Monitoring Interfaces for Faults

Amir Pnueli\textsuperscript{1,2} Aleksandr Zaks\textsuperscript{1,2}

New York University

Lenore Zuck\textsuperscript{1,3}

University of Illinois at Chicago

Abstract

We consider the problem of a module interacting with an external interface (environment) where the interaction is expected to satisfy some system specification $\Phi$. While we have the full implementation details of the module, we are only given a partial external specification for the interface. The interface specification being partial (incomplete) means that the interface displays only a strict subset of the behaviors allowed by the interface specification.

Based on the assumption that interface specifications are typically incomplete, we address the question of whether we can tighten the interface specification into a strategy, consistent with the given partial specification, that will guarantee that all possible interactions resulting from possible behaviors of the module will satisfy the system specification $\Phi$. We refer to such a tighter specification as $\Phi$-guaranteeing specification. Rather than verifying whether the interface, which is often an off-the-shelf component, satisfies the tighter specification, the paper proposes a construction of a run-time monitor which continuously checks the existence of a $\Phi$-guaranteeing interface.

We view the module and the external interface as players in a 2-player game. The interface has a winning strategy if it can guarantee that no matter what the module does, the overall specification $\Phi$ is met. The problem of incomplete specifications is resolved by allowing the interface to follow any strategy consistent with the interface specification. Our approach essentially combines traditional run-time monitoring and static analysis. This allows going beyond the focus of traditional run-time monitoring tools – error detection in the execution trace, towards the focus of the static analysis – bug detection in the programs.

\textsuperscript{1} This research was supported in part by NSF grant CCR-0205571 and ONR grant N00014-99-1-0131.
\textsuperscript{2} Email: \{amir,zaks\}@cs.nyu.edu
\textsuperscript{3} Email: \{lenore@cs.uic.edu\}

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.nl/locate/entcs
1 Introduction

The process of constructing software is undergoing rapid changes. Instead of a monolithic software development within an organization, increasingly, software is being assembled using third-party components (e.g., JavaBeans, .NET, etc.). The developers have little knowledge of, and even less control over, the internals of the components comprising the overall system.

One obstacle to composing agents is that current formal methods are mainly concerned with “closed” systems that are built from the ground up. Such systems are fully under the control of the user. Hence, problems arising from ill-specified components can be resolved by a close inspection of the systems. When composing agents using “off-the-shelf” ones, this is often no longer the case. Out of consideration for proprietary information, or in order to simplify presentation, companies may provide incomplete specifications. Despite being ill-specified, “off-the-shelf” components might still be attractive enough so that the designer of a new service may wish to use them. In order to do so safely, the designer must be able to deal with the possibility that these components may exhibit undesired or unanticipated behavior, which could potentially compromise the correctness and security of the new system.

The main problem addressed in this paper is that of under-specification. As a simple example of the phenomenon, consider an interface specification that guarantees “after input query $q$ is received, output $r = \text{response}(q)$ is produced.” The designer of the interface probably meant a stronger specification, “after $q$ is received, nothing else is produced until $r$ is produced.” Assume that the later version is sufficient and necessary to ensure the correctness of the entire system consisting of the module and the interface. Applying formal methods, like model checking, would most likely fail since there is no algorithmic way to provide the model checker with the proper strengthening of the interface specification. Yet, under the assumption that interface specifications may be partial, there may exist a subset of the allowed behaviors that guarantees correctness, and one may still use the component, provided deviations of the interface from this “good” set of behaviors can be detected.

Assume that we are given:

- A finite-state module $M$, designed by our designer and accompanied by the full details of its implementation;
- An interface specification $\Phi_I$ for the external component interacting with the module $M$; and
- A goal specification $\Phi$ for the entire system which must be satisfied by the interaction between the module and the interface.

We view the module and interface as players in a 2-player game. At any stage in the computation we ask whether the interface has a strategy, consistent with its specification $\Phi_I$, such that for any possible behavior of the module $M$, the behavior of the system resulting from the interaction of the module
and the interface is guaranteed to satisfy the goal $\Phi$. We use this successive game solving as a basis for run-time monitoring of the system, where we raise an alarm as soon as the system reaches a state from which the interface can no longer guarantee that the system behaves correctly (i.e., satisfies $\Phi$).

A naive interpretation of the above description seems to imply that we solve a complete game after each move of either player. Such an implementation would make the process prohibitively expensive. Instead, we restrict our attention to games in which the winning condition is universal liveness – that is, closed under insertion and removal of arbitrary finite prefixes. For such games, it is sufficient to solve the game only once, and then just monitor the progress of the computation within the game structure. The single game solving process can be performed at compile-time, so there is very little to be done during the monitoring phase. In Section 3, we show that every game can be converted to a game with a universal liveness winning condition.

