letter in which both signals, $r_1$ and $g_1$, have value false.) For the second automaton, the cost corresponds to the number of unanswered requests.*

*The costs on the edges are given by the user. For instance, the user might consider a violation of the mutual-exclusion properties $G_3$ more severe and associate with it a higher cost than a violation of the response properties $G_1$ or $G_2$.*

*Given cost automata $C_{G_1}$, $C_{G_2}$, and $C_{G_3}$ that describe the cost and the behavior associated with a violation of the corresponding property (cf. Fig. 2), we can construct a cost automaton $C_G = C_{G_1} + C_{G_2} + C_{G_3}$ for $G = G_1 \wedge G_2 \wedge G_3$. The automaton $C_G$ defines the error function of the system. The cost automaton for the environment $C_A$ (cf. Fig. 2(a)) specifies the error function of the environment. The product $C = C_A \times C_G$ is the error specification.*

*Figure 3(a) shows a system $M$ (synthesized with* Lily *[24]) for the specification $A \to G$. It is easy to see that $M$ satisfies $A \to G$. As long as the environment satisfies $A$, which means that it does not provide $r_1$ and $r_2$ simultaneously, the system responds to each $r_i$ with the corresponding $g_i$ in the next step. However, $M$ is not robust with respect to $C$: The input sequence $i = (\{r_1, r_2\}\{r_2\})^\omega$ has cost one, but the output of the system has cost $\infty$.*

*Figure 3(b) and 3(c) show two systems that are robust with respect to the error specification, for any word with finitely many environment errors the systems produce finitely many system errors. The system in Figure 3(b) is 2-robust with respect to the error specification whereas the system in Figure 3(c) is 1-robust. For the input $(\{r_1, r_2\})^\omega$ the output of the first Moore machine is $(\{g_2\}\{\})^\omega$ and for the second it is $(\{g_1\})^\omega$.*

*Note that out of the three systems in Figure 3 (which all satisfy $A \to G$) the system in Figure 3(c) is the most robust one. In our opinion, it is also the one most likely to please the designer.*

## *B. Verification and Synthesis of Robust Systems*

In this section we shortly summarize the key results for verifying and synthesing robust systems. We start with presenting some facts about automata and games with ratio objective, which we use to solve the robust verification and synthesis problem.

**Theorem 10** (Ratio games [9]). *Given a game $G$ with ratio objective, where $|S|$ denotes the size of the state space, $|E|$ the number of edges (transitions), and $W$ the maximal weight, then the following assertions hold:*

1) *Both players have optimal positional strategies in $G$. (If $G$ is an automata, then there are ultimately periodic words achieving the optimal value.)*
2) *The values of all states in $G$ with respect to the ratio objective can be computed in $O(|S|^3 \cdot W^2 \cdot |E| \cdot \log(|S| \cdot W))$ time. If $G$ is an automaton, then the values can be computed in $O(|S|^2 \cdot |E|)$ time.*
3) *Positional optimal strategies for both players can be found in $O(|S|^4 \cdot \log(\frac{|E|}{|S|}) \cdot |E| \cdot \log(|S| \cdot W) \cdot W^2)$ time in $G$.*

The problem of checking if a system is robust can be reduced to checking if an automaton with a Streett objective has an empty language. We can verify $k$-robustness by computing the maximal value over all words in a cost automaton with respect to a ratio objective. These reductions together with Theorem 10 lead to the following complexity bounds. Furthermore, note that any robust system is also $k$-robust for a sufficiently large $k$.

**Theorem 11** (Verification [9]). *Given a Moore machine $M$ with $n_M$ states, and an error specification $C$ over the alphabet $\Sigma$, with $n_C$ states and maximal cost $W$, we can decide if $M$ is robust in $O(n_C \cdot n_M \cdot \Sigma)$ time. Given a $k$, we can check if $M$ is $k$-robust in $O(n_C^3 \cdot n_M^3 \cdot \Sigma)$ time.*

*If $M$ is robust with respect to $C$, then $M$ is also $(n_C \cdot n_M \cdot W)$-robust.*

We can automatically construct a robust (or $k$-robust) system that realizes an error specification by solving a game with Streett objective (or Ratio objective, respectively).

