A Game-Theoretic Foundation for the Maximum Software Resilience against Dense Errors*

Chung-Hao Huang[1] Doron A. Peled[2] Sven Schewe[3] Farn Wang[1,4]

[1]Graduate Institute of Electronic Engineering, National Taiwan University, Taiwan 106, ROC

[2]Department of Computer Science, Bar Ilan University, Ramat Gan, Israel

[3]Department of Computer Science, University of Liverpool, Liverpool, UK

[4]Department of Electrical Engineering, National Taiwan University, Taiwan 106, ROC

keywords: fault tolerance, resilience, formal verification, model-checking, game, strategy, complexity

Abstract

Safety-critical systems need to maintain their functionality in the presence of multiple errors caused by component failures or disastrous environment events. We propose a game-theoretic foundation for synthesizing control strategies that maximize the resilience of a software system in defense against a realistic error model. The new control objective of such a game is called k-resilience. In order to be k-resilient, a system needs to rapidly recover from infinitely many waves of a small number of up to k close errors provided that the blocks of up to k errors are separated by short time intervals, which can be used by the system to recover. We first argue why we believe this to be the right level of abstraction for safety critical systems when local faults are few and far between. We then show how the analysis of k-resilience problems can be formulated as a model-checking problem of a mild extension to the alternating-time μ-calculus (AMC). The witness for k resilience, which can be provided by the model checker, can be used for providing control strategies that are optimal with respect to resilience. We show that the computational complexity of constructing such optimal control strategies is low and demonstrate the feasibility of our approachh through an implementation and experimental results.

1 Introduction

Today's software systems can consist of tens of million lines of code. Such a system may interact with hundreds of distributed processes that are created and destroyed dynamically in an evolving environment. With such a scale of complexity and unpredictability, users and developers have learned to deal with the reality that software systems most likely still contain defects after delivery. In fact, various empirical studies show that the defect density of commercial software systems varies from 1 to 20 defects in every 1000 lines of source code [41]. Programmers and software

*This article is an extended version of [25]. All tool implementation and related experiment materials are available at `https://github.com/yyergg/Resil`. The work is partially supported by NSC (Grant NSC 97-2221-E-002-129-MY3, Taiwan, ROC). For more information, please email to `farn@ntu.edu.tw`.

designers have developed many engineering techniques to contain the damage that could be caused by such defects. For example, when observing that a critical service request is not acknowledged, a software system may have several measures to its disposal to avoid system failure, including resending the request, resetting the server, clearing the communication buffers, etc. But, in general, it is difficult to estimate how to organize the measures for the maximal resilience of the system against realistic errors. At the moment, an automated support for the synthesis of control mechanism to defend a system against software errors is missing. Such an automated support, if available, can suggest defense techniques against software defects to development teams, and help these development teams to identify the vulnerabilities of software systems. We use a game-theoretic approach to study this aspect and have carried out experiments to observe how our techniques can be used in synthesizing the most resilient defense of software systems against multiple errors.

Intuitively, the defensive strength of a software system should be proportional to the number of errors that it can endure. A subtle issue in designing the foundation is the realistic assumption on how many errors a system can endure before running into disasters. Apparently, no non-trivial system can endure an unlimited flood of errors without degrading to inevitable system failure. Thus, if we do not employ a realistic error model, then no meaningful analysis of the resilience level of these systems to software errors can proceed, and no practical control mechanism can be devised to defend them against errors. We are interested in fending the system against a more restricted error model, but still want to provide the error model with a quantifiable level of power in order to be able to defend the system against many error scenarios.

Considering that most software systems have a life-time much longer than the duration needed for a reasonably designed software system to recover from an error, a reasonable foundation needs to take the difference between these two time scales into account. In this work, we propose to evaluate control mechanism of software systems on how many errors the control can endure before recovery to safe behavior. We then present an algorithm to synthesize a control strategy that can endure the maximal number of such errors.

Before proceeding further, let us standardize the basic terms. In embedded systems, a design defect in software or hardware is called a *fault*. Different to a fault, an *error* (sometimes called *component failure* in the literature) is the effect of a fault that results in a difference between the expected and the actual behavior of a system, e.g., measurement errors, read/write errors, etc. An error does not necessarily lead to a system failure, but may instead be repaired by, e.g., a defense mechanism in the software. That is, an error may be detected and corrected/neutralized before it creates any harm to the whole system or its users. Only when the effect of an error creates faulty behaviors that can be observed by the users, it becomes a *failure*.

Our specific goal is to develop a technique for synthesizing a control mechanism of a software system against the maximal number of dense errors without degrading to failure. We took our inspiration from methods for resilient avionic systems [39], where fault tolerance is designed to recover from a *bounded* number of errors. The number of errors a system needs to tolerate can be inferred from the

k	0	1	2	3	4	5	6	...
k errors	0.865	0.594	0.333	0.143	0.053	0.017	0.005	...
k dense errors	0.865	$2 \cdot 10^{-4}$	$2 \cdot 10^{-9}$	$2 \cdot 10^{-14}$	$2 \cdot 10^{-19}$	$2 \cdot 10^{-24}$	$2 \cdot 10^{-29}$	...

Table 1. Probabilities of k dense errors

given maximal duration of a flight and the mean time between errors of the individual components. To demonstrate the difference between the objective to tolerate up to k errors and sequences of separated blocks of up to k *dense errors* in a short period, we exemplify the quality guarantees one obtains for a system (e.g., an airplane) with an operating time of 20 hours and a mean time between exponentially distributed errors of 10 hours, assuming a repair time of 3.6 seconds. The mean time between dense errors (consecutive errors before system recovery) is calculated in Table 1. The figures for k errors (component failures) are simply the values for the Poisson distribution with coefficient 2. To explain the figures for k dense errors, consider the density of 2 dense errors occurring in close succession. If an error occurs, the chance that the next error occurs within the repair time (3.6 seconds) is approximately $\frac{1}{10000}$. The goal to tolerate an arbitrary number of up to k-dense errors is, of course, much harder than the goal of tolerating up to k errors, but, as the example shows, the number k can be much smaller. Tolerating an arbitrary number of errors (with a distance of at least 3.6 seconds between them) creates the same likelihood to result in a system failure as tolerating up to 9 errors overall, and tolerating up to 15 errors still results in a 70% higher likelihood of a system failure than tolerating blocks of up to 2 errors in this example. Only errors for which this is the case could cause a system failure. The mean time between blocks of two dense errors is therefore not ten hours, but 100,000 hours. Likewise, it increases to 1,000,000,000 (one billion) hours for blocks of three dense errors, and so forth. Maximizing the number of dense errors that are permitted before full recovery is therefore a natural design goal. After full recovery, the system is allowed again the same number of errors. Now, if the *mean time between errors* (*MTBE*) is huge compared to the time the system needs to fully recover, then the mean time between system failures (MTBF) grows immensely.

We view the problem of designing a resilient control mechanism towards dense errors as a two-player game, called *safety resilience game*, between the system (protagonist[1], 'he' for convenience) and a hostile agent (antagonist[2], 'she' for convenience) that injects errors into the system under execution. The protagonist wants to keep the system from failure in the presence of errors, while the antagonist wants to derail the system to failure. Specifically, system designers may model their system, defense mechanism, and error model as a finite game graph. The nodes in the graph represent system states. These system states are partitioned into three classes: the safe states, the failure states, and the recovery states. Some transitions are labeled with errors while others are considered normal transitions. The game is played with respect to a resilience level k. If a play ever enters a failure state, then the antagonist wins in the play. Otherwise, the protagonist wins.

The protagonists plays by selecting a move, intuitively

[1] In game theory, a protagonist sometimes is also called *player 1*.
[2] In game theory, an antagonist sometimes is also called *player 2*.

the 'normal' event that should happen next (unless an error is injected). The antagonist can then decide to trigger an error transition (injecting an error) with the intention to eventually deflect the system into a failure state. Our error model, however, restricts the antagonist to inject at most k errors before she allows for a long period of time that the system may use to recover to the safe states. (If the antagonist decides to use less than k errors, the protagonist does not know about this. It proves that this information is not required, as we will show that the protagonist can play memoryless.) After full recovery by the protagonist to the safe states, the antagonist is allowed again to inject the same number of errors, and so forth.

If the system can win this game, then the system is called *k-resilient*. For k-resilient systems, there exists a control strategy—even one that does not use memory—to make the system resilient in the presence of blocks of up to k dense errors. We argue that, if the component MTBF is huge compared to the time the system needs to fully recover, then the expected time for system breakdown grows immensely.

Besides formally defining safety resilience games, we also present algorithms for answering the following questions.

- Given an integer k, a set F of failure states, and a set S of safe states (disjoint from F), is there a recovery mechanism that can endure up to k dense errors, effectively avoid entering F, and quickly direct the system back to S. Sometimes, the system designers may have designated parts of the state space for the recovery mechanism. The answer to this question thus also implicitly tells whether the recovery mechanism is fully functional in the recovery process.
- Given an integer k and the set of failure states, what is the maximal set of safe states, for which the system has a strategy to maintain k-resilience? In game theory, this means that safety resilience games can be used for synthesizing safety regions for a given bound on consecutive errors before the system is fully recovered.

  The question can be extended to not only partition the states into safety, recovery, and failure states, but also for providing memoryless control on the safety and recovery states.
- Given a set of failure states, what is the maximal resilience level of the system that can be achieved with proper control? We argue that this maximal resilience level is a well-defined and plausible indicator of the defense strength of a control mechanism against a realistic error model.

With our technique, software engineers and system designers can focus on maximizing the number of dense errors that the system can tolerate infinitely often, providing that they are grouped into blocks that are separated by a short period of time, which is sufficient for recovery.

We investigate how to analyze the game with existing techniques. We present an extension to alternating-time μ-calculus (AMC) and propose to use the AMC model-checking algorithm on concurrent games to check resilience levels of embedded systems. We present reduction from safety resilience games to AMC formulas and concurrent game structures. Then we present a PTIME algorithm for answering whether the system can be controlled to tolerate up to a given number of dense errors. The algorithm can then be used to find the maximal resilience level that can

be achieved of the system. The evaluation is constructive: it provides a control strategy for the protagonist, which can be used to control a system to meet this predefined resilience level.

The remainder of the article is organized as follows. Section 2 reviews some standard terminology and results. Section 3 outlines our work and motivates it on three examples. Section 4 defines safety resilience game. Section 5 defines a variation of the alternating-time μ-calculus (AMC) for specifying our k-resilience properties. Section 6 presents our resilience level evaluation algorithm. We report on our implementation and the experimental evaluation of our techniques in Section 7. Section 8 reviews related work. Finally, Section 9 summarizes the work.

2 Two-player concurrent game structures

To facilitate our explanation of resilience analysis in a game's perspective, we start by reviewing the game concepts related to our work. A concurrent game may involve several players, who make concurrent move decisions at the same time during transitions. The destination of a transition is jointly determined by the moves chosen by all players. Such a game model is very expressive and handy in describing interactions in a complex system. In this work, we adapt the finite concurrent games from [3] with event concepts on transitions. For the analysis of system resilience, we only have to consider two players in the game, the first is the system, and the second is the error model.

Definition 1 (2-player concurrent game structure): *A* concurrent *game structure is a tuple* $\mathcal{K} = \langle Q, r, P, \lambda, E_1, E_2, \delta \rangle$, *where*

- Q *is a finite set of states.*
- r *is the initial state in* Q.
- P *is a finite set of atomic propositions.*
- $\lambda : Q \mapsto 2^P$ *is a proposition-labeling function of the states.*
- E_1 *and* E_2 *are finite sets of move symbols that the protagonist and the antagonist can respectively choose in transitions. A pair in* $E_1 \times E_2$ *is called a* move vector.
- δ *is a function that maps from* $Q \times E_1 \times E_2$ *to* Q. δ *is called the transition function and conceptually specifies a successor state that results from a state and moves of the players.*

Given a state $q \in Q$ *and a vector* $[e_1, e_2] \in E_1 \times E_2$, $\delta(q, e_1, e_2)$ *is the successor state from* q *when each player* $a \in \{1, 2\}$ *chooses her respective move* e_a. ■

We prefer to represent the moves available to the players by symbols (rather than integers as in [3]), as move (or event) symbols can be used to reflect some physical meaning. For example, a move can correspond to the turning-off of a switch, the detection of an airplane, or the execution of an error handling routine. (Technically, representing moves as either integers or symbols does, of course, make no difference.)