It is interesting to compare our methodology to the conventional run-time monitoring approach as the one described in [2,7]. As far as we know, all the traditional run-time monitoring systems to a large degree ignore the implementation details of the program under consideration and concentrate on analyzing a specific behavior. Such monitoring systems work especially well if one is mostly interested in certifying that the observed execution trace is error-free and, possibly, collecting some statistical information. However, the conventional approaches are usually unacceptable if the main goal is to find faults in the design itself – not just in a particular computation. In contrast, in our framework, in addition to the run-time information, we are trying to use all the available implementation details. That ultimately leads to a higher precision since we monitor not only the current trace, but considerably more. The idea is similar to the target enlargement [15] and is especially useful when debugging multi-threaded applications.

The paper is organized as follows. In Section 2, we describe our formal models for systems and games and show how a system can be associated with a game. In Section 3, we show how to solve such games, and Section 4 describes the construction of the monitor. In Section 5, we discuss various aspects of our methodology. Finally, in Section 6, we present our conclusions and some possible future research directions.

2 The Model

2.1 Fair Discrete Systems

We take a fair discrete system (FDS), which is a variant of fair transition system [12], as our computational model. Under this model, a system $M : \langle V, W, \Theta, \rho, J \rangle$ consists of the following components (for simplicity, we omit compassion):

- $V$: A finite set of system variables. A state of the system $S$ provides a type-consistent interpretation of the system variables $V$. For a state $s$ and
a variable \( v \in V \), we denote the value assigned to \( v \) by the state \( s \) by \( s[v] \).
Let \( \Sigma \) denote the set of all states over \( V \). We assume that \( \Sigma \) is finite.

- \( W \subseteq V \): A subset of owned variables which only the system itself can modify. All other variables can also be modified by the environment.
- \( \Theta \): The initial condition. A state is defined to be initial if it satisfies the assertion (state formula) \( \Theta \).
- \( \rho(V,V') \): The transition relation.
- \( J \): A set of justice (weak fairness) requirements (assertions).

For a subset of variables \( U \subseteq V \), we introduce the abbreviation \( \text{pres}(U) = \bigwedge_{u \in U} (u' = u) \), specifying a transition in which all the variables in \( U \) preserve their values. With no loss of generality, we assume that if \( s' \) is a \( \rho \)-successor of \( s \), then at least one owned variable changes its value between \( s \) and \( s' \), i.e., \( s'[W] \neq s[W] \). We define an extended transition relation: \( \rho^* = \rho \lor \text{pres}(W) \)

This extended relation allows, in addition to \( \rho \)-steps, also environment steps. Such steps are allowed to change all variables arbitrarily, as long as they preserve the values of all owned variables.

An open computation of an FDS \( S \) is an infinite sequence of states \( \sigma : s_0, s_1, s_2, ... \) satisfying the requirements:

- Initiality: \( s_0 \) is initial.
- Consecution: For each \( \ell = 0,1,... \), the state \( s_{\ell+1} \) is a \( \rho^* \)-successor of state \( s_\ell \). That is, \( \langle s_\ell, s_{\ell+1} \rangle \models \rho^*(V,V') \) where, for each \( v \in V \), we interpret \( v \) as \( s_\ell[v] \) and \( v' \) as \( s_{\ell+1}[v] \).
- Justice: For every \( J \in J \), \( \sigma \) contains infinitely many occurrences of \( J \)-states. In addition, we require that there exist infinitely many \( j \)'s such that \( s_j \) and \( s_{j+1} \) agree on the values of the owned variables. This guarantees that the environment is given infinitely many opportunities to take a step.

From now on, we will refer to an open computation simply as “computation”.

Given two FDS’s \( M_1 \) and \( M_2 \), the systems are compatible if their sets of owned variables are disjoint. If the systems are compatible, their asynchronous parallel composition, \( M_1 || M_2 \), is the FDS whose sets of variables, owned variables, and justice are the unions of the corresponding sets in the two systems, whose initial condition is the conjunction of the initial conditions, and whose transition relation is the disjunction of the two transition relations. Thus, a step in an execution of the composed system is a step of system \( M_1 \) or a step of system \( M_2 \).

All our concrete examples are given in SPL (Simple Programming Language), which is used to represent concurrent programs (e.g., [12,3]). Every SPL program can be compiled into an FDS in a straightforward manner. In particular, every statement in an SPL program contributes a disjunct to the transition relation. For example, the assignment statement “\( \ell_0: x := y + 1; \ell_1: \)” contributes to the transition relation, in the FDS that describes the program,
the disjunct $at_{\ell_0} \land at_{\ell_1} \land x' = y + 1 \land pres(V \setminus \{x, \pi\})$. The predicates $at_{\ell_0}$ and $at_{\ell_1}$ stand, respectively, for the assertions $\pi = 0$ and $\pi' = 1$, where $\pi$ is the control variable denoting the current location within the process to which the statement belongs (program counter).