**Theorem 12** (Synthesis [9]). *Given an error specification $C$ with $n$ states and alphabet $\Sigma$, we can decide if a robust and realizing system exists in $O(n^2 \cdot \Sigma)$ time. The system can be synthesized in $O(n^2 \cdot \Sigma)$ time.*

*Given an error specification $C$ with $n$ states, input alphabet $\Sigma_I$, output alphabet $\Sigma_O$, and maximal cost $W$, if a robust and realizing system exists, a $k$-robust system with minimal $k$ that realizes the specification can be synthesized in $O(n^5 \cdot (|\Sigma_I| + |\Sigma_O|)^4 \cdot \log(\frac{(|\Sigma_O|+n\cdot|\Sigma_I|)}{|\Sigma_I|+|\Sigma_O|}) \cdot (|\Sigma_O| + n \cdot |\Sigma_I|) \cdot \log(n \cdot (|\Sigma_I| + |\Sigma_O|) \cdot W) \cdot W^2)$.*

## IV. Robustness wrt Liveness Specifications

In the previous section we discussed the verification and synthesis of robust systems for safety specifications. In the case of safety, environment failures are immediately apparent and the difficulty is how the system can best recover from them. A violation of a liveness property, however, cannot be detected at any point in time [1]. Thus, a system that is robust to liveness failures must attempt to fulfill its guarantees under all circumstances, without knowing whether the environment satisfies the assumptions.

We will define several possible notions of robustness in the presence of liveness, all aiming at maximizing the set of guarantees that is fulfilled for any set of fulfilled assumptions. Suppose a specification has two assumptions and two guarantees. In order for the specification to hold, both guarantees must be met when both assumptions are. A system that meets both guarantees when only one assumption is met is more robust than one that meets one (or zero) guarantees when one assumption is met. We start with an example to illustrate.

**Example 13.** *We consider a variant of the dining philosophers problem [12]. There are $n$ philosophers sitting at a round table. There is one chopstick between each pair of adjacent philosophers. Because each philosopher needs two chopsticks to eat, adjacent philosophers cannot eat simultaneously. We are interested in schedulers that use input variables $h_i$ signifying that philosopher $i$ is hungry and output variables $e_i$ signifying that philosopher $i$ is eating.*

*We have the following requirements. First, an eating philosopher prevents her neighbors from eating. Formally, $G_{1i} = \mathsf{always}(e_i \rightarrow \neg e_{(i-1) mod\ n} \wedge \neg e_{(i+1) mod\ n})$. Second, an eating philosopher eats until she is no longer hungry: $G_{2i} = \mathsf{always}(e_i \wedge h_i \rightarrow \mathsf{next}\, e_i)$. Third, every hungry philosopher eats eventually $G_{3i} = \mathsf{always}(h_i \rightarrow \mathsf{eventually}\, e_i)$. We add the assumption that an eating philosopher eventually loses her appetite: $A_{1i} = \mathsf{always}(e_i \rightarrow \mathsf{eventually}\, \neg h_i)$. Our final specification consists of $n$ assumptions and $3n$ guarantees: $\bigwedge_{i=1}^n A_{1i} \rightarrow \bigwedge_{i=1}^n (G_{1i} \wedge G_{2i} \wedge G_{3i})$.*

*We have synthesized a system realizing this specification for 5 philosophers using our synthesis tool RATSY[2]. The system constructed by RATSY is not very robust: When philosopher 1 violates the assumption by always being hungry, then philosophers 1 and 3 eat forever, while the other philosophers starve. Thus the three guarantees $\mathsf{always}(h_2 \rightarrow \mathsf{eventually}\, e_2)$, $\mathsf{always}(h_4 \rightarrow \mathsf{eventually}\, e_4)$, and $\mathsf{always}(h_5 \rightarrow \mathsf{eventually}\, e_5)$ are violated. A more robust system would let philosopher 3 and 4 take turns, thus violating only two guarantees.*