For convenience, we assume that we are in the context of a given 2-player concurrent game structure $\mathcal{K} = \langle Q, r, P, \lambda, E_1, E_2, \delta \rangle$. In the following, we review some standard concepts from game theory.

Definition 2 ***(Plays and play prefixes):*** *A* play prefix ρ *of length* h *is a sequence* $q_0, \overrightarrow{e_0}, q_1, \overrightarrow{e_1}, \ldots, q_{h-1}$ *that alternates between states and move vectors (starting and ending in a state), such that, for all* $i \in [0, h)$, $\delta(q_i, \overrightarrow{e_i}) = q_{i+1}$ *holds. Similarly, a* play ρ *is an infinite sequence*

$q_0, \overrightarrow{e_0}, q_1, \overrightarrow{e_1}, q_2, \overrightarrow{e_2}, \ldots$ *that alternates between states and move vectors (starting and ending in a state), such that* $\delta(q_i, \overrightarrow{e_i}) = q_{i+1}$ *holds.*

In both cases, we use $\rho(i) = q_i$ *and* $\rho_e(i) = \overrightarrow{e_i}$ *by abuse of notation.* ■

The following notations are for the ease of presentation. Given a play prefix $\rho = q_0, \overrightarrow{e_0}, q_1, \overrightarrow{e_1} \ldots q_{h-1}$, we denote the *length* of ρ, h, by $|\rho|$. For plays, we write $|\rho| = \infty$. Given two integers j and h in $[0, |\rho|)$ with $j \leq h$, we use $\rho[j, h]$ to denote the play prefix $\rho(j), \rho_e(j), \rho_e(j+1), \rho(j+1), \ldots, \rho(h)$. For play prefixes ρ, we use $last(\rho) \stackrel{\text{def}}{=} \rho(|\rho| - 1)$ to denote the last state in ρ.

We may also use regular expressions to represent sets of play prefixes. Specifically, given two sets A and B of play prefixes, AB represents the set of concatenation of play prefixes $\rho_1\rho_2$ such that $\rho_1 \in A$ and $\rho_2 \in B$. A^* then represents finite concatenation of play prefixes from A. For example, $a, abc, abcacbcbcbc$ are all elements of $\{a, bc, ac\}^*$.

Please recall that a play has infinite length. A play ρ with $\rho(0) = q$ is called a *q-play*. When choosing moves at a state, a player may look up the play prefix that leads to the current state, investigate what decisions the other players have made along the prefix, and select his or her next move. Such decision-making by a player can be captured by a strategy.

Definition 3 ***(Strategy)*** *A* strategy *is a function from finite play prefixes to a move symbol. Formally, a strategy* σ_a *for a player* $a \in \{1, 2\}$ *is a function from play prefixes to* E_a*. The next state after a play prefix* $\rho \in \big(Q(E_1 \times E_2)\big)^* Q$ *is determined as* $\delta(last(\rho), \sigma_1(\rho), \sigma_2(\rho))$*.*

A strategy σ *is* memoryless *(*positional*) if the choice of* σ *only relies on the current state, that is, if, for every two play prefixes* ρ *and* ρ'*,* $last(\rho) = last(\rho')$ *implies* $\sigma(\rho) = \sigma(\rho')$*. If* σ *is not memoryless, it is called* memoryful*.* ■

Given regular expressions [24] $\eta_1, \ldots, \eta_n$ with alphabet Q and move symbols $e_1, \ldots, e_n \in E$, we may use $[\eta_1 \mapsto e_1, \ldots, \eta_n \mapsto e_n]$ to (partially) specify a strategy. For a strategy σ, a rule like $\eta_i \mapsto e_i$ means that, for every play prefix $\rho \in \eta_i$, $\sigma(\rho) = e_i$. To disambiguate the interpretation of the strategy, a rule with index i supercedes all rules with indices $> i$. Moreover, to make a strategy complete, we may require η_n to be $\big(Q(E_1 \times E_2)\big)^* Q$, the set of strings of interleaving states and move vectors that end in a state (which includes the set of all play prefixes). For example, a memoryless strategy of the protagonist can be specified with $[\big(Q(E_1 \times E_2)\big)^* q_0 \mapsto e_1, \big(Q(E_1 \times E_2)\big)^* q_3 \mapsto e_2, \big(Q(E_1 \times E_2)\big)^* \mapsto e_3]$. A memoryful strategy of the protagonist can be specified with $[q_0 \mapsto e_1, \big(Q(E_1 \times E_2)\big)^+ q_0 \mapsto e_2, \big(Q(E_1 \times E_2)\big)^* q_3 \mapsto e_2, \big(Q(E_1 \times E_2)\big)^* Q \mapsto e_3]$.

Note that, in Definition 3, we do not distinguish between the strategies of the players. We call a play ρ *σ-conform* for a strategy σ of player a if, for all $i \in \mathbb{N}$, there are e_1 and e_2 with $\rho(i+1) = \delta(\rho(i), e_1, e_2)$ and $e_a = \sigma(\rho)$.

In the remainder of the article, we denote the set of all strategies by Σ and the set of all memoryless strategies by $\Sigma^{(0)}$. Together with an initial state r, strategies $\sigma_1, \sigma_2 \in \Sigma$ of the m players respectively, define a unique play, which conforms to σ_1, σ_2. We denote this play by $\mathsf{play}(\mathsf{r}, \sigma_1, \sigma_2)$.

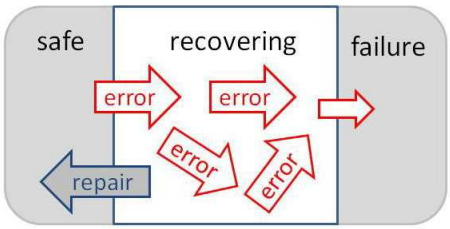

Figure 1. Framework of resilience design

3 Motivation

3.1 Background

Resilience to errors in computer systems is usually achieved through error recovery design as illustrated in Figure 1. The system states can be partitioned into three regions: safe, recovery, and failure. The left part of the figure represents the safety region. The states in this safe region can be viewed as those for 'normal' operation. When an error occurs, the system goes through a recovery stage, where it follows some recovery mechanism. This is shown as the "recovering" area in Figure 1. In this region, the system intuitively tries to repair the effects of an error and thus to recover to the safety region.

During the recovery (or: in the recovery region), however, errors may still happen. In general, fault-tolerant systems are built under the assumption that error detection and recovery is speedy and that there can only be a few errors during the process of recovery. If the recovery mechanism is not resilient enough, a few errors may drive the system into failure.

We illustrate this on the following examples.

Example 1 (Fault-tolerant computer architectures): *In computer architectures, fault-tolerance is usually achieved via hardware duplication. Consider an example of a multiprocessor system that includes n processor copies and m memory copies. The n processors each can follow the instructions of the original system, or be engaged in memory recovery. When a copy of the memory fails, a processor can be assigned to recover it. Majority check can be used to detect that a processor is faulty or that memory copy is faulty (often, both would happen at the same time). For recovery, we can set a free processor to recover some memory copy, or make a processor follow the code of the majority of processors.*

The key to error resilience is to decide whether to make a processor follow the execution of the majority, or to assign it to recover faulty memory. If too many errors occur in a short while before the errors can be recovered from, then there may be no more processors left to carry out any more recovery. When such a critical situation arises, the system enters failure state when another error is induced.

The recovery mechanism described above is typical in the design of fault-tolerant systems [36]. As explained, a practical recovery mechanism usually does not rely on the detailed structure of the system. Instead, error-detection techniques such as parity checks, voting (for majority checks), etc., are usually employed. In fact, the number of duplicates is usually critical to the resilience of the system to errors. As long as the majority of the duplicate modules can be recovered in time (i.e., before the next wave of errors), resilience of the system can be achieved. ■

Example 2 (Exception handling): *At the operating system level, errors are usually signaled via interrupt lines and handled with routines called handlers. The first thing that needs to be done by a handler is to save the CPU state of*

the interrupted process. In some operating systems, a static memory space is used for this purpose for each handler. In such a scheme, if the same error happens again while executing the error handler, then the system can run into the risk that the CPU states of the interrupted handler can be overwritten and destroyed.

Another scheme is to use a stack to save the CPU states of the interrupted processes. Such a scheme seems resilient to errors that happen during the execution of error handlers. Still, too many errors that happen during the execution of error handlers can deny critical functions of the system and incur failures, including missed timer updates and priority inversions. Thus, a proper assumption on the timely error recovery by the error handling routines is critical to the design of error resilience in such cases. ■

Example 3 (Security attacks): *Security in the Internet also relies on resilience to attacks of hackers, viruses, malware, etc. For example, one common technique of attacks to communication modules is to overflow the communication buffers. In such attacks, the sizes of the buffers and the ability of the security procedures to detect and recover from such overflowing attacks is crucial to the resilience design.* ■

These examples show that recovery is a crucial concept for designing systems that are resilient to errors. When system errors are detected in such a system, the system activates a recovery mechanism so as to remove the effect of the errors. When designing such systems, the system designers usually have in mind what errors and failures the systems can expect, according to the specification. To avoid failures in the occurrence of dense errors, the system designers usually incorporate many error recovery mechanism in the system, e.g., exception handlers and hardware/software redundancy. But, in general, it would be difficult for the designers to evaluate how effective their recovery mechanism is to dense errors. To overcome this difficulty, we believe that it is important to support them with automated analytical tools with a solid foundation.

Resilience has also been used in [8, 19] with a similar goal. When synthesising code, one relies on assumptions of the behavior of the environment, and the formal specification would only ask for the provision of guarantees under the condition that the assumptions are satisfied. When assessing the quality of an implementation, the behavior in cases where the environment does not comply with the assumption matters. In [8, 19], the resilience model we have introduced in the conference version [25] of this paper has been followed up upon, and proven to be well suited for reactive synthesis.

In this work, we use these observations to design a theoretical framework for synthesizing a control mechanism that provides the maximal resilience against software errors in a realistic error model.

3.2 Resilience in a Nutshell

From Example 1 to 3 in Subsection 3.1, it is easy to see the common paradigm of error recovery in software systems.

> When errors are detected, a recovery mechanism will be activated to avoid failures and try to get back to normal execution.

Moreover, such a recovery mechanism usually needs to operate under the assumption that more errors may also hap-

pen during the recovery process. In practice, system designers have already implemented many defensive modules, e.g., exception handlers, which are certainly good candidates for the recovery segments. Thus, the recovery scheme we discuss is likely to have arisen in an ad-hoc fashion as a natural concept when software architects and programmers designed recovery mechanisms for critical software.

The vast state spaces of critical systems make an automated support for and a solid foundation of evaluating design alternatives particularly valuable.

In the following, we will use the examples from the previous subsection as a motivation for defining a new game, called *safety resilience game*, between the recovery mechanism (the protagonist) and the error-injecting agent (the antagonist). The game is specified with a set F of failure states, a set S of safe states (the safety region), the moves by the antagonist to inject errors, and the resilience level k that the designers want to achieve. The objective of the protagonist is to identify a control strategy so that the whole system can achieve the prescribed level (or the highest level) k of resilience for safety region S (a set of states) and failure state set F.

The game is played round by round. When the antagonist issues an error move, the play may be deflected into a recovery segment. If there are no more than $k-1$ errors in the recovery segment, then a k-resilient control mechanism must direct the recovery segment to end at a safe state. The above observation suggests that a safety region can be abstracted as a fixed point to the recovery procedure that transforms a safe state to another safe state via the recovery segment with at most $k-1$ errors. Conceptually, a fixed point to a procedure $f(x)$ is a set S of elements in the domain of x such that $S = \{f(x) \mid x \in S\}$. To calculate the fixed point of the recovery procedure, we can use the greatest fixed point algorithm. The idea is to start from a superset of the recovery procedure fixed point. For convenience, we call a superset of the fixed point a *pseudo fixed point* (*PFP*). Then we iteratively check every state q in the PFP and eliminate q from the PFP if, after at most k errors from q, the recovery mechanism either cannot avoid failure or cannot direct the system back to the PFP. As the iterative checking and elimination goes on, the PFP will shrink and eventually stabilize. Note that its size is always finite, since the initial PFP must be no bigger than Q. The final PFP is then a greatest fixed point to the recovery mechansim for k-resilience and is the legitimate safety region.