2.2 Game Structures

Following [5], we define a (two-player) game $G = (S, A, \Gamma_1, \Gamma_2, \delta)$ to consist of:

- A set $S$ of states;
- A finite set $A$ of actions;
- Action assignment functions $\Gamma_1, \Gamma_2 : S \to 2^A \setminus \emptyset$ that define, for each state, a non-empty set of actions available to player-1 and player-2 respectively;
- A transition function $\delta : S \times A \times A \to S$ mapping each state $s$ and each pair of actions $(a_1, a_2) \in \Gamma_1(s) \times \Gamma_2(s)$ to a successor state $\delta(s, a_1, a_2)$;

From each state, the players simultaneously choose their actions. The two actions define the next state of the system.

Assume we are given a game $G$ as above. For $i \in \{1, 2\}$, a player-$i$ strategy is a function $\xi_i : S^+ \to A$ that maps every nonempty finite sequence $\bar{s} \in S^+$ to a single action that is consistent with $\Gamma$, (i.e., for every $\bar{s} \in S^+$ and $s \in S$, $\xi_i(\bar{s}; s) \in \Gamma_i(s)$). The set of strategies for player-$i$ is denoted by $\Xi_i$.

Given a game structure $G$, a run $r$ of $G$ is a nonempty, possibly infinite, sequence $s_0(a_0^0, a_0^1)s_1(a_1^0, a_1^1)s_2\ldots$ of alternating states and action pairs such that, for every $j \geq 0$ and $i \in \{1, 2\}$, $a_i^j \in \Gamma_i(s_j)$ and $s_{j+1} = \delta(s_j, a_1^j, a_2^j)$. For a run $r : s_0(a_0^0, a_0^1)s_1(a_1^0, a_1^1)s_2\ldots$, we refer to the state sequence $\sigma(r) : s_0, s_1, s_2, \ldots$ as the history induced by $r$. Given a pair of strategies $\xi_1 \in \Xi_1$ and $\xi_2 \in \Xi_2$ and a state $s \in S$, the outcome of the strategies from $s$, $R_{\xi_1, \xi_2}(s)$, is a run that starts in $s$ and whose actions are consistent with the strategies.

Let $h : s_0, s_1, \ldots, s_k = s$ be a finite history and $\Psi$ a linear temporal logic formula over $S$. History $h$ is said to be a winning history for player-$i$, $i \in \{1, 2\}$, with respect to objective $\Psi$ in $G$ if player-$i$ has a strategy $\xi_i \in \Xi_i$ such that for all strategies $\xi_{3-i} \in \Xi_{3-i}$, $h \cdot \sigma(R_{\xi_i, \xi_{3-i}}(s)) \models \Psi$. A suitable strategy $\xi_i$ is a winning player-$i$ strategy for $\Psi$ from $h$ in $G$. In case a winning history $h$ consists of the single state $s$, we refer to $s$ as a player-$i$ winning state.

**Example 1** Let $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$ and $A = \{a, b, c\}$. Let $\Gamma_1, \Gamma_2$, and $\delta$ be defined as follows:

\[
\begin{align*}
\Gamma_1(s_0) &= \{c\}; & \Gamma_2(s_0) &= \{a, b\}; & \delta(s_0, c, a) &= s_1; & \delta(s_0, c, b) &= s_2; \\
\Gamma_1(s_3) &= \{a, b\}; & \Gamma_2(s_3) &= \{c\}; & \delta(s_3, a, c) &= s_4; & \delta(s_3, b, c) &= s_5.
\end{align*}
\]

For a state $s \in \{s_1, s_2, s_4, s_5\}$, $\Gamma_1(s) = \Gamma_2(s) = a$. For $j \in \{1, 2\}$, $\delta(s_j, c, c) = s_3$, and for $j \in \{4, 5\}$, $\delta(s_j, c, c) = s_j$. The corresponding game structure is shown in Fig. 1. Note that player-2 fully controls the transitions out of $s_0$. Meaning, that when the game is at $s_0$, player-2 can decide whether the next
state will be $s_1$ or $s_2$. In a similar way, player-1 controls the exits out of $s_3$.

![Game structure](image)

Fig. 1. A Game structure for Example 1.

The objective of the game is defined as:

$$
\Psi = \Box((s_1 \rightarrow \Box s_3 \land \Box \Box s_4) \land (s_2 \rightarrow \Box s_3 \land \Box \Box s_5)).
$$

The objective requires that any visit to $s_1$ should be immediately followed by a subsequent visit to $s_3$, which in turn should be immediately followed by a visit to $s_4$, and similarly, $s_2$ should be followed by visits to $s_3$ and then to $s_5$.

In this game, states $\{s_3, s_4, s_5\}$ are winning for both players for $\Psi$. This is because no path from any of these states leads to either $s_1$ or $s_2$. The other states – $s_0, s_1, and s_2$ – are winning only for player-1 which, starting at any of these states, has a strategy that guarantees $\Psi$. Note that the winning strategy starting at $s_0$ depends, when we reach $s_3$, on the path leading to $s_3$, that is, on whether the previous state is $s_1$ or $s_2$. Examples of winning non-singleton histories are $h : s_0, s_1, s_3$ and $h' : s_0, s_2, s_3$, which are winning for player-1.