[2]http://rat.fbk.eu/ratsy/index.php/Main/HomePage

We will consider Generalized Reactivity specifications of rank 1 (GR(1) specifications). GR(1) is an expressive specification formalism with a natural distinction between assumptions and guarantees [29]. Efficient tools exist for GR(1) specifications, which have been used to synthesize relatively large specifications [25], [8]. GR(1) specifications are of the form $\varphi \rightarrow \psi$. Here, $\varphi$ represents the environment assumptions and $\psi$ represents the system guarantees and both $\varphi$ and $\psi$ are given as a set of deterministic Büchi automata. These automata are combined into a product automaton with state space $Q$, transition relation $\delta$, and acceptance condition $\bigwedge_{i=1}^m \mathsf{always\,eventually}\, a_i \rightarrow \bigwedge_{i=1}^n \mathsf{always\,eventually}\, g_i$.

GR(1) specifications do not require any guarantees to be fulfilled when some assumption is violated. An intuitive notion of robustness that prescribes, for any number of environment assumptions that is violated, a minimal number of system guarantees that must still be fulfilled. We have shown in [6] that this and related measures of robustness can be transformed to a specification of the form $\bigwedge_{j=1}^k (\bigwedge_{i=1}^m \mathsf{always\,eventually}\, a_{ji} \rightarrow \bigwedge_{i=1}^n \mathsf{always\,eventually}\, g_{ji})$, which is a Generalized Reactivity (generalized Streett) formula of rank $k$. We address the problem of verification and especially of synthesis of such formulas, which allows us to construct robust systems. In the following subsections we present the measures of robustness and then discuss the solution for verification and synthesis.

### A. Defining Measures of Robustness

In this subsection we discuss how to compare systems with respect to robustness. Usually, multiple systems satisfy a specification, but which one is most robust? The measure of robustness for a safety specification $\varphi \rightarrow \psi$ is the ratio between how often the environment violates $\varphi$ and how often the system violates $\psi$. For specifications with liveness properties, this approach does not work because we cannot count the number of violations of a liveness property. Instead, we propose to count the number of properties violated. In the following we show two different robustness measures, the single and the multiple counting requirements measure. Then we formally state the requirements a robustness measure has to satisfy.

**Single Counting Requirements.** Recall the dining philosophers example with $n = 5$ philosophers (given in Example 13). Suppose system $D_1$ always lets one philosopher eat until she is not hungry anymore and then moves to the next hungry philosopher in a round robin manner. If one philosopher is hungry forever, then no other philosopher gets to eat again. Thus, the violation of one assumption leads to the violation of four guarantees.

Suppose system $D_2$ lets two non-adjacent philosophers eat at the same time until neither is hungry anymore. They take turns in the following order: first philosopher 1 and 3 eat, then philosopher 2 and 4, and last philosopher 3 and 5 eat. If one of the currently eating philosopher is hungry forever, then the two currently eating philosophers eat forever and no other philosopher gets to eat again. Thus, the violation of one assumption leads to the violation of three guarantees. System $D_2$ is thus more robust than system $D_1$.

An even more robust system ($D_3$) is the one described in Example 13. Two philosophers eat at the same time, as soon as one of them is not hungry anymore another philosopher with free chopsticks is allowed to eat. If one philosopher is hungry forever, she eats forever and the other philosophers that are not her neighbors take turns eating. The violation of one assumption leads to the violation of two guarantees.