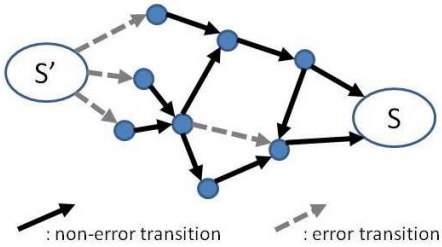

Figure 2. Illustration of the recovery operation

This recovery procedure can be illustrated as in Figure 2 for resilience to 2 errors. In this figure, the states in set S' are computed as the precondition of states in S through those transitions in the figure. Each path from S' to S is a recovery segment. S and S' may overlap. The blue circles represent states in the recovery segments. If we calculate S' out of S, then, for each state $q' \in S'$, we can find a path from $q' \in S'$ via a path in the recovery segment to another state $q \in S$. The maximal number of errors in a recovery segment is 2. Thus the protagonist has a strategy

to recover from errors in S' to S even when 2 errors happen in the corresponding recovery segment. When $S' = S$, then S is a fixed point to the precondition operator through the recovery segments in the figure.

Now we formally define the concept that we explained with Figure 2.

Definition 4 (k-safety): *Given a $k \in \mathbb{N}$, a state q is called k-safe with respect to a safety region $S \subseteq Q \setminus F$ of non-failure states, denoted $q \in \mathsf{sfrch}_k(S)$, if there is a strategy for the protagonist to guarantee that we can reach back to S from q, provided that the overall count of errors is at most k.* ■

However, the definition can be subtle in its interpretation. Specifically, the ability to stand against one wave of k errors is not the same as that against repeated recovery from waves of k errors. If the recovery mechanism is not designed properly, the system may gradually lose a bit of control after each wave of k errors and eventually degrade to system-level failure.

Example 4 (Fault-tolerant computer architectures): *Consider Example 1 with $2k + 1$ processor copies, with the objective to maintain majority checks and to identify the bad processors. Indeed, according to the first, naïve solution, any safe state with a recovery strategy to $Q \setminus F$ is good. After k processor copies fail, the majority checks are still capable to maintain the correctness of the combined behavior to follow the design of the original system. There seems to be nothing to do after k errors. Thus, naïvely, we can choose those states as the safety region if, at those states, majority checks still work.*

However, there is no expectation that the system will be able to recover at any point in the future into a situation where it can bear another wave of k errors. It will fail and lose the function of majority checks just after one more error. In contrast, in this work, we aim to propose a dense error resilience criterion that given no more errors for enough time to allow recovery, the system will eventually recover to resilience to k dense errors again. ■

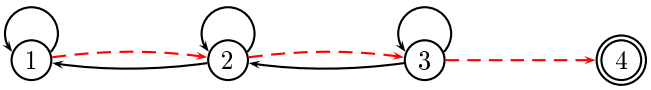

Figure 3. An example for calculating sfrch_k

To look at this issue in more detail, please consider the transition system with four states, including a single failure state (state 4, marked by a double line) shown in Figure 3. The controlled transitions are depicted as black solid arrows, the error transitions are depicted as red dashed arrows. For $S = Q \setminus F = \{1, 2, 3\}$, all states in S are in $\mathsf{sfrch}_0(S)$. For all $k \geq 1$, we have $\mathsf{sfrch}_k(S) = \{1, 2\}$: the protagonist can simply stay in $\{1, 2\}$ during the safety phase of the game, and once the antagonist plays an error transition, the game progresses into the recovery segment, where the protagonist's objective is satisfied immediately. This outlines the difference between k-sfrch-ty and the linear time property of being able to repeatedly tolerate waves of up to k errors, which would only be satisfied by states 1 and 2 for $k = 1$, and only for state 1 for $k = 2$.

This difference raises the question if the rules of our game are depriving the antagonist of some of the k errors that she should intuitively be allowed to insert in a wave. The answer is that this is not the case if we use any fixed point of sfrch_k as S. In this case, the protagonist would regain the capability to endure a wave of k errors when

reaching a safe state after recovery. Instead of depriving the antagonist, one could say that we reset the number of errors in any recovery segment that the antagonist can inject to k. Thus such a fixed point of sfrch_k should consist of states, from which we can use a control mechanism to fend off repetitive waves of k dense errors in the recovery segments. For convenience, we call states in such a fixed point of sfrch_k the k-resilient states.

For a state to be in $\mathsf{sfrch}_k(S)$, the system (protagonist) has a strategy to recover to S, given that a long enough execution commenced without another round of k errors happening. We say that two successive errors are in the same *group of dense errors* if the sequence of states separating them was not long enough for recovery to the safety region. Vice versa, if two successive errors are far enough apart such that the protagonist can guarantee recovery in this separation, then they do not belong to the same group.

To check whether recovering to S by the protagonist (the fault-tolerance mechanism) is always possible, provided that at most k errors occurred during a recovery segment, observe that nesting sfrch_k once, i.e., $\mathsf{sfrch}_k(\mathsf{sfrch}_k(\cdot))$, corresponds to tolerating up to two rounds of up to k dense errors, and so forth. Thus, for S to be a target of recovery for k-resilience, S must be a fixed point of the operator sfrch_k from Definition 4, or, equivalently, $S = \mathsf{sfrch}_k(S)$ must hold. Moreover, if S is the greatest fixed point to k-resilience, then we we can apply $\mathsf{sfrch}_k()$ any number of times to S and still obtain S. Computationally, the greatest fixed point of sfrch_k can be constructed as by executing

$$\mathsf{sfrch}_k(\mathsf{sfrch}_k(\mathsf{sfrch}_k(\ldots\ \mathsf{sfrch}_k(S)\ \ldots))),$$

using a sufficiently deep nesting that a fixed point is reached.

Note that this fixed point x to $x = \mathsf{sfrch}_k(x)$ is what we are really interested in, while $\mathsf{sfrch}_k(S)$ for a given S is an intermediate result that does not guarantee survival of the systems after waves of dense errors. If this greatest fixed point

$$R = \bigcup\{X \subseteq S \mid X = \mathsf{sfrch}_k(X)\}$$

is non-empty, the protagonist's strategy for the fixed point (guaranteeing eventual recovery to a state in the fixed point within no more than k errors, i.e., k-resilience) can be used to control the recovery mechanism, constraining its transitions to follow its winning strategy.

As explained in the introduction, there can be several natural control problems in our safety resilience game. First, the system designers may want to know whether the chosen safety region S can be supported by the recovery mechanism for resilience level k. Second, they may want to get design support for choosing the safety region for achieving resilience level k. Finally, they may want to know the maximal resilience level that they can achieve.

With the explanation in the above, in the rest of the manuscript, we will focus on the algorithm for constructing $\mathsf{sfrch}_k(\cdot)$ and evaluating k-resilient states.

4 Safety resilience games

A system is k-resilient if it can be controlled to tolerate infinitely many groups of up to k dense errors, provided that the system is given enough time to recover between these groups. As we have explained, in systems developed with defensive mechanism against errors, when errors are detected, recovery procedures should be activated. The major challenge is to decide given a set of failure states and a safety region, whether the recovery mechanism can sup-

port a resilience level required by the users. Our goal is to develop techniques with a solid foundation to assist the system designers in evaluating the resilience of their systems, to synthesize the controller strategy for the required resilience level, and to achieve the maximal resilience level.

We now formally define the safety resilience game played between a system (the protagonist) and an error-injector (the antagonist). Initially, the two players are given a 2-player concurrent game structure $\mathcal{K}$, a pebble in r, a set $F \subseteq Q$ of failure states, and a safety region $S \subseteq Q \setminus F$. Then the recovery region consists of states in $Q \setminus (F \cup S)$. The two players together make decisions and move the pebble from state to state. The antagonist tries to deflect a play into F by injecting sufficiently many errors, while the protagonist tries to avoid that the pebble reaches F. To achieve this, the protagonist can use the recovery region as the safety buffer and try to get back to S as soon as the play is deflected from S to the recovery region. If a system is resilient to k errors, then it means that the protagonist can handle up to $k-1$ errors while in the recovery region. Thus when checking whether a system is resilient to k errors, we only need to check those recovery segments with no more than $k-1$ errors.

In the following, we formalize the concept.

Definition 5 (Safety resilience game structure): *Such a structure is a pair $\langle \mathcal{K}, F \rangle$ with the following restrictions.*

- *$\mathcal{K}$ is a 2-player concurrent game structure $\langle Q, r, P, \lambda, E_1, E_2, \delta \rangle$. Conceptually, the first player represents the system / the protagonist, while the second player represents the error model / the antagonist.*
- *E_2 is partitioned into error and and non-error moves E_{error} and E_{noerr}, respectively. We require that only the 2nd player can issue error moves. Moreover, E_{noerr} must be non-empty.*
- *F is the set of failure states in Q with $r \notin F$.*

The antagonist can choose if she wants to respond on a move of the protagonist with an error move. We allow for different non-error moves to reflect 'normal' nondeterministic behavior, e.g., caused by abstraction. We allow for different error moves to reflect different errors that can occur in the same step.

We sometimes refer to transitions with error moves by the antagonist as error transitions *and to transitions with noerr moves by the antagonist as* controlled transitions.

For a party $A \subseteq \{1,2\}$, we refer with $\overline{A} = \{1,2\} \setminus A$ to the players not in the party, and by E_A to the moves made by the players in A, that is, $E_{\{1,2\}} = E_1 \times E_2$, $E_{\{1\}} = E_1$, etc.

The antagonist can use both error and non-error moves to influence the game. In a simple setting, the antagonist may only have the choice to insert error-moves, while there is only a single controlled transition. In this simple case, the protagonist can choose the successor state alone unless the antagonist plays an error transition. Specifically, a safety resilience game structure is simple *if E_2 contains only one error move. Considering simple safety resilience game structures leads to lower complexities, as it changes reductions from reachability in games (PTIME-complete [26]) to reachability in graphs (NL-complete [33]).* ■

Note that, in the game structure, only one system player and one error model player are allowed. This is purely for the simplicity of algorithm presentation. With proper reduction techniques, we can easily convert a game structure

with more than one system player and more than one error model player to the structure in Definition 5. The standard technique would be using the transition rules of the product automata of the system players for the protagonist while using the transition rules of the product automata of the error model players for the antagonist. In fact, we indeed use this reduction technique in our experiment for analyzing the resilience levels of multi-agent systems.

From now on, we assume that we are in the context of a given safety resilience game structure $\mathcal{G} = \langle \mathcal{K}, F \rangle$.

Definition 6 (Recovery segements): *We need to rigorously define* recovery segments. *A play prefix ρ is a* recovery segment *to safety region $S \subseteq Q \setminus F$ if it satisfies the following constraints.*

- $\rho(0) \in S$.
- *If $|\rho| = \infty$, then all states in $\rho[1, \infty)$ are in $Q \setminus (S \cup F)$. In this case, ρ is called a failed recovery segment.*
- *If $|\rho| \neq \infty$, then all states in $\rho[1, |\rho| - 2]$ are in $Q \setminus (S \cup F)$ and $last(\rho) = \rho(|\rho| - 1)$ is either in F or S. If $last(\rho) \in F$, ρ is also a failed recovery segment; otherwise, it is a successful one.*

We use $\mathrm{level}(\rho, S)$ *to denote the number of error moves between states in ρ with respect to the safety region S:* $\mathrm{level}(\rho, S) \stackrel{def}{=} \big|\{i \in [0, |\rho| - 1) \mid \rho_e(i) \models E_{error}\}\big|$. ■

As stated in the introduction, we propose a game-theoretic foundation for resilience analysis of software systems. With this perspective, the protagonist acts as a maximizer, who wants to maximize the resilience levels along all plays. For this, the protagonist fixes a strategy that describe what he is going to do on each play prefix. The antagonist acts as a minimizer, who wants to minimize the resilience level. She can resolve nondeterminism and inject errors in order to achieve this, and (although this plays no major role in this setting) she knows the strategy the protagonist has fixed and can use this knowledge in principle.

The goal of the protagonist is therefore the same as the goal of the system designer: to obtain a strategy that offers a maximal level of resilience in a safety game. However, in order to avoid degenerate behavior where the protagonist benefits from being in the recovery phase and from the antagonist therefore being allowed less errors in the current wave of errors she may inject, we have to strengthen his obligation to eventually recover to the safe states when the environment chooses not to inject further errors. This way, the protagonist has no incentive to cycle in the recovery region. Consequently, he can recover to the safe region within $|Q|$ moves after the antagonist has inserted the last error of the current wave, irrespective of whether the antagonist would be allowed to insert further errors in this wave. This is the key reason why memoryless optimal control exists for this error model, why it is reasonable to assume swift recovery, and, consequently, why it is a posteriori justified to leave the separation time between two waves implicit: the time to traverse $|Q|$ states suffices.