2.3 Associating Games with FDS’s

Given an FDS $M = (V_M, W, \Theta, \rho, J)$ that corresponds to some SPL module, a system specification $\Phi$, and an interface specification $\Phi_I$, we define a game $G = (S, A, \Gamma_1, \Gamma_2, \delta)$ between the module $M$ and the interface as follows.

Let $V = V_M$ augmented with a variable $\text{turn} \in \{1, 2\}$ that is not in $V_M$. Let $S$ be the set of all $V$-states. The set $A$ of the G’s actions is $S$ itself. In the game $G$, player-1 (the module) can take any step that is allowed by $M$, non-deterministically setting $\text{turn}$, thus deciding whether or not it wishes to take another step. Player-2 (the interface) can set any of the variables that are not owned by $M$, but it has to let Player-1 take the next step. Formally:

- $\Gamma_1(s) = \{s' \mid (s, s') \models \rho\}$
- $\Gamma_2(s) = \{s' \mid (s, s') \models (\text{turn}' = 1 \land \bigvee_{v \in V_M \setminus W} \text{pres}(V \setminus \{v, \text{turn}\}))\}$
- $\delta(s, a_1, a_2) = a_{s[\text{turn}]}$. That is, $\delta$ selects $a_1$ as the next state iff $\text{turn} = 1$ in the current state $s$.

The objective of the game is defined by

$$
\Psi = (\Phi \land \Phi_I) \lor \Box (\text{turn} = 1) \lor \bigvee_{J \in \mathcal{J}}(\Box \Box \neg J).
$$

Thus, the game is won by either meeting both $\Phi$ and $\Phi_I$, violating one of the justice requirements, or preventing the interface from taking infinitely many steps. We force the module to give up its turn infinitely many times to preserve the semantics of interleaving.
2.4 Rabin-Chain Automata

Assume an FDS $M = (V, W, \Theta, \rho, \mathcal{J})$. We refer to the interpretations of $V$ as computation states. A deterministic total Rabin-chain automaton of index $k$ over a set of computation states $S$ is a tuple $R = (Q, q_0, \Delta, c)$ where:

- $Q$ is a finite set of automaton states;
- $q_0 \in Q$ is the initial state;
- $\Delta : Q \times S \rightarrow Q$ is a transition function;
- $c : Q \rightarrow \{0, \ldots, 2k - 1\}$ is a coloring function.

A run of $R$ over an infinite computation $\sigma : s_0, s_1, s_2, \ldots$ is an infinite sequence $q_0, q_1, q_2, \ldots$ where $q_0$ is the initial automaton state and, for every $i \geq 0$, $\Delta(q_i, s_i) = q_{i+1}$. We say that $q_0, q_1, q_2, \ldots$ is the run induced by the computation $\sigma$. The run $q_0, q_1, \ldots$ is accepting if the maximal color that appears infinitely many times in the color sequence $c(q_0), c(q_1), \ldots$ is even. A computation $\sigma$ is accepted by the automaton $R$ if the run induced by $\sigma$ is accepting. The ($\omega$-) language of $R$, denoted by $L(R)$, is the set of computations accepted by $R$. We say that the automaton $R$ accepts the temporal formula $\varphi$ if $L(R)$ is exactly the set of computations satisfying $\varphi$.

**Example 2** In Fig. 2, we present a Rabin-chain Automaton for the LTL formula $\Psi = \square((s_1 \rightarrow \circ s_3 \land \circ \circ s_4) \land (s_2 \rightarrow \circ s_3 \land \circ \circ s_5))$.

![Fig. 2. A Rabin-chain automaton for the objective $\Psi$ of the game in Example 1](image)

The automaton has the set of states $Q = \{q_0, q_1, q_2, q_3, q_4, q_r\}$. We connect automaton state $q_i$ to $q_j$ by an edge labeled by $s$ to represent the transition function entry $\Delta(q_i, s) = q_j$. To simplify the presentation, we do not explicitly label the dashed edges which connect states to the special rejecting state $q_r$.

By convention, the dashed edges are implicitly assumed to be labeled by $S$-states that do not label other edges departing from the same automaton state. Finally, the coloring function $c$ is given by

$$c(q_0) = c(q_1) = c(q_2) = c(q_3) = c(q_4) = 0, \quad c(q_r) = 1.$$  

3 Solving Games

Our approach to run-time monitoring is based on the observation that when the interface does not have a winning strategy a violation of the specification is unpreventable. Therefore the monitoring algorithm traces the interaction
between module and interface and raises an alarm at the first time it detects that the interface no longer has a winning strategy. In Fig. 3, we present a naive implementation of this idea.

- Check whether the initial state is winning for the interface (player-2) in game $G$. If it is not, then raise an alarm.
- In all subsequent steps, let $h$ be the history observed since the beginning of the computation. If $h$ is not a winning for the interface, raise an alarm.