We specify robust systems by adding restrictions to the original specification. All three systems above satisfy the original specification $\varphi = \bigwedge_{i=1}^{n} A_{1i} \rightarrow \bigwedge_{i=1}^{n}(G_{1i} \wedge G_{2i} \wedge G_{3i})$, but only $D_2$ and $D_3$ guarantee that they violate at most three system guarantees if the environment violates one of its assumptions. Formally, $D_2$ and $D_3$ additionally satisfy

$$\begin{aligned}\psi_1 \;=\; & \Big(\bigvee_{i=1}^{n} \bigwedge_{j\in\{1,\ldots,n\}\setminus\{i\}} A_{1j}\Big) \\ & \rightarrow \Big(\varphi_S \wedge \bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} \bigvee_{k=j+1}^{n} \bigwedge_{l\in\{1,\ldots,n\}\setminus\{i,j,k\}} G_{3l}\Big),\end{aligned}$$

where $\varphi_S = \bigwedge_{i=1}^{n}(G_{1i} \wedge G_{2i})$. The antecedent of the formula states that the environment satisfies $n-1$ out of the $n$ assumptions. The consequent says that the system satisfies all the safety guarantees ($G_{1i}$ and $G_{2i}$) but might violate three of its liveness guarantees.

Note that in general, a robust system cannot violate a safety guarantee in response to a violation of a fairness assumption, since a violation of a fairness assumption can not be detected in finite time.

Since $D_3$ violates at most two system guarantees if one environment assumption is violated, it also satisfies the following formula.

$$\psi_2 = \Big(\bigvee_{i=1}^{n} \bigwedge_{j\in\{1,\ldots,n\}\setminus\{i\}} A_{1j}\Big) \rightarrow \Big(\varphi_S \wedge \bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} \bigwedge_{k\in\{1,\ldots,n\}\setminus\{i,j\}} G_{3k}\Big)$$

These two formulas allow us to distinguish between systems $D_1$, $D_2$, and $D_3$, which satisfy the same base specification but differ in how resilient they are with respect to violated environment assumptions. We propose to use formulas of this type, which relate the number of satisfied assumptions to a number of satisfied guarantees to measure how robust a system is.

Suppose $\mathcal{A}$ is a set of assumptions and $\mathcal{G}$ is a set of guarantees. Let $\mathcal{A}_k = \{A \subseteq \mathcal{A} \mid |A| = k\}$ be the set of all subsets of $\mathcal{A}$ of size $k$ and let $\mathcal{G}_k$ be defined similarly. We can augment the specification with a restriction of the form $(\bigvee_{A\in\mathcal{A}_k} \bigwedge_{A_i\in A} A_i) \rightarrow (\bigvee_{G\in G_l} \bigwedge_{G_i\in G} G_i)$ to check if a system satisfies $l$ guarantees when $k$ assumptions are satisfied. Naturally, a system that satisfies more guarantees with the same number of satisfied assumptions is more robust.

**Multiple Counting Requirements.** In some cases we might want to have a more fine-grained measure of robustness, which cannot be expressed by a single restriction of the form given above. Recall again the dining philosophers example but this time assume there are $n = 7$ philosophers. Suppose system $D_4$ allows two hungry philosophers to eat at the same time. Then, even if one philosopher does not stop eating, the other non-adjacent philosophers can still take turns eating. However, if two philosophers misbehave and they both get to eat (i.e., they do not sit next to each other), they will leave the other five philosophers to starve. Suppose another system $D_5$ allows three philosophers to eat at the same time. Now, if two philosophers misbehave and they both get to eat, the system $D_5$ still allows another philosopher to eat and only four philosophers are left to starve. Both $D_4$ and $D_5$ realize the specification $\varphi$. If we consider the restrictions from above, we see that both systems satisfy the formula $\psi_1$ and $\psi_2$. Our previous measure of robustness cannot distinguish between $D_4$ and $D_5$. Let us add another restriction $\psi_3$ to our specification:

$$\begin{aligned}\psi_3 \;=\; & \Big(\bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} \bigwedge_{k\in\{1,\ldots,n\}\setminus\{i,j\}} A_{1k}\Big) \\ & \rightarrow \Big(\varphi_S \wedge \bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} \bigvee_{k=j+1}^{n} \bigwedge_{l\in\{1,\ldots,n\}\setminus\{i,j,k\}} G_{3l}\Big).\end{aligned}$$