Besides obtaining this from intuition, we can also consider the tree of successful recoveries for any protagonist strategy that can endure k error moves by the antagonist. The tree of recoveries from up to k errors is finite according to the definition of successful recovery segments. Then for any subtree t in this tree of recoveries with a node v in t such that v is labeled with the same state as the root of t with no error on the path, we can always replace t with the subtree rooted at v. After the replacement, we have a tree of recoveries with no greater depth than the original one. Af-

ter repeating such replacements, this immediately provides a translation from such a strategy with unrestricted memory to one with memory of size k (the resilience level). The restriction to memoryless strategies follows from the construction we give in Section 6, which does not depend on the memory and still yields a strategy, which is memoryless. Thus, in this work, we should define the resilience level of software systems based on *memoryless* protagonist strategies.

Based on the argument above, the gain of the protagonist in a play can be defined as follows.

Definition 7 (Gain): *Given safety region $S \subseteq Q \setminus F$, the gain of a play ρ to S, in symbols $\mathit{gain}(\rho, S)$, denotes the maximal integer $k \in \mathbb{N}$ such that, for all recovery segments ρ_r to S in ρ, if $\mathrm{level}(\rho_r, S) \leq k$, then ρ_r is a successful recovery segment to S.* ■

The resilience level of a safety resilience game is defined as the maximum gain that the protagonist can guarantee in all plays with a memoryless strategy.

Definition 8 (Safety resilience game): *Such a game is zero-sum and defined on a safety resilience game structure $\mathcal{G} = \langle \mathcal{K}, F \rangle$ and a safety region $S \subseteq Q \setminus F$. The gain of $\mathcal{G}$ to S, in symbols $\mathit{gain}(\mathcal{G}, S)$, is defined as the maximum gain that the protagonist can manage with memoryless strategies. Rigorously,*

$$\mathit{gain}(\mathcal{G}, S) \stackrel{def}{=} \max_{\sigma \in \Sigma^{(0)}} \min_{\sigma' \in \Sigma} \mathit{gain}(\mathsf{play}(\mathsf{r}, \sigma, \sigma'), \mathsf{S})$$

Please be recall that $\mathsf{play}(\mathsf{r}, \sigma, \sigma')$ is the play from r according to strategies σ and σ' respectively of the two players. Moreover $\Sigma^{(0)}$ is the set of memoryless strategies.

*We say that the resilience level of $\mathcal{G}$ to S is $\mathit{gain}(\mathcal{G}, S)$. A strategy ω for the protagonist is optimal to S if $\min_{\sigma' \in \Sigma} \mathit{gain}(\mathsf{play}(\mathsf{r}, \omega, \sigma'), \mathsf{S}) = \max_{\sigma \in \Sigma^{(0)}} \min_{\sigma' \in \Sigma} \mathit{gain}(\mathsf{play}(\mathsf{r}, \sigma, \sigma'), \mathsf{S})$. When S is not given, we say that $\mathcal{G}$ is k-*resilient *if there exists a non-empty $S \subseteq Q \setminus F$ with $\mathit{gain}(\mathcal{G}, S) \geq k$.* ■

Remark. While the option of using memoryless strategies plays a minor role in the technical argument, it plays a paramount role in the usefulness of the resulting control strategy: choosing memoryless strategies implies that all recovery segments are short. In particular, all sub-paths (recovery segments) between two waves of dense errors injected by the antagonist are shorter—and usually significantly shorter—than the size of $\mathcal{G}$. In consequence, any time span long enough for traversing the recovery segment will lead to a full recovery. It is therefore sufficient for a temporal distance we have to assume between two waves of dense errors.

5 Alternating-time μ-calculus with events (AMCE)

We propose to solve our resilience game problems with an existing technology, i.e., model-checking of alternating-time μ-calculus (*AMC*) formulas. AMC is a propositional temporal logic with fixed point operators. For example, the following formula

$$\mu X.(\mathit{safe} \vee \langle 1 \rangle \bigcirc X) \qquad \text{(A)}$$

uses least fixed point operator μ to declare a fixed point variable X for a set of states. Subformula $\langle 1 \rangle \bigcirc \phi$ existentially quantifies over the protagonist strategies that can direct the plays to a successor state satisfying ϕ. Together, the formula specifies a set X of states that can inductively reach a safe state with the control of the protagonist. Specifically,

the formula says that a state is in X if either it is *safe* or the protagonist can direct to a successor state known to be in X. For our game structures, we only need strategy quantification of up to two players.

However, we need extend AMC with some simple syntax sugar. There are two extensions. The first is for Boolean combinations of path modalities in the scope of strategy quantification. For example, the following AMCE formula

$$\langle 1\rangle((\mathit{smoke} \Rightarrow \bigcirc \mathit{alarmOn}) \vee \bigcirc \mathit{windowClosed}) \qquad \text{(B)}$$

says that the protagonist can enforce either of the following two path properties with the same strategy.

- If there is smoke, then the alarm will be turned on in the next state.
- The window will always be closed in the next state.

Such a formula is not in ATL and AMC [3].

The second extension is for restricting transitions that may participate in the evaluation of path formulas. The restriction is via constraints on moves on transitions and can, in our extension to AMC, be specified with a move symbol set to the next-state modal operators. For example, the following AMCE formula

$$\langle 1\rangle((\bigcirc^{2:\mathit{error}} \mathit{alarmOn}) \wedge (\bigcirc^{\neg 2:\mathit{error}} \neg \mathit{alarmOn})) \qquad \text{(C)}$$

says that the protagonist can

- turn on the alarm when an error occurs; and
- keep the alarm silent when no error occurs.

Before we formally present AMCE, we need define expressions for constraints on moves of players in transitions. We adapt an idea from [44]. Specifically, a *move expression* η is of the following syntax.

$$\eta ::= a : e \mid \eta_1 \vee \eta_2 \mid \neg \eta_1$$

Here, a is a player index in $\{1, 2\}$ and e is a move symbol in $E_1 \cup E_2$. $\vee$ and $\neg$ are standard disjunction and negation. Typical shorthands of Boolean operations can also be defined out of $\vee$ and $\neg$. A total move vector can be expressed as $[e_1, e_2]$ where for all $a \in \{1, 2\}$, $e_a \in E_a$ is the move by player a specified in the vector. We say $[e_1, e_2]$ satisfies η, in symbols $[e_1, e_2] \models \eta$, if and only if the following constraints are satisfied.

- $[e_1, e_2] \models a : e$ if, and only if, e_a is e.
- $[e_1, e_2] \models \eta_1 \vee \eta_2$ if, and only if, $[e_1, e_2] \models \eta_1$ or $[e_1, e_2] \models \eta_2$.
- $[e_1, e_2] \models \neg \eta_1$ if, and only if, $[e_1, e_2] \not\models \eta_1$.

5.1 Syntax

A formula ϕ in AMCE has the following syntax.

$$\begin{array}{rcl} \phi & ::= & p \mid X \mid \phi_1 \vee \phi_2 \mid \neg \phi_1 \mid \mu X.\phi_1 \mid \langle A\rangle \psi \\ \psi & ::= & \mid \psi_1 \vee \psi_2 \mid \neg \psi_1 \mid \bigcirc^{\eta} \phi_1 \end{array}$$

Here, ϕ is a state formula, ψ is a path formula, p is an atomic proposition symbol in P (atomic proposition set, as in Definition 1), and X is a set variable for subsets of Q. The Boolean connectors are the common ones: $\vee$ for disjunction and $\neg$ for negation. Note that we allow for Boolean combinations of the next operators $\bigcirc$ under strategy quantification $\langle A\rangle$. This is one major difference of AMCE from AMC.

Formula $\mu X.\phi_1$ is the usual least fixed point operation to ϕ_1. According to the tradition in [3], we require that all free occurrences of X in ϕ_1 must occur within an even number of scopes of negations. This is because sentences with a negative occurrence, like $\mu X.\neg X$, have no natural semantics. A set variable X is *bound* in a formula ϕ if it is inside a declaration scope of X. If it is not bound, then

it is *free*. An AMCE sentence is an AMCE state formula without free set variables. In most cases, we are interested in specifications given as AMCE sentences.

The A in $\langle A \rangle$ is a finite set of player indices in $[1, 2]$. Conceptually, $\langle A \rangle \psi$ means that players in A can collaborate to make ψ true. For example, $\langle \{1, 2\} \rangle \bigcirc p$ means that players 1 and 2 can collaborate to make p true in the next state. We follow the notations in [3] and omit the parentheses in formulas like $\langle A \rangle \psi$. For example, $\langle \{2\} \rangle \bigcirc p$ and $\langle \{1, 2\} \rangle \bigcirc p$ will be abbreviated as $\langle 2 \rangle \bigcirc p$ and $\langle 1, 2 \rangle \bigcirc p$ respectively.

We allow event restrictions as superscripts in $\bigcirc^{\eta} \phi_1$ with a move expression η. The operator is important in supporting the evaluation of safety resilience levels with traditional model-checking technology. Note that since AMC [3] only allows for the next-state temporal modality, only the choice of moves to the next states of a strategy matters. Formula $\bigcirc^{\eta} \phi_1$ is thus evaluated at states with respect to move vectors satisfying constraint η. The formula is true of a move vector $[e_1, e_2]$ if and only if $[e_1, e_2] \models \eta$ implies the satisfaction of ϕ at state $\delta(q, e_1, e_2)$. Also $\bigcirc^{1:E_1} \phi_1$ can be written as $\bigcirc \phi_1$ in AMC [3] and the superscript to $\bigcirc$ can be omitted.

We also adopt shorthands in the below. The β refers to state or path formulas.

$$
\begin{array}{rcl}
\mathit{true} & \stackrel{\text{def}}{=} & p \vee \neg p \\
\mathit{false} & \stackrel{\text{def}}{=} & \neg p \wedge p \\
\beta_1 \wedge \beta_2 & \stackrel{\text{def}}{=} & \neg((\neg\beta_1) \vee (\neg\beta_2)) \\
\beta_1 \Rightarrow \beta_2 & \stackrel{\text{def}}{=} & (\neg\beta_1) \vee \beta_2 \\
\nu X.\phi & \stackrel{\text{def}}{=} & \neg\mu X.\neg\phi \\
[A]\psi & \stackrel{\text{def}}{=} & \neg\langle A \rangle \neg\psi
\end{array}
$$

5.2 Semantics

In the following, we adapt the presentation style of [3] to define the semantics of AMCE inductively over the structure of the subformulas. The value of a state formula at a state is determined by the interpretation of the set variables. Such an interpretation I maps set variables to subsets of Q. In comparison, the value of a path formula at a state is determined by both the interpretation of the set variables and the move vector chosen by the players. For convenience and conciseness of presentation, we extend the definition of interpretation of [3] also to record the chosen move vector by some players. Specifically, we use an auxiliary variable "move" for the present chosen move vector in the evaluation of path formulas. Given an interpretation I, $I(\mathsf{move})$ records the chosen move vector of all players in I. For example, $I(\mathsf{move}) = [\texttt{setAlarm}, \perp]$ means the chosen move vector that player 1 sets on an alarm while player 2 does nothing under interpretation I.

We need the following concept for collaborative choices of moves to the next states by some players. An *enforced move vector set* by $A \subseteq [1, 2]$ is a maximal set of move vectors that agree on the choices of moves by players with indices in A. Specifically, given an enforced move vector set C by A, we require that, for every $[e_1, e_2] \in C$, $[e'_1, e'_2] \in C$, and $a \in A$, $e_a = e'_a$. For convenience, we let Γ^A denote the set of all enforced move sets by A.

Following the semantics style of [3], we can extend I to be an interpretation of all state and path formulas. Intuitively, given a state or path formula β, $I(\beta)$ is the set of states that satisfy β according to the assumption on values of set variable values and auxiliary variable "move." More precisely, $I(\beta)$ is a subset of Q that satisfies the following

inductive rules.