Fig. 3. A naive algorithm for prevention maintenance

While the algorithm in Fig. 3 fully captures the spirit of our monitoring approach, it is computationally unacceptable. This is because it implies that we have to analyze the game each time afresh, with respect to an unbounded set of possible histories that may arise during computation.

Note that the naive algorithm is feasible if we can partition the states of the game into good and bad ones so that, regardless of the history of the game, the interface can win for $\Psi$ when the game is in a good state, and, similarly, the module can win for $\neg \Psi$ from a bad state. Formally, we define a state $s$ to be good for player-$i$ with respect to the objective $\Psi$ if:

- State $s$ is good for player-$(3-i)$ with respect to the objective $\neg \Psi$.

For example, states $\{s_0, s_1, s_2, s_3\}$ of the game of Fig. 1 are good for player-1 (w.r.t $\Psi$). These are the only states which are good for any of the players in this game. For example, the question whether state $s_4$ is winning depends on the path by which we reached $s_4$. If the path went through $s_1$, then $s_4$ is winning. If the path went through $s_2$, then $s_4$ is not a winning state. From here on, we apply the terms good and bad only with respect to player-2 (interface) and objective $\Psi$. A game $G$ is called partitionable if every state $s$ is either bad or good.

In case a game $G$ is partitionable, it is easy to construct a monitor. First, we solve a game $G$ by finding all good states. Note that in a partitionable game, the notions of a good state and a winning state coincide. Therefore, we can use the algorithm presented in [5] to find all player-2 winning states to solve a game. At run-time, we just need to make sure that the game doesn’t enter a bad state and raise an alarm if it does.

Unfortunately, it is not always the case that a state $s$ can be identified as good or bad. Note that a game associated with an FDS can be easily represented as a turn based game with a Borel winning condition, which is known to be determinate [4]. Determinacy guarantees that for each particular history $h$ either player-1 can win for $\Psi$ or player-2 can win for $\neg \Psi$, which is not true in general for the concurrent games described in Section 2.2 [6]. Therefore, the only reason why we cannot always partition the states is that
for some states player-2 has a winning strategy for a history \( h \), but not for some other history \( h' \). This was the case, as shown above, for state \( s_4 \). Therefore, whenever the game reaches \( s_4 \) a monitor cannot immediately decide whether it should raise an alarm. We can, in principle, make a right decision by taking a closer look at the history of the game and using a variation of an algorithm that computes winning states. However, that would call for a fresh game analysis on each visit to \( s_4 \). Clearly, that would make monitoring too expensive.

To solve the problem, we characterize the games for which it is possible to partition the states into good and bad. An LTL formula \( \Omega \) represents a universal liveness property if the following holds:

For every \( \sigma_1 \in \Sigma^* \) and \( \sigma_2 \in \Sigma^\omega \), \( \sigma_1 \cdot \sigma_2 \models \Omega \) iff \( \sigma_2 \models \Omega \).

Note that absolute liveness [1] satisfies only one direction of this definition, (i.e., if \( \sigma_2 \models \Omega \) then \( \sigma_1 \cdot \sigma_2 \models \Omega \)). The notion of universal liveness has been considered in [14] under the name of fairness. Also, it should be stressed that universal liveness should not be confused with the concept of memoryless strategies (i.e strategies such that the choice of an action only depends on the last state of a game history). Indeed, a winning strategy for a game with a universal liveness winning condition may require memory and vice versa.

It can be shown that if the objective \( \Psi \) of game \( G \) is a universal liveness property, then \( G \) is a partitionable game. Intuitively, if there is a winning strategy for some history \( h \), we can replace it by any other history \( h' \), sharing the same last state. Therefore, there can be no state from which player-2 has a winning strategy only with respect to some history \( h \) but not with respect to some other history \( h' \) sharing the same last state.

In the remainder of this section, we will show how to transform an arbitrary game \( G \) to an equivalent game \( G' \) with an objective \( \Psi' \) that represents a universal liveness property. Essentially, we split undecided states like \( s_4 \) in Fig. 1 so that the new states can be identified as good or bad.

### 3.1 Converting the objective of a game into universal liveness

Given a game structure \( G = (S, A, \Gamma_1, \Gamma_2, \delta) \) and objective \( \Psi \), we first build a Büchi automaton that accepts \( \Psi \) (see, e.g., [8]). Next we use the construction of [13] to build a total deterministic Rabin-chain automaton \( R = (Q, q_0, \Delta, c) \) over \( S \) such that accepts \( \Psi \).

The composition of the game structure \( G \) with the Rabin chain automaton \( R \) is the game \( G \times R = (S', A', \Gamma'_1, \Gamma'_2, \delta') \), where:

- \( S' = S \times Q \);
- \( A' = A \);
- \( \Gamma'_i((s, q)) = \Gamma_i(s) \) for \( i = 1, 2 \);
- \( \delta'((s, q), a_1, a_2) = (\delta(s, a_1, a_2), \Delta(q, s)) \).