System $D_5$ realizes $\varphi \wedge \psi_2 \wedge \psi_3$ but system $D_4$ does not. We can measure the number of satisfied guarantees for several numbers of satisfied assumptions. The restrictions we add to the specifications are of the form $\bigwedge_{(k,l)\in L}((\bigvee_{\mathcal{A}\in\mathcal{A}_k} \bigwedge_{A_i\in\mathcal{A}} A_i) \rightarrow (\bigvee_{\mathcal{G}\in\mathcal{G}_l} \bigwedge_{G_i\in\mathcal{G}} G_i))$, where $L$ is a list of pairs $(k, l)$, requiring $l$ guarantees to be satisfied if $k$ assumptions are satisfied.

**Definitions.** Both single and multiple counting requirements, as defined above, can be put in the following form.

**Definition 14.** *Given a GR(1) specification $A^{GR(1)}$ with assumptions $J_1^a, \ldots, J_m^a$ and guarantees $J_1^g, \ldots, J_n^g$, a* robustness specification *for $A^{GR(1)}$ has the form*

$$\bigwedge_{l=1}^{k} \left(\bigwedge_{i=1}^{m_l} \mathcal{B}(J_{l,i}^a) \rightarrow \bigwedge_{i=1}^{n_l} \mathcal{B}(J_{l,i}^g)\right),$$

*where $J_{l,i}^a \in \{J_1^a, \ldots, J_m^a\}$ and $J_{l,i}^g \in \{J_1^g, \ldots, J_n^g\}$.*

There is a natural partial order on robustness specifications: If, for each set of satisfied assumptions, a specification $S$ requires a superset of the guarantees required by specification $S'$, then $S$ is more robust than $S'$. Let us denote this order by $\prec$.

**Definition 15.** *A* robustness measure *for a GR(1) specification is a set of robustness specifications together with a total order that respects $\prec$.*

For example, consider again the 'simple counting requirements' robustness specifications from above. A possible total order is $(k = 0, l = |\mathcal{G}|) > (k = 0, l = |\mathcal{G}| - 1) > \ldots > (k = 0, l = 1) > (k = 1, l = |\mathcal{G}|) > \ldots > (k = |\mathcal{A}|, l = 0)$, where $k$ is the number of satisfied assumptions and $l$ the number of satisfied guarantees. Another possible total order is $(k = 0, l = |\mathcal{G}|) > (k = 1, l = |\mathcal{G}|) > \ldots > (k = |\mathcal{A}| - 1, l =$

$|\mathcal{G}|) > (k = 0, l = |\mathcal{G}| - 1) > \ldots > (k = |\mathcal{A}|, l = 0)$. A total order is necessary to synthesize the most robust specification.

To synthesize a robust system, we solve games with the above robustness specifications as objectives. In the following subsection we summarize the results for games and synthesis.

### *B. Verification and Synthesis of Robust Systems*

It was shown in [6] that the synthesis problem for robustness wrt liveness specifications as presented in the previous subsection can be reduced to solving games with generalized reactivity conditions. We first summarize the results of [6], [26] for game solving with generalized reactivity conditions and then present the result obtained for synthesis as a consequence.

**Theorem 16** ([26], [6]). *The following assertions hold:*

1) *Games with GR(1) conditions can be solved in $O(|S| \cdot |E| \cdot m \cdot n)$ time; and*
2) *games with Generalized Reactivity GR(k) conditions can be solved in $O(|S|^k \cdot |E| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$ time;*

*where $|S|$ is the size of the state space of the game; $|E|$ is the number of edges (transitions); and $m$ and $n$ are the number of assumptions and guarantees, respectively.*

The translation of robustness for liveness specifications to games with generalized reactivity conditions and the results for games (Theorem 16) yield the following results for verification and synthesis of systems that are robust wrt to liveness specifications (details in [6]).