- $I(p) = \{q \mid p \in \lambda(q)\}$.
- $I(\beta_1 \vee \beta_2) = I(\beta_1) \cup I(\beta_2)$.
- $I(\neg\beta_1) = Q - I(\beta_1)$.
- $I(\mu X.\phi_1)$ is the smallest set $Y \subseteq Q$ with $Y = I[X \mapsto Y](\phi_1)$, where $I[X \mapsto Y]$ is a new interpretation identical to I except that X is interpreted as Y.
- $I(\langle A\rangle\psi)$ is the set of states such that there is an enforced move vector set C by A such that, for all move vectors $\epsilon \in C$, $I[\mathsf{move} \mapsto \epsilon](\psi)$ holds:

$$I(\langle A\rangle\psi) = \bigcup_{C\in\Gamma^A} \bigcap_{\epsilon\in C} I[\mathsf{move} \mapsto \epsilon](\psi)$$

- Given $I(\mathsf{move}) = [\mathsf{e}_1, \mathsf{e}_2]$, if $[e_1, e_2] \models \eta$, then $I(\bigcirc^\eta\phi_1) = \{q \in Q \mid \delta(q, e_1, e_2) \in I(\phi_1)\}$; otherwise $I(\bigcirc^\eta\phi_1) = Q$.

A concurrent game structure is a model of an AMCE sentence ϕ, if its initial state r is in the interpretation of ϕ ($r \in I(\phi)$) for any interpretation I.

Note that, strictly speaking, AMCE does not add much to the expressiveness of AMC. In the literature, propositions have often been used to record events. Intuitively, we would need one atomic proposition for each event to mark that it has just occurred. This event marker would be true exactly at states right after the event happened. (One would possibly have to create multiple copies of states to reflect this.)

As discussed in [43], such a modeling technique leads to an unnecessary blow up of the state space, which could be exponential in the number of players in general concurrent games. By properly selecting the transitions with respect to operators like $\bigcirc^\eta$, such auxiliary propositions are not necessary when encoding the state space. Thus, AMCE can also be of interest to practitioners for the efficient analysis and verification of general concurrent games.

6 Resilience level checking algorithm

In Subsection 3.2, we have proposed the idea of the $\mathsf{sfrch}_k(\cdot)$ operator and proposed to use its greatest fixed point for the evaluation of k-resilience. In the following, we first establish some properties of k-safety and then use AMC model-checking technology to solve the safety resilience games.

6.1 High-level description of the algorithm

The following lemma shows the sufficiency of k-safety as a building block for solving safety resilience games.

Lemma 5 *For a safety resilience game $\mathcal{G}$, $\mathsf{sfrch}_k(\cdot)$ has a greatest fixed point.*

Proof : The lemma follows from the facts that the function sfrch_k is monotonic ($S \subseteq S'$ implies $\mathsf{sfrch}_k(S) \subseteq \mathsf{sfrch}_k(S')$ because a winning strategy for the protagonist for S is also a winning strategy for S' for all states in $\mathsf{sfrch}_k(S)$) and operates on a finite domain. ■

For the example in Figure 3, considering $S = \{1\}$ ($\{1\} = \mathsf{sfrch}_2(\{1, 2, 3\})$), the only state in S, state 1, is 2-resilient: it can recover with the recovery strategy to always go to the left.

The set of k-resilient states of $\mathcal{G}$, can be calculated as the greatest solution to $S = \mathsf{sfrch}_k(S)$ with $S \subseteq Q \setminus F$. Technically we can start the inductive calculation of the greatest fixed point from base case $S_0 = Q \setminus F$, and successively calculate $S_{i+1} = \mathsf{sfrch}_k(S_i)$, for each $i \geq 0$. The set of k-resilient states is then the limit S_∞. As soon as we have $S_{i+1} = S_i$, a fixed point is reached. We then have $S_i = S_\infty$ and can stop the inductive construction. Since S_0 is finite

and $S_{i+1} \subseteq S_i$ holds for all $i \geq 0$, we will eventually reach a j with $S_{j+1} = S_j = S_\infty$.

6.2 Realization with AMCE model-checking

We need formally define the interaction among strategies of players. We borrow the notation of function composition. Given two partial functions β_1 and β_2, we use $\beta_1 \circ \beta_2$ to represent their composition. Specifically, we have the following definition.

$$\beta_1 \circ \beta_2(a) = \begin{cases} \beta_1(a) & \text{if } \beta_2(a) \text{ is undefined.} \\ \beta_2(a) & \text{otherwise} \end{cases}$$

For our purpose, a partial strategy vector is a mapping from $\{1,2\}$ to Σ and can be undefined for some players in $\{1,2\}$. It is for a party $A \subseteq \{1,2\}$ if it is defined only for players in A and represents a collaborative strategy of the players with a defined strategy in A. It is total if it is defined for all players.

For convenience, we also define partial move vectors as mappings from $\{1,2\}$ to E. A partial move vector is for a party $A \subseteq \{1,2\}$ if it is defined only for players in A. It is total if it is defined for all players in $\{1,2\}$. Given two partial move vectors γ_1 and γ_2, we define $\gamma_1 \circ \gamma_2$ to represent the composition of the two vectors.

Given an S, we propose to construct $\mathsf{sfrch}_k(S)$ in an induction on k. We need the following preliminary concepts for the presentation.

Definition 9 ***(Traps)*** *For $A \subseteq \{1,2\}$, a* trap *for A is a subset $Q' \subseteq Q$ that party $\{1,2\} \setminus A$ has a strategy vector β to keep all plays from leaving Q'. Formally, we require that, for every $q \in Q'$ and partial move vector γ for A, there exists a partial move vector γ' for $\{1,2\} \setminus A$ such that $\delta(q, \gamma \circ \gamma'(1), \ldots, \gamma \circ \gamma'(m)) \in Q'$.* ■

6.2.1 Base case, $\mathsf{sfrch}_0(S)$

In the base case, $\mathsf{sfrch}_0(S)$ characterizes those states, from which the protagonist can direct the plays to S and stay there via a protagonist strategy when there is no error injected by the antagonist. Thus $\mathsf{sfrch}_0(S)$ is the greatest trap for the antagonist to S when no error happens and the greatest solution to the following equation.

$$X = \left\{ q \;\middle|\; \begin{array}{l} q \in X \cap S, e \in E_1, \\ \forall e' \in E_2 (e' \neq \mathit{noerr} \Rightarrow \delta(q,e,e') \in X) \end{array} \right\}.$$

In AMCE, we can alternatively define $\mathsf{sfrch}_0(S)$ as follows.

$$\mathsf{sfrch}_0(S) \stackrel{\text{def}}{=} \nu X.(S \wedge \langle 1 \rangle \bigcirc^{\neg 2:\mathit{error}} X).$$

This is the usual safety kernel of S, which consists of those states, from which any controlled transition is safe. It can be computed by the usual greatest fixed point construction.

Lemma 6 *$\mathsf{sfrch}_0(S)$ can be constructed, together with a suitable memoriless control strategy, in time linear to the size of $\mathcal{G}$.*

Proof : A state $q \in S$ can stay in $\mathsf{sfrch}_0(S)$ if there is a choice $e \in E_1$ such that for all $f \in E_2$, $\delta(q,e,f) \in \mathsf{sfrch}_0(S)$. Basically, we can use the typical approach of iterative elimination to calculate $\mathsf{sfrch}_0(S)$. That is, we first let $K_0 = Q - S$. Then we a sequence of mutually disjoint sets $K_1, K_2, \ldots, K_i, \ldots$ such that for all $i \geq 1$, states in K_{i+1} can be shown to be not in $\mathsf{sfrch}_0(S)$ by evidences of states in $K_i \cup \ldots \cup K_0$. Linear time can be achieved with careful book-keeping of the choices of moves at all states in S. We need a counter c_q for each $q \in S$ initialized to $|E_1|$ for the initial number of candidate choices of moves. Then for each $[q,e] \in S \times E_1$, we need a Boolean flag $b_{[q,e]}$ initialized to *true* to represent that $\{[e,f] \mid f \in E_2\}$ is still a valid choice of moves at q to satisfy $\mathsf{sfrch}_0(S)$. For each

Table 2. Algorithm for $\mathsf{sfrch}_0(S)$ by iterative elimination

$\mathsf{sfrch}_0(S)$

1: **for** $q \in S$ **do** $c_q = |E_1|$ **end for**
2: **for** $q \in S, e \in E_1$ **do** $b_{[q,e]} = \mathit{true}$ **end for**
3: Let $i = 0$ and $K_0 = Q - S$.
4: **while** $K_i \neq \emptyset$ **do**
5: Let $K_{i+1} = \emptyset$.
6: **for** $q \in K_i$ and $[q', e, f] \in L_q$ **do**
7: **if** $b_{[q',e]}$ is *true* **then**
8: Let $c_{q'} = c_{q'} - 1$.
9: **if** $c_{q'}$ is 0 **then** add q' to K_{i+1}. **end if**
10: **end if**
11: Set $b_{[q',e]}$ to *false*.
12: **end for**
13: Increment i by 1.
14: **end while**
15: **return** $S - (K_0 \cup \ldots \cup K_i)$.

state q, we also need to maintain a list of transition source states. That is, for each $\delta(q', e, f) = q$, we need record $[q', e, f]$ in list L_q. Then the iterative elimination proceeds as the algorithm in table 2. The algorithm is linear time since each transition $\delta(q, e, f)$ is checked exactly once. ■

6.2.2 Inductive cases, $\mathsf{sfrch}_k(S)$

Now we explain how to define the inductive cases of $\mathsf{sfrch}_k(S)$. The condition is for those states from which plays can be directed to S via a recovery segment in $Q \setminus (S \cup F)$ with k or less errors injected by the antagonist. An intermediate step for the construction of k-sfrch states is the construction of an attractor that controls, through controlled moves, the play prefixes to stay in a subset $L \subseteq Q \setminus F$ of non-failure states. As only controlled (non-error) moves are allowed, this is merely a backward reachability cone.

The *controlled limited attractor set* of a set X for a limited region $L \subseteq Q$, denoted $\mathsf{cone}_L(X)$ is the set from which there is a protagonist strategy to move to X without leaving L and errors injected by the antagonist. Technically, $\mathsf{cone}_L(X)$ is the least solution to equation:

$$Y = X \cup \left\{ q \;\middle|\; \begin{array}{l} q \in L, e \in E_1, \\ \forall e' \in E_2 \setminus \{\mathit{error}\}(\delta(q, e, e') \in Y) \end{array} \right\}.$$

The controlled limited attractor set $\mathsf{cone}_L(X)$ can be constructed using simple backward reachability for X of controlled transitions through states of L. In AMCE, this can be constructed as follows.

$$\mathsf{cone}_L(X) \stackrel{\text{def}}{=} \mu Y.(X \vee (L \wedge \langle 1 \rangle \bigcirc^{\neg 2:\mathit{error}} Y))$$

Note that the protagonist must use the same move irrespective of the move of the antagonist to both stay in L and approach X, provided that the antagonist does not inject an error.

The controlled limited attractor set $\mathsf{cone}_L(X)$ is used in the construction of $\mathsf{sfrch}_k(S)$. We further construct a descending chain $V_0 \supseteq V_1 \supseteq \ldots \supseteq V_{k-1}$ of limited attractors V_i. From V_i we have an attractor strategy towards S for the protagonist, which can tolerate up to i further errors. The respective V_i are attractors that avoid failure states. Moreover, from a state in V_i with $i > 1$, any error transition leads to V_{i-1}.

A state $q \in Q$ is *fragile* for a set $B \subseteq Q$ if, for all moves of the protagonist, at least one of its successors is outside of B. (The intuition is that this is an error move, and for simple safety resilience game structures, we can restrict the definition to failure states.) The set of fragile states for B is

$$\mathsf{frag}(B) \stackrel{\text{def}}{=} \{q \mid \forall e \in E_1 \exists e' \in E_2(\delta(q, e, e') \notin B)\}.$$

In AMCE, we have the following formulation of $\mathsf{frag}(B)$.

$$\mathsf{frag}(B) \stackrel{\text{def}}{=} [1] \bigcirc \neg B.$$

Technically, it is, however, easier to construct its dual

$$Q \setminus \mathsf{frag}(B) = \langle 1 \rangle \bigcirc B.$$

This dual can be constructed using a controlled backward reachability to B with any strategy of the protagonist.

The limited regions L_i of states allowed when approaching S also form a descending chain $L_0 \supseteq L_1 \supseteq \ldots \supseteq L_k$. Using these building blocks, we can compute the k-sfrch states as follows. The states in L_{i+1} are the non-failure states from which all error transitions lead to a state in V_i. The sets V_i contain the states from which there is a controlled path to S that progresses through L_i; all error transitions originating from any state of this path lead to V_{i-1}. V_0 is therefore just the set of states from which there is a controlled path to S.

From all states in V_{k-1}, the protagonist therefore has an optimal strategy in the recovery segment of the game described earlier: if the antagonist can play at most $k-1$ errors, then the protagonist can make sure that S is reached.