It is straightforward to convert the acceptance condition of \( R \) into an LTL
objective $\Psi'$ for the game $G' = G \times R$. Indeed, we can define $\Psi'$ as:

$$\Psi' = \bigvee_{i=0}^{k-1}(\Box \lozenge (c = 2i) \land \Box \square (c \leq 2i)),$$

where $c$ is interpreted in a state $(s, q)$ as $c(q)$. Since the formula $\Psi'$ consists of a boolean combination of formulas of the form $\Box \lozenge p$ and $\Box \square p$ for assertions $p$, it is easy to show that $\Psi'$ represents a universal liveness property. As mentioned before, we can solve $G'$ using the algorithm from [5].

**Example 3** Consider the game structure in Fig. 1. The automaton for the objective $\Psi$ is given in Fig. 2, and the composed game is outlined in Fig. 4.

Fig. 4. The composed game

The only good states in the composed game are $(s_4, q_0)$ and $(s_5, q_0)$. All other states are bad. Thus, as desired, this composed game with universal liveness objective is partitionable.

### 3.2 Deterministic Büchi Specifications

The construction above is quite expensive. The size of the resulting game is doubly exponential in the length of the original objective $\Psi$. In addition, the size of the new objective $\Psi'$ is exponential in the length of $\Psi$. An important and frequently occurring special case is when the specification $\Phi \land \Phi_I$ can be represented as a deterministic Büchi automaton. In this case, we can skip the expensive determinization step. We present an efficient algorithm for monitoring of such special cases.

As defined in Section 2.3, the objective of our game $G$ is defined by

$$\Psi = (\Phi \land \Phi_I) \lor \lozenge \Box \square (\text{turn} = 1) \lor \bigvee_{J \in \mathcal{J}}(\lozenge \Box \neg J).$$

Recall that to solve a game for the purpose of run-time monitoring, we need to use a partitionable game. In Section 3.1, we have presented a general methodology for transforming an arbitrary game into a game with universal liveness objective. Here we consider the special case that $(\Phi \land \Phi_I)$ can be represented by a deterministic Büchi automaton. We proceed as follows:

- Build a deterministic Büchi automaton $B$ that accepts $(\Phi \land \Phi_I)$. Let $\mathcal{F}$ represent the accepting set.
- Compose the automaton $B$ with the game $G$ to obtain $G' = G \times B$ as in Section 3.1. Let the objective of the resulting game $G'$ be defined by

$$\Box \lozenge \mathcal{F} \lor \Box \square (\text{turn} = 1) \lor \bigvee_{J \in \mathcal{J}}(\lozenge \Box \neg J).$$
Consider an LTL objective $\Phi$ in the form:

$$\Box \Diamond p \lor \bigvee_{i=1}^{n}(\bigcirc \Box r_i),$$

where $p, r_1, \ldots, r_n$ are assertions. Clearly, the objective of $G'$ has such a form. Let $Win$ be a set of player-2 winning states for $\Phi$. Following [9], we can compute $Win$ as follows:

$$Win = \nu Z.\mu Y.\left(\bigvee_{i=1}^{n}(\nu X.(p \land \bigcirc Z) \lor \bigcirc Y \lor (r_i \land \bigcirc X))\right),$$

where $[[\bigcirc f]]_G^e_G = \{s \in S \mid \exists b \in \Gamma_2(s).\forall a \in \Gamma_1(s). \delta(s, a, b) \in [[f]]_G^e_G\}$. That is, $[[\bigcirc f]]_G^e_G$ characterizes the set of states from which player-2 can force the game to move to a state belonging to $[[f]]_G^e_G$ in one step. For example, applying $\Box$ to the set $\{s_1, s_2\}$, in Example 1, yields $\{s_0\}$.

The above formula can be evaluated symbolically, requiring at most $|S|^2$ steps, in spite of the fact that it has three alternating fix-point operators [10].

### 4 Feasible Interface Monitoring

Assume a module $M$, a specification $\Phi$, and an interface specification $\Phi_I$. A finite prefix $\sigma$ of $M$-states is safe with respect to $M$, $\Phi$, and $\Phi_I$ if there exists an FDS $I^*$ (a possible interface), such that

- $\sigma$ is a prefix of some computation of $M \parallel I^*$;
- For every computation $\sigma'$ of $M \parallel I^*$ that has $\sigma$ for a prefix, $\sigma' \models \Phi \land \Phi_I$.

The FDS $I^*$ can be viewed as a concrete implementation of a winning strategy of player-2 (the interface). In fact, it can be shown that every such interface induces a winning strategy applicable after observing $\sigma$.

Based on the discussion in the preceding sections, we can now formulate a more efficient version of run-time monitoring process, in which we analyze the game only once — prior to the beginning of the monitored production run. Using the methods of the previous section, we construct a partitionable game $G$ which represents the possible interaction between the module and the interface, while assessing whether this interaction satisfies the conjunction $\Phi \land \Phi_I$ and the relevant fairness requirements of the module and proper interleaving between module and interface. Since $G$ is partitionable, we will use the terms “winning” and “good” interchangeably. Let $Init$ be the set of states that satisfy the initial condition $\Theta$ of $M$ and any initial conditions induced by any automaton that may have been combined into $G$. A sketch of a feasible monitoring algorithm is presented in Fig. 5.