**Theorem 17** (Verification [6]). *Given a GR(1) specification $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$, a robustness specification $\bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \rightarrow \bigwedge_{i=1}^{n} \mathcal{B}(G_{l,i}))$, and a system $M$, verifying that $M$ satisfies the robustness specification takes $O(k \cdot m \cdot n \cdot |Q|^2 \cdot |\delta|)$ time.*

**Theorem 18** (Synthesis [6]). *Given a GR(1) specification $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$, and a robustness measure with $h$ robustness specifications $r_p = \bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \rightarrow \bigwedge_{i=1}^{n} \mathcal{B}(G_{l,i}))$, with $1 \leq p \leq h$, and a total order, synthesis of the most robust system can be performed in $O(h \cdot |Q|^k \cdot |\delta| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$ time. The size of the resulting system is $((m+1) \cdot (n+1))^k \cdot k! \cdot |Q|$.*

## V. Related Work

Our notion of robustness for safety specifications defines a system to be robust if a small environment error leads to a small system error. Other approaches are possible. In the continuous domain, it is natural to require systems to be continuous, which guarantees robustness in the sense that a small output error can be guaranteed by an appropriately small input error [23]. This notion is not directly applicable in the discrete setting, because discrete functions are in general not continuous. Consider, for example, a specification that requires that the value of the output $g$ is always true (false) if the initial input $r$ is true (false, respectively): $(r \rightarrow \mathsf{always}\, g) \wedge (\neg r \rightarrow \mathsf{always}\, \neg g)$. Here, a minimal difference in the input, namely a change of the initial input, causes a maximal difference in the output. Another challenge in defining robustness as a form of continuity is to choose an appropriate notion of distance. Doyen et al. [14] proposed in interesting notion of distance, which they called *common suffix distance* for studying the robustness of sequential circuits. The common suffix distance gives the last position in which two sequences mismatch. Their approach to robustness furthermore splits the input signals to a circuit into control and disturbance signals. Intuitively, a circuit is then classified as robust if a finite number of changes in the disturbance values results in a bounded number of changes in the computed outputs. Unlike our notions, this notion does not take the specification into account.

The importance of robustness is widely recognized. Rinard, for instance, advocates acceptability-oriented computing, stating that “complex computer systems should have a natural resilience to errors” [32]. In the context of distributed systems, the notion of fault-tolerance has been studied (cf. [20], [5]) and approaches [27], [16] to retrofit fault tolerance to existing programs have been developed. Closely related to these works is the recent work by Girault and Rutten [22] and the work by Cheng et al. [11]. Both provide a complete framework and tool to add fault-tolerant behavior to a given system based on fault and recovery models using controller synthesis [31]. Our work differs from previous work on fault-tolerance in the sense that our fault and recovery model is very general. Intuitively, we allow the environment as well as the system to make arbitrary faults but we require the faults to be related, meaning that the numbers of violations the system makes is proportional to the number of faults the environment makes.

Attie et. al [4] argue that fault-intolerant programs are often unrealistic. They introduce a framework to specify fault-tolerant concurrent programs with CTL formulas and different levels of tolerance, and show how to synthesize such programs. Contrary to our work, this work considers closed systems and requires the developer to specify possible faults explicitly. Cury and Krogh consider synthesis of robust controllers for discrete event systems, where a controller is optimal if it produces the correct behavior for a maximal set of plants including the original. This approach can beneficially be combined with ours to yield systems that fulfill the guarantees in a maximal set of cases and gracefully degrade otherwise.

Faella [18] considers the question of the appropriate behavior when a game is lost. He considers two notions, one based on dominating strategies and one based on a probability distribution over the input. In the former setting, he maximizes the set of inputs for which the game is won, and in the latter setting, the probability that the game is won. A similar problem is considered in [10], where an unrealizable specification $G$, which corresponds to a lost synthesis game, is generalized to a specification $A \rightarrow G$ for a maximally weak environment assumption $A$. None of these papers considers appropriate behavior in the cases where a system failure is inevitable, which is central to our notion of graceful degradation.