Starting with $L_0 \stackrel{\text{def}}{=} Q \setminus F$ that characterizes cones on the way to S without any errors, we define the V_k's and L_k's inductively by

$$L_k \stackrel{\text{def}}{=} L_0 \setminus \mathsf{frag}(Q \setminus \mathsf{cone}_{L_{k-1}}(S)),$$

In AMCE, this can be defined inductively as follows.

$$\begin{array}{lcl} L_0 & \stackrel{\text{def}}{=} & \neg F \\ L_k & \stackrel{\text{def}}{=} & L_0 \wedge \langle 1 \rangle \bigcirc \mathsf{cone}_{L_{k-1}}(S). \end{array}$$

Finally, we choose $\mathsf{sfrch}_k(S) \stackrel{\text{def}}{=} \mathsf{sfrch}_0(S \cap L_k)$. In AMCE, this can be expressed as follows.

$$\mathsf{sfrch}_k(S) \stackrel{\text{def}}{=} \mathsf{sfrch}_0(S \wedge L_k).$$

6.2.3 Algorithm for the set of k-resilient states

Finding a control strategy for k-sfrch control within $\mathsf{sfrch}_k(S)$ is simple: as long as we remain in $\mathsf{sfrch}_k(S) = \mathsf{sfrch}_0(S \cap L_k)$, we can choose any control move that does not leave $\mathsf{sfrch}_k(S)$. Once $\mathsf{sfrch}_k(S)$ is left through an error transition to $V_{k-1}, V_{k-2}, \ldots$, we determine the maximal i for which it holds that we are in V_i and follow the attractor strategy of $\mathsf{cone}_{L_i}(S)$ towards S.

In summary, we present our algorithms for the set of k-resilient states in Table 3. In fact, we have presented two algorithms. The first constructs $\mathsf{sfrch}_k(S)$, which can be used for checking whether the safety region S provided by the users is indeed a good one. The way to do it is to simply check whether S is a solution to $\mathsf{sfrch}_k(x) = x$.

Then our second algorithm calculates $\mathsf{res}_k(\mathcal{G})$ as the greatest fixed point S of $\mathsf{sfrch}_k(.)$ as the recommendation for the safety region:

$$\mathsf{res}_k(\mathcal{G}) = \bigcup \{S \subseteq Q \mid S = \mathsf{sfrch}_k(S) \text{ and } S \cup F = \emptyset\}.$$

In this way, the users do not have to calculate and provide the safety region, which would be error prone. According to the argument and lemmas from above, we get the following theorem.

Theorem 7 *$\mathcal{G}$ is k-resilient if, and only if, $r \in \mathsf{res}_k(\mathcal{G})$.* ■

6.3 Complexity

A rough complexity of our resilience level checking algorithm straightforwardly follows the complexity of AMC model-checking. Specifically, the following lemma explains the maximal resilience level that we need consider. For convenience, let $k_{\max}$ be the maximal resilience level of $\mathcal{G}$.

Lemma 8 *$k_{\max}$ is either infinite or no greater than $|Q \setminus F|$.*

$$
\begin{array}{rcl}
L_0 & \stackrel{\text{def}}{=} & \neg F \\
L_k & \stackrel{\text{def}}{=} & \neg F \wedge \langle 1 \rangle \bigcirc \mu y.S \vee (L_{k-1} \wedge \langle 1 \rangle \bigcirc^{error} y \wedge \bigcirc L_{k-1}) \\
\mathsf{sfrch}_0(S) & \stackrel{\text{def}}{=} & \nu x.(S \wedge \langle 1 \rangle \bigcirc^{error} x) \\
\mathsf{sfrch}_k(S) & \stackrel{\text{def}}{=} & \mathsf{sfrch}_0(S \wedge L_k) \\
\mathsf{res}_k(\mathcal{G}) & \stackrel{\text{def}}{=} & \nu S.((Q \smallsetminus F) \wedge \mathsf{sfrch}_k(S)) : \text{the set of } k\text{-resilient states}
\end{array}
$$

Table 3. Algorithm for k-resilient states

Proof : We assume that $k_{\max}$ is greater than $|Q \smallsetminus F|$ but not infinite. This means that there exists a failed recovery segment ρ with $k+1$ errors injected by the antagonist. Since the protagonist can only use memoryless strategies, there must be two position indices $i < j < |\rho| - 1$ with $\rho(i) = \rho(j)$ in the recovery segment such that at $\rho(i)$ and $\rho(j)$, the protagonist makes the same move while the antagonist makes different moves. This implies the existence of a shorter failed recovery segment $\rho[0,i]\rho[j+1,|\rho|-1]$. By repeating the above argument, we can eventually identify a failed recovery segment of length $\leq |Q \smallsetminus F|$ that contradicts the assumption and establishes the lemma. ■

With Lemma 8, we can use the complexity of AMC model-checking problem [3] to straightforwardly establish the $O(k_{\max}|E|)^2 = O(|Q \smallsetminus F| \cdot |E|)^2$ complexity of $\mathsf{res}_k(\mathcal{G})$ when k is $k_{\max}$. In the following, we present a more detailed analysis of the complexity of our resilience level checking algorithm. All individual steps in the construction (intersection, difference, predecessor, and attractor) are linear in the size of the safety resilience game, and there are $O(k)$ of these operations in the construction. This provides a bi-linear (linear in k and $|\mathcal{G}|$) algorithm for the construction of sfrch_k and a strategy for the protagonist.

Lemma 9 *A memoryless control strategy for the states in $\mathsf{sfrch}_k(S)$ can be constructed in time linear in both k and the size $|\mathcal{G}|$ of the safety resilience game $\mathcal{G}$.* ■

The construction of $\mathsf{res}_k(\mathcal{G})$ uses the repeated execution of $(Q \smallsetminus F) \wedge \mathsf{sfrch}_k(\cdot)$. The execution of $\mathsf{sfrch}_k(\cdot)$ needs to be repeated at most $|Q \smallsetminus F|$ times until a fixed point is reached, and each execution requires at most $O(k \cdot |\mathcal{G}|)$ steps by Lemma 9.

For the control strategy of the protagonist, we can simply use the control strategy from $\mathsf{sfrch}_k(S_\infty)$ from the fixed point S_∞. This control strategy is memoryless (cf. Lemma 9).

Lemma 10 *$\mathsf{res}_k(\mathcal{G})$ and a memoryless k-resilient control strategy for $\mathsf{res}_k(\mathcal{G})$ can be constructed in $O(k \cdot |Q \smallsetminus F| \cdot |\mathcal{G}|)$ time.* ■

Finding the resilience level $k_{\max}$ for the initial state r requires at most $O(\log k_{\max})$ many constructions of $\mathsf{res}_i(\mathcal{G})$. We start with $i = 1$, double the parameter until $k_{\max}$ is exceeded, and then use logarithmic search to find $k_{\max}$.

Corollary 11 *For the initial state r, we can determine the resilience level $k_{\max} = \max\{i \in \mathbb{N} \mid r \in \mathsf{res}_i(Q \smallsetminus F)\}$ of r, $\mathsf{res}_{k_{\max}}(Q \smallsetminus F)$, and a memoryless $k_{\max}$-resilient control strategy for $\mathsf{res}_{k_{\max}}(Q \smallsetminus F)$ in $O(|Q \smallsetminus F| \cdot |\mathcal{G}| \cdot k_{\max} \log k_{\max})$ time.* ■

Simple safety resilience game structures. For simple safety resilience game structures, checing if a state is in $\mathsf{sfrch}_0(S)$ is NL-complete.

Lemma 12 *Testing if a state is in* $\mathsf{sfrch}_0(S)$ *is NL-complete.*

Proof: NL completeness can be shown by reduction to and from the repeated ST-reachability [33] (the question whether there is a path from a state S to a state T and from T to itself in a directed graph). ∎

Likewise, the controlled limited attractor set $\mathsf{cone}_L(S)$ can be constructed using simple backwards reachability for G of controlled transition through states of L. For $A = \mathsf{cone}_L(S)$, determining whether a state is in A is NL-complete (see [33]).

The complexity of determining whether or not a state q is in $\mathsf{sfrch}_k(S)$ thus depends on whether or not we consider k to be a fixed parameter. Considering k to be bounded (or fixed) is natural in our context, because k is bounded by the redundancy.

Lemma 13 *For a fixed parameter k, testing if a state s of a simple safety resilience game structures is in* $\mathsf{sfrch}_k(S)$ *is NL-complete.*

Proof: Testing if a state is in L_0 is in NL. By an inductive argument, we can show that

- provided that testing if a state is in L_i is in NL, we can test if a state is in $A_i = \mathsf{cone}_{L_i}(S)$ by using the nondeterministic power to guess a path towards S, while verifying that we are in L_i in every state we pass before S is reached; and
- if we can check if a state is in A_i in NL, then we can check if it is in $Q \setminus A_i$ [27], in $\mathsf{frag}(Q \setminus A_i)$ (with one nondeterministic transition), and in $L_{i+1} = L_0 \setminus \mathsf{frag}(S \setminus A_i)$ [27] in NL.

Testing that a state is in $S \cap L_k$ is therefore in NL and testing if it is in $\mathsf{sfrch}_0(S \cap L_k)$ reduces to guessing a state t in $\mathsf{sfrch}_k(G)$ and an ST path (a path from s to t followed by a loop from t to t), verifying for all states on the path that they are in $S \cap L_k$.

For hardness, note that the last step of the construction alone is NL-complete (Lemma 6). ∎

If k is considered an input, then reachability in AND-OR graphs can easily be encoded in LOGSPACE: It suffices to use the nodes of an AND-OR graph as the states, the outgoing edges of OR nodes as the result of the choice of the protagonist only (while the move of the antagonist has no influence on the outcome, no matter whether or not she induces an error), and to model the AND nodes as a state, where the no-error move of the antagonist will lead in cycling in the state, while the antagonist can choose the successor from the graph when inducing an error. Choosing k to be the number of nodes of the AND-OR graph and F to be the target nodes of the AND-OR graph, the target nodes of the AND-OR graph are not reachable from a state s iff $s \in \mathsf{sfrch}_k(Q \setminus F)$.

Given that reachability in AND-OR graphs is PTIME-complete [26], this provides:

Lemma 14 *If k is considered an input parameter, then testing if a state s of a simple safety resilience game structures is in* $\mathsf{sfrch}_k(S)$ *is PTIME-complete.* ∎

The complexity of $\mathsf{res}_k(S)$ is (almost) independent of the parameter k:

Theorem 15 *The problem of checking whether or not a state s is k-resilient for a set S is PTIME-complete for all $k > 0$ and NL-complete for $k = 0$.*

Proof: We have shown inclusion in PTIME in Lemma 10. For hardness in the $k > 0$ case, we can use the same reduction from the reachability problem in AND-OR graphs as for $\mathsf{sfrch}_k(S)$.

For $k = 0$, $\mathsf{sfrch}_0(G) = \mathsf{sfrch}_0\big(\mathsf{sfrch}_0(G)\big)$ implies $\mathsf{res}_0(G) = \mathsf{sfrch}_0(G)$. The problem of checking if a state is in $\mathsf{res}_0(G)$ is therefore NL-complete by Lemma 12. ∎

Hardness for general safety resilience game structures. For general resilience game structures, we can again use a LOGSPACE reduction from the reachability in AND-OR graphs: We again use the nodes of an AND-OR graph as the states, and the outgoing edges of OR nodes are selected based on the choice of protagonist only. For the AND nodes, we leave the choice to the antagonist only, whithout the need to invoke an error. (That is, errors play no role in this reduction. The antagonist may be allowed to insert one, but she can always obtain the same transition without doing so.)

Marking F as the target nodes, we get $\mathsf{res}_k(Q \setminus F) = \mathsf{sfrch}_l(Q \setminus F)$ for all non-negative integers k, l, and $s \in \mathsf{sfrch}_0(Q \setminus F)$ iff the target nodes of the AND-OR graph are not reachable from s. With Lemmas 9 and 10, we get the following theorem.

Theorem 16 *For all $k \geq 0$, the problems of checking whether or not a state s is in $\mathsf{res}_k(Q \setminus F)$ and $\mathsf{res}_k(Q \setminus F)$, respectively, are PTIME-complete for general safety resilience game structures.*

7 Tool implementation and experimental results

In the following, we report our implementation and experiment with our constructions. Our implementation is based on symbolic on-the-fly model-checking techniques and built on the simulation/model-checking library of REDLIB in `https://github.com/yyergg/Resil` for fast implementation. Our implementation and benchmarks can also be found in the same page.