The following theorems state the soundness of Algorithm $mon$.

**Theorem 4.1** If $Init \not\subseteq Win$, then $M$ is incompatible with $\Phi \land \Phi_I$. That is, there exists no interface $I^*$ such that $M \parallel I^* \models \Phi \land \Phi_I$.

**Theorem 4.2** Algorithm $mon$ alerts only after observing an unsafe history.
Prior to the monitored production run, analyze the game $G$, computing the set $\text{Win}$ of states which are winning for the interface. If $\text{Init} \not\subseteq \text{Win}$, then raise an alarm.

Start the production run. For every observed finite prefix $\sigma$ of $M$-states, let $h$ be the corresponding history of $G$, induced by the $\sigma$. Let $s$ be the last state of $h$. If $s \not\in \text{Win}$, then raise an alarm.

Fig. 5. A feasible algorithm $\text{mon}$ for prevention maintenance

The proof of both theorems is based on the observation that if a state $s$, reachable by history $h$, is not winning for the interface, then there exists no interface $I^*$ which, when run in parallel with $M$, can guarantee that all continuations of $h$ will not violate $\Phi \wedge \Phi_I$.

If there existed such an interface, we could derive from it a strategy which is winning for the interface, contradicting the fact that the monitor observed a state which is bad for the interface. Thus, if an alarm is raised, there is no way to prevent a computation which violates $\Psi$.

It remains to show that every infinite history $h$ violating $\Psi$ induced by a run of a game is an open computation of $M$ violating $\Phi \wedge \Phi_I$. Indeed, consider a history $h$ which violates the objective $(\Phi \wedge \Phi_I) \lor \Diamond \Box (\text{turn} = 1) \lor \bigvee_{J \in J}(\Diamond \Box \neg J)$. Violation of $\bigvee_{J \in J}(\Diamond \Box \neg J)$ guarantees that $h$ satisfies all the fairness requirement of $M$. Violation of the conjunct $\Diamond \Box (\text{turn} = 1)$ guarantees that $h$ contains infinitely many interface steps. Thus, $h$ is an open computation of $M$ violating $\Phi \wedge \Phi_I$.

Note that since in the absence of design faults all prefixes are safe, Theorem 4.2 implies that if an alert is generated, there is a bug in the system. Since we are interested in finding faults, the above statement signifies soundness of our method. In contrast, in model checking, the above statement usually means completeness since the goal is to prove program correctness.

Example 4 Consider the FDS corresponding to the SPL module in Fig. 6(a). Assume the specification is $\Phi : \Box \Diamond \text{at}_{-\ell_0} \land \Box \Diamond \text{flag}_2$ and the interface specification is the trivial $\Phi_I : \text{true}$. In Fig. 6(b) we present an SPL program for an interface $I^*$ such that $M \parallel I^* \models \Phi \wedge \Phi_I$. Therefore, the module is compatible with $\Phi \wedge \Phi_I$. As was mentioned before, we can view $I^*$ as a concrete realization of a winning strategy for the interface. Consider a state $\text{at}_{-\ell_2} \land \text{flag}_1 \land \text{flag}_2$. It is easy to verify that it is a bad state, assuming it is the module's turn. Consequently, our monitor will raise an alarm when such state is reached. The alarm is really an early warning. Although the violation is unpreventable, it would take infinitely many steps to confirm the violation.

Besides catching the problem early on, there is another significant advantage of our approach – the fact that we were able to identify the problem at all. There is no way to identify a violation of a liveness property, like $\Phi : \Box \Diamond \text{at}_{-\ell_0} \land \Box \Diamond \text{flag}_2$, using traditional run-time monitoring unless the program terminates. Of course, we cannot always catch a violation of liveness.
either, but, sometimes, we can catch an eventual violation, as we did in this example. Moreover, let’s modify $\Phi$ to be $\Box \neg \text{at}_{\ell_4}$. Even in that case, there is no guarantee that a user would be alerted when traditional run-time monitoring is employed since there is still a chance that the specification will hold. Indeed, the module can give up the turn and the interface can reset the $\text{flag}_2$ to false, which would restore the game into a good state.

![Figure 6. Module and a possible interface for Example 4](image)

### 5 Discussion

**Soundness and Completeness of the Method** Our method is sound, meaning that whenever $\text{mon}$ raises an alert, there is a bug in the implementation of the module and/or external components. However, as we mentioned before, an execution trace that generates an alert is not actually bound to violate specification. Nevertheless, an alert should be treated with the same amount of respect as a negative result of model checking. Of course, in practice, it might be desirable to distinguish when there is just a bug in the program or when a bug will cause the currently observed execution to violate specification. To accomplish this, we construct a game as before but let the interface make choices not only for itself but also for the module. If the interface cannot win even under these relaxed conditions, then a violation is guaranteed to occur whenever an alert is generated.