D'Souza and Gopinathan [15] consider a specification that is built from a ranked set of requirements, which may be contradictory. The requirements are “conflict tolerant”, i.e., when overruled, they continue giving "advice." This is achieved through means closely related to our weighted edges. D'Souza and Gopinath describe how to synthesize controllers in which advice from a requirement is alternately followed and ignored. The question they answer is how to synthesize a system that always follows the highest-ranked advice. The approach differs from ours in the focus on contradictory specifications rather than environment failures, and in the fact that the proper action is chosen greedily, whereas we solve a global optimization problem to find the appropriate behavior.

Alur, Kanade, and Weiss [2], consider prioritized requirements and present an efficient way to synthesize the highest-priority requirement. This is related in the sense that the ideal specification may be left unfulfilled if necessary. What is missing, from our perspective, is a way to "return" to a higher-priority requirement.

Eisner considers properties in CTL of the form $\psi = \mathsf{AG}\,\varphi$ ($\varphi$ always holds) and calls a system robust if $\psi$ holds in all states, not just in the reachable states. This implies that the system behaves well in the presence of environment failures (assuming that any invariants used as antecedents are weak), but Eisner states that control-intensive applications are typically not robust [17].

Furthermore, measures of robustness for different fault models, for example internal malfunctions of circuits [19], have been studied. Classical notions of fault tolerance such as self-stabilization [13] and the notions of closure and convergence suggested in [3] focus on safety properties. Convergence requires that a system restores its invariant after an error has occurred, and closure requires that the system satisfies a second, larger invariant even when errors recur. Our approach for liveness can be viewed as an extension of closure to liveness, where we require that some weaker set of guarantees is fulfilled when the environment behaves unexpectedly.

## VI. Conclusion

We have introduced two notions of robustness based on graceful degradation for functional specifications. Both notions assume that specifications are split into a set of assumptions on the environment and a set of guarantees the system should satisfy if the environment adheres to all its assumption.

The first notion addresses the case in which every assumption and every guarantee is given by a safety automaton. In this setting we transform every automaton into an error function that maps every possible behavior to a value indicating how many times this behavior violates specification. So, a higher distance means that the behavior is further away from a correct behavior. Then, a system is said to be robust if for all input traces, the number of errors of the system on this input trace is linearly bounded with some $k$ by the number of errors the environment made on this input trace.

Our second notion of robustness is targeted at liveness specifications. More precisely, we consider specifications in which assumptions and guarantees are given by deterministic Büchi automata. The aim of this notion is to maximize the number of guarantees that are fulfilled for any number of assumptions that may be violated. We have several different interpretations of this notion, which all can be reduced to Generalized Reactivity formulas.

For both notions, we summarized our results on (i) verifying robustness with respect to a specification and (ii) deriving a robust system from a specification.

Our notions of robustness should be seen as only one way of extracting more information from a specification than just correctness. In the future, we expect to see a range of different proposals that aim to augment specifications in order to better capture the user's intentions. In this context, we believe that quantitative techniques will be the key ingredients because they allow the user to state, for instance, robustness independent of a concrete fault or recovery model. This way, the burden on providing complete specification is reduced, while still retaining guarantees on the behavior of the system. Moving to a quantitative setting also gives raise to many interesting theoretical problems in graph games (as we have already seen with ratio games and lexicographic mean-payoff games [7]). For example, our robustness measure for safety leads to a ratio game. A natural extension would be to broaden this measure to systems including liveness, which would yield a ratio-parity game in which the parity condition must be fulfilled while minimizing the ratio.

From a practical point of view, we are currently looking into ways to implement robustness within a GR(1) synthesis setting. For instance, implementing robustness (not $k$-robustness) leads to a GR(2) game, which can reduced to a 2-pair Streett game, which can be implemented efficiently using [28].