We adopt *CEFSM* (*communicating extended finite-state machine*) [7] as a convenient language for the description of abstract models of our concurrent game structures. A CEFSM consists of several finite-state machines extended with shared variables for the modeling of shared memory and with synchronizations for the modeling of message-passing in distributed systems. This is justifiable since the fault-tolerant algorithms may themselves be subject to restrictions in concurrent or distributed computation. Indeed, we found CEFSM very expressive in modeling the benchmarks from the literature [12,38].

The translation from our CEFSMs to state transition systems, such as finite Kripke structures, is standard in the literature. All state spaces, conditions, preconditions, post-conditions, fixed points, etc., are represented as logic formulas. The logic formulas are then implemented with multi-value decision diagrams (MDD) [32].

We then took advantage of the support of REDLIB for writing down template automatas for constructing complex models. We specified a template automata with REDLIB to describe the moves of the players. Conceptually, the player automatas are constructed as an instance of the template automata. Then the whole game structure is constructed as the

product of all player automatas. Finally, we use the API of REDLIB to do on-th-fly construction of the game structure which can be advantageous since unreachable states will never be generated.

7.1 Benchmarks

We use the following five parameterized benchmarks to check the performance of our techniques. Each benchmark has parameters for the number of participating modules in the model. Such parameterized models come in handy for the evaluation of the scalability of our techniques with respect to concurrency and model sizes.

1. We use the example of a fault-tolerant computer architecture (Example 1) as our first benchmark. An important feature of this benchmark is that there is an assumed mechanism for detecting errors of the modules. Once an error is detected, a processor can be assigned to recover the module, albeit to the cost of a reduced redundancy in the executions.

2. Voting is a common technique for fault tolerance through replication when there is no mechanism to detect errors of the modules [36]. In its simplest form, a system can guarantee correctness, provided less than half of its modules are faulty. This benchmark implements this simple voting mechanism. Every time a voting is requested, the modules submit their ballots individually. Then we check how many module failures the system can endure and recover.

3. This is a simplified version of the previous voting benchmark, where we assume that there is a blackboard for the client to check the voting result.

4. *Practical Byzantine fault-tolerance* (*PBFT*) algorithm: We use an abstract model of the famous algorithm by Castro and Liskov [12]. It does not assume the availability of an error-detection mechanism but uses voting techniques to guarantee the correctness of computations when less than one third of the voters are faulty. This algorithm has impact on the design of many protocols [2, 14, 15, 23, 29] and is used in Bitcoin [1], a peer-to-peer digital currency system.

5. *Fault-tolerant clock synchronization algorithm*: Clock synchronization is a central issue in distributed computing. In [38], Ramanathan, Shin, and Butler presented several fault-tolerance clock synchronization algorithms in the presence of Byzantine faults with high probability. We use a nondeterministic abstract model of the convergence averaging algorithm from their paper. The algorithm is proven correct when no more than one third of the local clocks can drift to eight time units from the median of all clock readings.

7.2 Modeling of the fault-tolerant systems

Appropriate modeling of the benchmarks is always important for the efficient verification of real-world target systems. Many unnecessary details can burden the verification algorithm and blow up the computation, while sketchy models can then give too many false alarms and miss correct benchmarks. We have found that there is an interesting aspect in the modeling of the above benchmarks. Replication and voting are commonly adopted techniques for achieving fault-tolerance and resilience. Such fault-tolerant algorithms usually consist of several identical modules that

use the same behavior templates. This observation implies that the identity of individual modules can be unimportant for some benchmarks. For such benchmarks, we can use counter abstraction [20, 31] in their models. Specifically, with counter abstraction, we can model all system players with one player that keeps a counter $c(l)$ for each control location l in the template automatas. Then at a state of the whole game graph, $c(l)$ records the number of system players at location l. With this technique, a system with $m-1$ system player and one error model player is then reduced to two players: one counter-abstraction player for all the system players and one remaining error model player. If a system player enters a location l in a global transition, then in the model, $c(l)$ is incremented by one in the abstract global transition. If a system player leaves l in the global transition, then $c(l)$ is decremented by one in the abstract global transition. But the succession of location movements of a particular player is omitted from the abstraction.

We found that we can use counter abstraction to prove the correctness of benchmarks 1, 2, and 3. In contrast, the PBFT and the clock synchronization algorithms use counters for each module to model the responses received from its peer modules. As a result, we decided not to use counter abstraction to model these two algorithms in this work.

In the following, we explain how to apply our techniques to analyze the resilience levels of the avionic systems in Example 1. The application is achieved in three steps. We first model the system under analysis either as a plain CEFSM or with counter abstraction (if our analysis tool cannot handle the complexity of the plain CEFSM). We then build the product automaton of the CEFSM as the resilience game structure except for the move vectors. Finally, we convert the labels on the transitions of the product automaton to move vectors of the two players. Note that the moves may not correspond to the transition labels of the CEFSM.

Step 1: the construction of the CEFSM

We first present the CEFSM model template of Example 1 in Figure 4. The CEFSM model has n processors and m memory modules. Figures 4(a) and (b) are for the abstraction of processors and memory copies, respectively. The ovals represent local states of a processor or a memory module, while the arrows represent transitions. The transitions of a CEFSM are labeled with '*error*', 'C' (for Control), or 'R' (for recovery).

We also use synchronizers to bind process transitions. For example, when a memory module moves into a faulty state, an idle processor may issue an `fd` (error-detected) event and try to repair the module by copying memory contents from normal memory modules. Such error-detection is usually achieved with standard hardware. Note that the benchmarks are models that reflect the recovery mechanism, abstracting away the details of the original systems. A central issue in the design of this recovery mechanism is then the resilience level of the controlled systems. We need three synchronizers: *fd* for error detection by a processor, *rs* for recovery success, and *rf* for recovery failure. The three synchronizers are used to bind a transition from a processor and another from a memory module into a synchronized transition. For example, a processor at state *pidle* and a memory module at state *mfaulty* may simultaneously enter their *pcopy* and *mcopy* states respectively through synchronizers !*fd* (for sending the synchronizer) and ?*fd* (for receiving). We also conveniently use a variable q in this

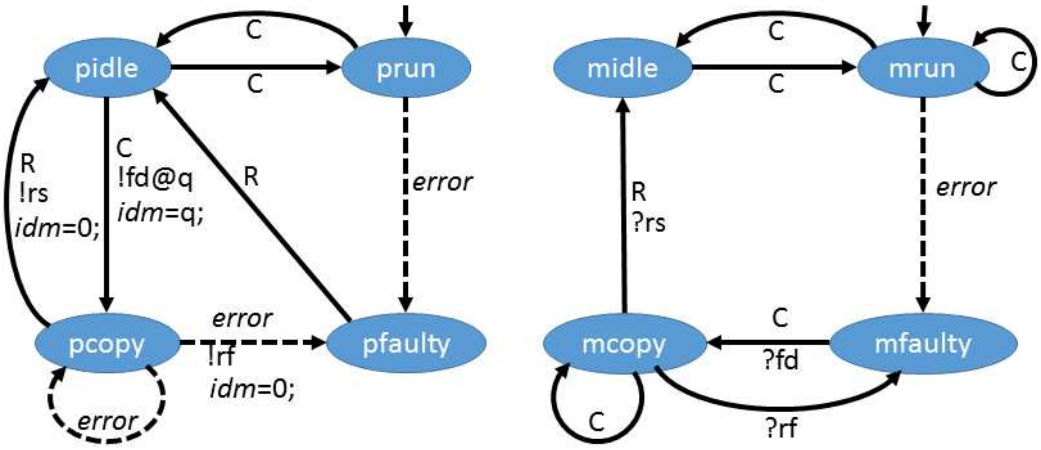

Figure 4. CEFSM templates of n processors and m memory copies

synchronized transition to capture the identifier of the memory module receiving the synchronizer. A transition without synchronization labels is considered a trivial synchronized transition. The transition system of the CEFSM operates with interleaving semantics at the abstraction level of the synchronized transitions.

For counter abstraction, we need four global variables *crp*, *cfp*, *crm*, and *cfm* respectively to keep track of the numbers of running processors, faulty processors, running memory modules, and faulty memory modules. We also need a local variable *idm* for each processor to record the faulty memory module identifier that the processor is responsible for recovery. We label the controllable, error, and recovery transitions respectively with 'C', '*error*', and 'R'. We also label each transition with synchronizers and actions. At any moment, the processors and the memory modules may enter their running states, execute a task, and generate the outcome. A processor starts its execution from state *prun* while a memory module starts from state *mrun*.

Step 2: building the product automata

The product automata is a Kripke structure whose states are of is a vector $[p_1, \ldots, p_n, i_1, \ldots, i_n, s_1, \ldots, s_m]$ of $2n+m$ elements. For all k, p_k and i_k respectively represent the current location and the current *idm* value of processor k while s_k represents the current location of memory module k. Then interleaving semantics that each time only a global transition (a single local process transition without synchronizers or two local process transition bound by a synchronizer) is executed is adopted to determine the transition relation from one state to another. Such techniques are standard in model construction. REDLIB can help in this regard by constructing the Kripke structure in an on-the-fly style to avoid the construction of those states not reachable from the initial state.

Step 3: the labeling of the move vectors

After the second step, we have the game structure ready except for the move vectors on the transitions. We use $E_1 = \{C, R, nop\}$, where *nop* represents "no operation,"

and $E_2 = \{noerr, error\}$. Then we use the following three rules to label move vectors.

- Every global transition with one component local process transition labeled with *error* is labed with move vector $[nop, error\}$.
- Every global transition with a component local process transition labeled with R is labeled with move vector $[R, noerr]$.
- All other global transitions are labeled with move vector $[C, noerr]$.

Counter abstraction of the example

We also use the CEFSM in figure 4 to explain counter abstraction. We need eight counter variables: *pr*, *pi*, *pc*, *pf*, *mr*, *mi*, *mc*, and *mf* to respectively record the number of processes in location prun, pidle, pcopy, pfaulty, mrun, midle, mcopy, and mfaulty in a state. Then the counter abstraction of the CEFSM is in Figure 5. The initial state are specified with constraint: $pr = n \wedge pi = 0 \wedge pc = 0 \wedge pf = 0 \wedge mr = m \wedge mi = 0 \wedge mc = 0 \wedge mf = 0$ on the counters. The state in the product automata must satisfy the following constraints: $pr + pi + pc + pf = n \wedge mr + mi + mc + mf = m$. As can be seen, we do not care which processor is in the idle mode, in the running mode, etc., in this abstraction. Similarly, we do not care which memory module is in the idle mode, in the running mode, and etc. The local state transition only keeps tracks of the number of processors in each mode and the number of memory modules in each mode. We also do not care which processor is in charge of the recovery of which memory module. Such an abstraction can be done automatically.

The labeling of the move vectors on the transitions in the Kripke structure (product automaton) follows the same rules for the product automaton from the CEFSM in Figure 4.

Analysis of the game structure

The majority outcome of the processors and memory copies is used as the outcome of the system. A processor may enter the faulty state. A memory module may also enter the faulty state. Processors may control to recover themselves or a faulty memory module by copying the contents of a functioning memory module to the faulty one. At any moment, we want to make sure that we can always recover to a global condition with the following two restrictions.

- There are at least two more processors in the running mode than the processors in the faulty mode.
- There are at least two more memory copies in the running mode than memory copies in the faulty mode.

Together, the failure condition is $crp - cfp < 2 \vee crm - cfm < 2$. That is, all states in the transition system satisfying $crp - cfp < 2 \vee crm - cfm < 2$ are in set F.

Tool implementation and the benchmarks used in the experiment can all be found in our Sourceforge REDLIB project at `https://github.com/yyergg/Resil`.