We also need to mention incompleteness of algorithm $\text{mon}$. Meaning that the existence of a winning strategy for the interface does not necessarily imply that this strategy can be implemented by an SPL program. It is mainly due to the following two reasons. First, we allow the interface to observe all the variables of $M$, even the local ones. In addition, the interface has access to the complete history of a computation. The practical effect of incompleteness is that a monitor may not spot a bug as early as it could have done; sometimes it may not deduce a bug at all. However, it should be stressed that as soon as a computation violates specification an alert will be generated. Therefore, in this respect, $\text{mon}$ is always better than traditional run-time monitoring tools.
Note that incompleteness of algorithm mon should not be confused with the following fact. The absence of an alarm does not guarantee that the computation actually satisfies the objective if we let the game continue forever. Just because the game stays within good states, the system may not be getting any closer to satisfying its liveness properties. Unfortunately, any sound run-time monitoring system has the above characteristic.

**State Explosion** Consider the formula for computing the winning states at the end of Section 3.2. The amount of work needed to compute the set $Win$ is comparable to model checking the module in the presence of fairness. Both require handling at least two alternating fix-point operators. Consequently, both suffer from state explosion problem. Fortunately, the parallel with model checking does not stop here, and we can apply some popular remedies used for model checking. In particular, one can abstract the module before applying run-time monitoring. To preserve the soundness of the method, we can allow the interface to resolve all the non-deterministic choices caused by abstraction. Of course, this may adversely effect completeness.

Interestingly enough, one of the reasons to use run-time monitoring in the first place may be to cope with the complexity of model checking. Indeed, we can label parts of the program that are hard to deal with as external components and apply run-time monitoring. Since we allow specifications given as Rabin-chain automaton, one can abstract to any degree that is necessary. In the extreme case when specification of the external components matches their implementation, run-time monitoring is degraded to regular model checking.

### 6 Our Contribution and Related Work

Game-theoretic formalisms have been widely applied for solving various verification related problems. For example, a problem of verification and synthesis of open systems is studied in [11] and is closely related to our work. However, as far as we know, we are the first to utilize game-theoretic approach for run-time monitoring. In addition, we have identified and proposed a solution for a problem of solving games in the presence of dynamic information.

We have already mentioned some differences between our work and "traditional" approaches to run-time monitoring. Most importantly, we are pursuing different goals: trace monitoring vs fault discovering. Therefore, we largely view our work as being orthogonal to the existing methods. However, even if we concentrate on a specific behavior like traditional monitors do, our approach offers several advantages. First, we are able to deal with liveness properties, raising an alarm when a violation of a liveness property can no longer be prevented with absolute assurance by the interface. Furthermore, in most cases, we can detect violations long before they actually happen. One such case is when an individual thread in a multi-threaded application performs a sequence of local computations, which are usually deterministic. If something is about to go wrong during this time, an alert is generated be-
fore any actual computations take place. This information can be helpful in a number of ways, for example, it might be prudent to increase the level of logging to facilitate the debugging/recovery process later on.

References

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct 1985.

[2] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proc. VMCAI’04*, LNCS 2937, pages 44–57, 2004.

[3] N. Bjørner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In *Proc. CAV’96*, LNCS 1102, pages 415–418. Springer Verlag.

[4] Morton Davis. Infinite games of perfect information. In *Advances in game theory*, pages 85–101. Princeton Univ. Press, Princeton, N.J., 1964.

[5] L. de Alfaro, T. Henzinger, and R. Majumdar. From verification to control: dynamic programs for omega-regular objectives. In *Proc. LICS ’01*, pages 279–290, 2001.

[6] L. de Alfaro and T.A. Henzinger. Concurrent omega-regular games. In *Proc. LICS’00*, pages 141–154. IEEE Computer Society Press, 2000.

[7] D. Drusinsky and M.T. Shing. Monitoring Temporal Logic Specifications Combined with Time Series Constraints. *JUCS*, 9(11):1261–1276, 2003.

[8] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. PSTV’95*, pages 3–18, 1995.

[9] Yonit Kesten, Nir Piterman, and Amir Pnueli. Bridging the Gap between Fair Simulation and Trace Inclusion. In *Proc. CAV’03*, LNCS 2725. Springer, 2003.

[10] D. Long, A. Browne, E. Clarke, S. Jha, and W. Marrero. An improved Algorithm for the Evaluation of Fixpoint expressions. In *Proc. CAV’94*, LNCS 818, pages 338–350. Springer, 1994.

[11] F.Y.C. Mang. *Games in Open Systems Verification and Synthesis*. PhD thesis, University of California, Berkeley, 2002.

[12] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[13] S. Safra. On the complexity of ω-automata. In *Proc. FOCS’88*, pages 319–327, White Plains, NY, 1988.

[14] A.P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6:495–511, 1994.

[15] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proc. DAC’98*, June 1998. San Francisco, CA.