We believe that robustness is an important property of reactive systems. It remains to be seen in which ways real-life systems are robust, and if and how robustness checks will be eventually included into industrial verification systems.

## Acknowledgment

This work was supported by EU grants 248613 (DIAMOND) and 248776 (PRO3D), the ERC Advanced Grant QUAREM, the FWF NFN Grant S11402-N23 (RiSE), the EU NOE Grant ArtistDesign and a Microsoft faculty fellowship.

## References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.

[2] R. Alur, A. Kanade, and G. Weiss. Ranking automata and games for prioritized requirements. In *Computer Aided Verification*, pages 240–253, 2008.

[3] A. Arora. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transatcions of Software Engineering*, 19:1015–1027, 1993.

[4] P. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26:125–185, 2004.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[6] R Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, and B. Jobstmann. Robustness in the presence of liveness. In *CAV*, pages 410–424, 2010.

[7] R. Bloem, K. Chatterjee, T. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *Int. Conf. Computer Aided Verification (CAV)*, pages 140–156, 2009.

[8] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *In Proceedings of the Design, Automation and Test in Europe*, pages 1188–1193, 2007.

[9] R. Bloem, K. Greimel, T. Henzinger, and B. Jobstmann. Synthesizing robust systems. In *Proc. Formal Methods in Computer Aided Design (FMCAD)*, pages 85–92, 2009.

[10] K. Chatterjee, T. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *International Conference on Concurrency Theory (CONCUR)*, pages 147–161, 2008.

[11] C.-H. Cheng, H. Rueß, A. Knoll, and C. Buckl. Synthesis of fault-tolerant embedded systems using games: From theory to practice. In *VMCAI*, pages 118–133, 2011.

[12] E. W. Dijkstra. Cooperating sequential processes. In Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[13] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[14] L. Doyen, T. A. Henzinger, A. Legay, and D. Nickovic. Robustness of sequential circuits. In *10th International Conference on Application of Concurrency to System Design, ACSD 2010*, pages 77–84, 2010.

[15] D. D'Souza and M. Gopinathan. Conflict-tolerant features. In *Computer Aided Verification (CAV)*, pages 227–239, 2008.

[16] A. Ebnenasir, S. S. Kulkarni, and A. Arora. Ftsyn: a framework for automatic synthesis of fault-tolerance. *Software Tools for Technology Transfer*, 10:455–471, 2008.

[17] C. Eisner. Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 97–109, Bad Herrenalb, September 1999. Springer-Verlag. LNCS 1703.

[18] M. Faella. Games you cannot win. In *Workshop on Games and Automata for Synthesis and Validation*, 2007.

[19] G. Fey and R. Drechsler. A basis for formal robustness checking. In *ISQED*, pages 784–789, 2008.

[20] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[21] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

[22] A. Girault and É. Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design*, 35(2):190–225, 2009.

[23] T. Henzinger. Two challenges in embedded systems design: Predictability and robustness. *Philosophical Transactions of the Royal Society*, 2008.

[24] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 117–124, 2006.

[25] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification*, pages 258–262, 2007.

[26] S. Juvekar and N. Piterman. Minimizing generalized büchi automata. In *CAV*, pages 45–58, 2006.

[27] S. S. Kulkarni and A. Ebnnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2:1–15, 2005.

[28] N. Piterman and A. Pnueli. Faster solutions of rabin and streett games. In *LICS*, pages 275–284, 2006.

[29] N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380. Springer, 2006. LNCS 3855.

[30] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[31] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.

[32] M. C. Rinard. Acceptability-oriented computing. In *OOPSLA Companion*, pages 221–239, 2003.

[33] W. Thomas. Infinite games and verification. In *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 58–64. Springer-Verlag, 2002. LNCS 2404.

[34] C. von Essen and B. Jobstmann. Synthesizing systems with optimal average-case behavior for ratio objectives. In *International Workshop on Interactions, Games and Protocols (iWIGP)*, pages 17–32, 2011.