7.3 Performance data

We report the performance data in Table 4 for the resilience algorithms described in Section 7.1 against the parameterized benchmarks in the above with various parameters. The second column shows the concurrency sizes. The third column shows the values of k for the rows. The fourth and fifth columns show the sizes of the concurrent game

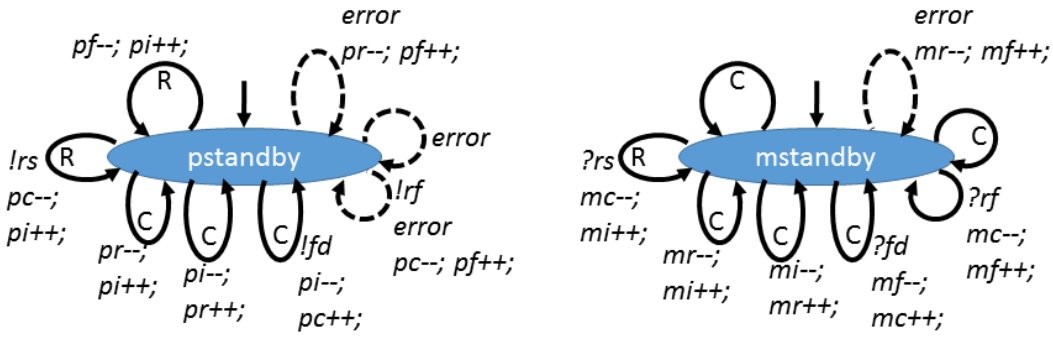

Figure 5. Counter abstraction of the CEFSM templates of n processors and m memory copies

structures. The sixth and seventh columns show the time and spaces used to calculate $\mathsf{sfrch}_k()$. Similarly, the eighth and ninth columns show the time and spaces for calculating the $\mathsf{res}_k()$.

The benchmark in Figure 4 does not have nodes in $\mathsf{sfrch}_2(G)$ and $\mathsf{res}_2(G)$. So we changed the benchmark to see how we check our implementation with $k > 1$. The change is that the recovery transition from state *pcopy* to *pidle* of processors are relabeled as controllable. This change significantly limits the ability of the system errors to derail the system. For the avionics system, the resilience level k is set to one less than half the number of processors. For the voting and simple voting benchmarks, the value of k is set to one less than half the number of replicas (voters). For the PBFT and clock synchronization algorithm, we choose k to be one less than one third of the number of replicas.

The performance data has been collected with a Virtual Machine (VM) running opensuse 11.4 x86 on Intel i7 2600k 3.8GHz CPU with 4 cores and 8G memory. The VM only uses one core and 4G memory.

The time and space used to calculate resilience is a little bit more than that to check for sfrch. The reason is that sfrch_k is a pre-requisite for calculating res_k. In our experiment, sfrch_k is usually very close to res_k and does not require much extra time in calculating res_k out of sfrch_k.

The experiments show that our techniques scale to realistic levels of redundancy. For fault-tolerant hardware, usually the numbers of replicas are small, for example, less than 10 replicas. Thus our techniques seem very promising for the verification and synthesis of hardware fault-tolerance.

On the other hand, nowadays, software fault-tolerance through networked computers can create huge numbers of replicas. Our experiment shows that counter abstraction can be a useful techniques for the modeling and verification of software resilience. Specifically, for the avionics benchmark, we can verify models of much higher concurrency and complexity with counter abstraction than without.

8 Related work

We have applied game-based technqiues [13, 34, 37] for synthesizing a control mechanism with maximal resilience to software errors. The synthesis of control strategies is essential in solving games with temporal and ω-regular objectives. For these more complex objective, synthesis goes back to Church's solvability problem [13] and inspired Rabin's work on finite automata over infinite structures [37]

Table 4. Performance data for resilience calculation **s: seconds; M: megabytes**

benchmarks	concurrency	k	game sizes		sfrch_k		res_k	
			#nodes	#edges	time	memory	time	memory
avionics	2 processors & 2 memory modules	2	118	750	0.62s	114M	0.85s	116M
	2 processors & 3 memory modules	2	414	3252	0.94s	139M	1.10s	153M
	3 processors & 3 memory modules	3	1540	15090	4.67s	225M	8.38s	267M
	3 processors & 4 memory mdules	3	5601	63889	42.86s	815M	155s	846M
avionics (counter abstraction)	6 processors & 6 memory modules	2	1372	6594	2.89s	129M	3.54s	516M
	7 processors & 7 memory modules	3	2304	11396	10.7s	216M	23.4s	808M
	8 processors & 8 memory modules	3	3645	18432	43.8s	1009M	135s	2430M
voting	1 client & 20 replicas	9	9922	23551	7.01s	260M	36.7s	297M
	1 client & 26 replicas	12	20776	49882	19.9s	474M	79.6s	611M
simple voting	1 client & 150 replicas	74	458	1056	0.71s	159M	31.7s	219M
	1 client & 200 replicas	99	608	1406	1.06s	161M	162s	337M
	1 client & 250 replicas	124	758	1756	1.36s	163M	307s	499M
PBFT	1 client & 6 replicas	2	577	897	0.34s	72M	1.05s	193M
	1 client & 9 replicas	4	2817	4609	13.3s	564M	58.5s	1657M
clock sync	1 client & 15 servers	7	16384	229376	45.1s	3075M	62.4s	3264M
	1 client & 17 severs	8	65536	1070421	870s	14725M	915s	15433M

and Büchi and Landweber's works on finite games of infinite duration [10,11]. A righ body of literature on synthesis has since been developed [6,18,22,30,35,39,40].

Traditionally, fault tolerance refers to various basic fault models [6], such as a limited number of errors [28]. These traditional fault models are subsumed by more general synthesis or control objectives [5,6,42]; as simple objectives with practical relevance, they have triggered the development of specialized tools [18,22].

Dijkstra's self-stabilization criterion [4,16] suggests to build systems that eventually recover to a 'good state', from where the program commences normally. Instead of *constructing* a system to satisfy such a goal, one might want to apply control theory to *restrict* the execution of an existing system to achieve an additional goal. Our control objective is a recovery mechanism for up to k errors. After recovery, the system has to tolerate up to k errors again, and so forth. In this work, we suggest a mechanism to synthesize a recovery mechanism for a given fault model and recovery primitives.

In [17], an interesting notion of robustness based on Hamming and Lewenstein distance related to the number of past states is defined. It establishes a connection between these distances with a notion of synchronization that characterizes the ability of the system to reset for combinatorial systems. In [9], 'ratio games' are discussed, where the objective is to minimize the ratio between failures induced by the environment and system errors caused by them.

Besides using our simple game model that neither refer directly to time, nor to probabilities, one can also consider models that make these aspects explicit. Their analysis is far more complex (with [21] offering the best complexity bounds), and so are the resulting strategies. If we, for example, return to the example of airplanes with an operation time of 20 hours referred to in Table 1, then an optimal timed model would take the remaining operation time into account. When the remaining time is two minutes, the balance between being resilient against waves of two errors

and being resilient against 5 errors looks very different, and the optimal control would change over time rather than being static. Another implication of more complex models would be that the error model would have to be more detailed. Even if one assumes that a simple concept like safe states persists, it depends on the fineties of such a model if a two step path back to it where an error after step one leads to system failure is preferable over a much longer path, say through 10,000 intermediate states, where one error can be tolerated during recovery.

We believe that the independence from such details is an advantage of our technique, partly because it is simpler and cheaper, and partly because the further advantages one can obtain from more detailed error models rely heavily on very knowledge of (or, realistically, on very detailde assumptions on) how errors are distributed.

In [8, 19] the resilience model we have introduced [25] has been applied for synthesising robust control in an assume-guaranee setting to produce robustness against occasional noncompliance of the environment with the assumptions of its behavior.

9 Conclusion

We have introduced an approach for the development of a control of safety critical systems that maximizes the number of *dense* errors the system can tolerate. Our techniques are inspired by the problem of controlling systems with redundancy: in order to deflect the effect of individual errors, safety critical systems are often equipped with multiple copies of various components. If one or more components fail, such systems can still work properly as long as the correct behavior can be identified.

This has inspired the two-phase formulation of the safety resilience problems in this article. In the first phase, we identify a k-resilient region, while we develop a control strategy for recovery in the second phase. After an error, the controller can recover to the k-resilient region without encountering a system failure, unless the error is part of a group of more than k errors that happen in close succession. Such a recovering strategy is memoryless. Being memoryless on a small abstraction in particular implies that the recovery is fast.

The system can, once recovered, tolerate and recover from k further dense errors, and so forth. Consequently, our control strategy allows for recovery from an arbitrary number of errors, provided that the number of dense errors is restricted. This is the best guarantee we can hope for: our technique guarantees to find the optimal parameter k. This parameter is bound to be small (smaller than the number of redundant components). Optimizing it is computationally inexpensive, but provides strong guarantees: the likelihood of having more than k errors appear in short succession after an error occurred are, for independent errors, exponential in k. As errors are few and far between, each level of resilience gained reduces the likelihood of system-level failures significantly.

References

[1] Bitcoin website. "`http://bitcoin.org/`".

[2] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM Symposium on Operating Systems Principles*, 2005.

[3] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5):672–713, September 2002.

[4] A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering (TSE)*, 19(11):1015–1027, 1993.

[5] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, volume LNCS 999. Springer-Verlag, 1995.

[6] P. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems*, 26(1):125–185, 2004.

[7] F. Belina and D. Hogrefe. The ccitt-specification and description language sdl. *Computer Networks and ISDN Systems*, 16(4):311–341, March 1989.

[8] R. Bloem, R. Ehlers, S. Jacobs, and R. Könighofer. How to handle assumptions in synthesis. In K. Chatterjee, R. Ehlers, and S. Jha, editors, *Proceedings of the 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014*, volume 157 of *EPTCS*, pages 34–50, 2014.

[9] R. Bloem, K. Greimel, and B. J. T.A. Henzinger. Synthesizing robust systems. In *FMCAD*, pages 85–92, 2009.

[10] J. Büchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11, 1962.

[11] J. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138(4):295–311, 1969.

[12] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[13] A. Church. Logic, arithmetic and automata. In *International Congress of Mathematicians*, pages 23–35, August 1962 (Stockholm 1963).

[14] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *USENIX Symposium on Networked Systems Design and Implementation*, April 22-24, 2009.

[15] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *USENIX OSDI Symposium*, 2006.

[16] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[17] L. Doyen, T. Henzinger, A. Legay, and D. Nickovic. Robustness of sequential circuits. In *ACSD*, pages 77–84, 2010.

[18] A. Ebnenasir, S. Kulkarni, and A. Arora. Ftsyn: a framework for automatic synthesis of fault-tolerance. *STTT*, 10(5):455–471, 2008.

[19] R. Ehlers and U. Topcu. Resilience to intermittent assumption violations in reactive synthesis. In M. Fränzle and J. Lygeros, editors, *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014*, pages 203–212. ACM, 2014.

[20] E. Emerson and R. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model-checking. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 142–156, 1999.

[21] J. Fearnley, M. Rabe, S. Schewe, and L. Zhang. Efficient approximation of optimal control for continuous-time markov games. In *Proc. 31st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 399–410, 2011.

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

[23] R. Guerraoui, N. Knežević, M. Vukolić, and V. Quéma. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, 2010.

[24] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[25] C.-H. Huang, D. Peled, S. Schewe, and F. Wang. Rapid recovery for systems with scarce faults. In *Proceedings of the 3rd International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2012), 6–8 September, Naples, Italy*, volume 96 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–28, 2012.

[26] N. Immerman. Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences*, 22(3):65–72, 1981.

[27] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17:935–938, 1988.

[28] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):102–116, 2004.

[29] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4), 2009.

[30] O. Kupferman and M. Y. Vardi. Synthesis with incomplete informatio. In *In Advances in Temporal Logic*, pages 109–127. Kluwer Academic Publishers, 2000.

[31] B. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. *Acta Informatica*, 1984.

[32] D. Miller and R. Drechsler. Implementing a multiple-valued decision diagram package. In *The 28th Intrernational Symposium on multiple-valued logic (ISMVL)*, pages 52–57, 1998.

[33] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[34] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *16th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*, pages 179–190, 1989.

[35] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP)*, volume LNCS 372, pages 652–671, London, UK, 1989. Springer-Verlag.

[36] D. Pradhan. *Fault-tolerant computer system design*. Prentice-Hall, Inc., 1996.

[37] M. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the American Mathematical Society*, 141:1–35, 1969.

[38] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–42, 1990.

[39] J. Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In *FTRTFT*, pages 237–257, 1992.

[40] S. Schewe and B. Finkbeiner. Synthesis of asynchronous systems. In *LOPSTR*, pages 127–142, 2006.

[41] I. Sommerville. *Software engineering*. Addison-Wesley, 8th edition, 2007.

[42] W. Thomas. Finite-state strategies in regular infinite games. In *FSTTCS*, pages 149–158, 1994.

[43] F. Wang. Model-checking distributed real-time systems with states, events, and multiple fairness assumptions. In *International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume LNCS 3116. Springer-Verlag, 2004.

[44] F. WAng. Model-checking fair dense-time systems with propositions and events. *International Journal on Software Tools for Technology Transfer*, April 2014.